USENIX

Proceedings of LISA '10: 24th Large Installation System Administration Conference

San Jose, California, November 7–12, 2010

conference

proceedings

LISA '10: 24th Large Installation System Administration Conference

San Jose, California
November 7–12, 2010

Sponsored by

USENIX

in cooperation with
LOPSA & SNIA

ISBN 978-931971-78-2

USENIX Association

Proceedings of LISA ’10: 24th Large Installation System Administration Conference

November 7–12, 2010

San Jose, California

Conference Organizers

LISA '10:
24th Large Installation System Administration Conference
November 7–12, 2010
San Jose, California

Message from the Program Chair

Dear LISA Attendee,

Technology is advancing fast, and we all need to be on top of it. Our job is to design, build, and maintain it to serve business needs. Our job is a highly dynamic one, not only due to rapid changes in technology itself, but also because the way that technology is used or delivered changes as well. For example, new ways of computing mean that many of us are busily moving applications to the cloud or from machines that run multiple applications to virtualized boxes or appliances just running one task.

These developments have a huge impact on the sysadmin profession, both in a technical and a nontechnical fashion. It is important for us to have a venue to continuously learn, share experiences, and develop ourselves and our skills. LISA is there for you and by you to serve these needs. In addition to attending presentations, please get completely immersed and experience the hallway track and the Birds-of-a-Feather sessions. This is the place to discuss tech in general, as well as to present your issues in the workplace (tech and nontech alike) and help others with theirs.

LISA has come a long way from the small workshops of the 1980s. It has always been the place for sysadmins, but it has matured into one of the flagship conferences USENIX organizes. The week is packed full: 3 days of tech sessions, 6 days of training comprising 48 classes led by world experts, 7 workshops, 2 poster sessions, and numerous other sessions and events. Be proud that *you* are part of this event!

Organizing such a conference takes about a year and involves over 200 people. All have their own important parts to play in building the event. My job was just being one of those people. It happened to be labeled “Program Chair,” but the credit goes to all the others, who are just as important as “the chair.” Please give them a huge *thank you*.

This year we decided to add a second track, practice and experience reports, in addition to the regular refereed papers track. We did this because we want to encourage people to share experiences and best practices in a more formal way. We wanted the reports to include a small write-up, so that there would be a record of what was presented. Looking at the submission numbers, we find that this idea worked out quite well. Of the total of 63 submissions, we had 45 regular papers and 18 experience reports. In a day-long meeting the program committee accepted 18 papers and 9 experience reports. I hope you like the selection and take home some good experiences and practical ideas. Also, please think about submitting something yourself for next year!

As a last word, I would thank you all for coming. Be sure to enjoy yourselves and have as much fun at the conference as I did in helping organizing it.

Rudi van Drunen, Competa IT and Xlexit Technology, The Netherlands
Program Chair

A survey of system configuration tools

Thomas Delaet *Wouter Joosen*
Bart Vanbrabant
DistriNet, Dept. of Computer Science
K.U.Leuven, Belgium
{*thomas.delaet, wouter.joosen, bart.vanbrabant*}*@cs.kuleuven.be*

Abstract

We believe that informed choices are better choices. When you adopt a system configuration tool, it implies a significant investment in time and/or money. Before making such an investment, you want to pick the right tool for your environment. Therefore, you want to compare tools with each other before making a decision. To help you make an informed choice, we develop a comparison framework for system configuration tools. We evaluate 11 existing open-source and commercial system configuration tools with this framework. If you use our framework, you will make a better choice in less time.

1 Introduction

When you adopt a system configuration tool, it implies a significant investment in time and/or money. Before making such an investment, you want to know you have picked the right tool for you environment. Therefore, you want to compare tools with each other before making a decision.

Since there exist a lot of tools with different goals, characteristics and target users, it is a difficult and time-intensive task to make an objective comparison of system configuration tools. Moreover, people using a tool already made a significant investment in that tool (and not others) and as a consequence are involved in that tool. But they themselves have difficulty comparing their “own” tool to other tools.

To help you make an informed choice, we developed a comparison framework for system configuration tools. In addition to more subjective or political decision factors, this framework can help you with the more objective factors when selecting a system configuration tool that is right for you. The framework consists of four categories of properties.

1. Properties related to the input specification
2. Properties related to deploying the input specification
3. Process-oriented properties
4. Tool support properties

We evaluated 11 existing open-source and commercial system configuration tools with our framework. This paper contains a summary of these evaluations. The full evaluations are available on our website at `http://distrinet.cs.kuleuven.be/software/sysconfigtools`. You can comment on these evaluations, provide suggestions for modifications or add your own evaluations.

The remainder of this paper is structured as follows: We start with the description of the framework in Section 2. Next, we summarize our findings for the 11 tools we evaluated in Section 3. Section 4 answers the questions on how to choose a tool and how to evaluate another tool using the framework. In Section 5, we use our framework and the evaluations to analyze the gaps in the state of the art. Section 6 concludes the paper.

2 The comparison framework

Every system configuration tool provides an interface to the system administrator. Within this interface, the system administrator expresses the configuration of the devices managed by the tool. The tool uses this specification as input and enforces it on all machines it manages. This conceptual architecture of a system configuration tool is illustrated in Figure 1.

In Figure 1, the system administrators inputs the desired configuration of the devices managed by the tool. This input it stored in a repository. The tool uses this input to generate device-specific profiles that are enforced on every managed device. The translation agent is the component of the tool that translates the system administrator input to device-specific profiles. The deployment agent is the component of the tool that runs on the managed device and executes the generated profile.

Our comparison framework contains properties for both the specification of the input and the enforcement phase. The third type of properties that are present in our comparison framework are meta-specification properties: how does a tool deal with managing the input specification itself? The last type of properties deal with tool support: How easy is it to adopt the tool?

2.1 Specification properties

2.1.1 Specification paradigm

We define the specification paradigm of a tool by answering two questions:

1. Is the input language declarative or imperative?

2. Does the tool use a GUI-based or command-line user interface?

Tools that use a declarative input language enable to express the desired state of the computer infrastructure. The runtime of the tool compares this desired state with the configuration on every managed device and derives a plan to move to the desired state. In the system configuration literature, this process is described as *convergence* [1]. A system configuration tool that supports *convergence* has the additional benefit that divergences from the desired state are automatically corrected.

Tools that use an imperative input language distribute, schedule and deploy scripts written in its imperative input language on the managed devices. For an imperative script to work reliable, all possible states of the managed devices need to covered and checked in the script. Moreover, the system configuration tool must also keep track of what scripts are already executed on every device. An alternative is to make all the operations in the script idempotent.

Let us contrast the practical differences between an imperative and a declarative language. Suppose a system administrator does not want file `/etc/hosts_deny` to be present on a device.

In a declarative language, the system administrator must ensure that the file is not included in the model or explicitly define that the file must not exist.

In an imperative language, the system administrator must first write a test to verify if `/etc/hosts_deny` exists. If the file exists, another instruction is needed to remove the file. If the system administrator does not write the first test, the action fails if the file was already removed.

Orthogonal on the choice of declarative or imperative specification language is the choice of user interface: does the tool use a command-line or graphical user interface?

Command-line interfaces typically have a steeper learning curve than graphical approaches but, once mastered, can result in higher productivity. Command-line interfaces also have the advantage that they can be integrated with other tools through scripting. In contrast, system administrators are typically quicker up to speed with graphical approaches [12].

2.1.2 Abstraction mechanisms

A successful configuration tool is able to make abstraction of the complexity and the heterogeneity that characterises IT infrastructures where hardware and software of several vendors and generations are used simultaneously [3]. Making abstraction of complexity and heterogeneity is very similar to what general purpose programming languages have been doing for decades.

Abstraction from complexity is an important concept in programming paradigms such as object orientation. In object orientation, implementation details are encapsulated behind a clearly defined API. Encapsulation is a concept that is valuable for modeling configurations as well. Responsibilities and expertise in a team of system administrators are not defined on machine boundaries, but based on subsystems or services within the infrastructure, for example: DNS or the network layer. Encapsulation enables experts to model an aspect of the configuration and expose a well documented API to other system administrators.

Modern IT infrastructures are very heterogeneous environments. Multiple generations of software and hardware of several vendors are used in production at the same time. These heterogeneous "items" need to be configured to work together in one infrastructure.

Based on how a system configuration tool's language deals with complexity and heterogeneity, we define six levels to classify the tool. These levels range from high-level end-to-end requirements, to low-level bit-configurations. [3] inspired us in the definition of these levels.

1. **End-to-end requirements**: End-to-end requirements are typical non-functional requirements [23]. They describe service characteristics that the computing infrastructure must achieve. Figure 2 shows an example of a performance characteristic for a mail service. Other types of end-to-end requirements deal with security, availability, reliability, usability, ... One example of an approach that deals with end-to-end requirements is given in [17]. [17] uses first-order logic for expressing end-to-end requirements.

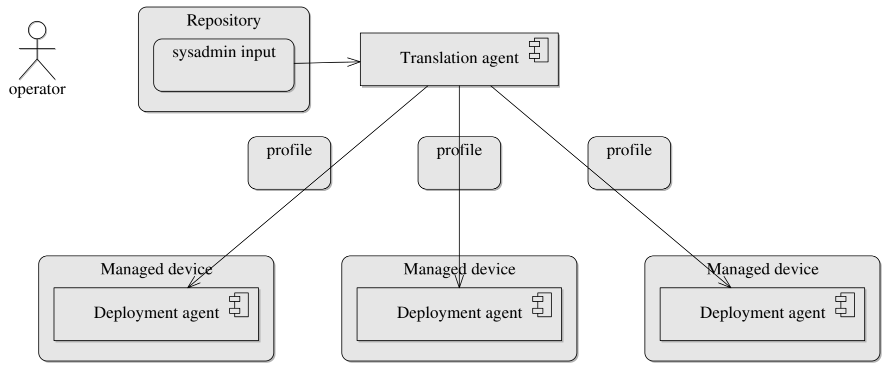


Figure 1: A conceptual architecture of system configuration tool.

2. **Instance distribution rules**: Instance distribution rules specify the distribution of instances in the network. We define an instance as a unit of configuration specification that can be decomposed in a set of parameters. Examples of instances are mail servers, DNS clients, firewalls and web servers. A web server, for example, has parameters for expressing its port, virtual hosts and supported scripting languages. In Figure 2, the instance distribution rule prescribes the number of mail servers that need to be activated in an infrastructure. The need for such a language is explicited in [3] and [2].

3. **Instance configurations**: At the level of instance configurations, each instance is an implementation independent representation of a configuration. An example of a tool at this level is Firmato [6]. Firmato allows modeling firewall configurations independent from the implementation software used.

4. **Implementation dependent instances** The level of implementation dependent instances specifies the required configuration in more detail. It describes the configuration specification in terms of the contents of software configuration files. In the example in Figure 2 a sendmail.cf file is used to describe the configuration of mail server instances.

5. **Configuration files**: At the level of configuration files, complete configuration files are mapped on a device or set of devices. In contrast with the previous level, this level has no knowledge of the contents of a configuration file.

6. **Bit-configurations**: At the level of Bit-configurations, disk images or diffs between disk images are mapped to a device or set of devices. This is the lowest level of configuration specification. Bit-level specifications have no knowledge of the contents of configuration files or the files itself. Examples of tools that operate on this level are imaging systems like Partimage [21], g4u [9] and Norton Ghost [24].

Figure 2 shows the six abstraction levels for system configuration, illustrated with an email setup. The illustration in Figure 2 is derived from an example discussed in [3]. The different abstraction levels are tied to the context of system configuration. In the context of policy languages, the classification of policy languages at different levels of abstraction is often done by distinguishing between high-level and low-level policies [16,25]. The distinction of what exactly is a high-level and low-level policy language is rather vague. In many cases, high-level policies are associated with the level that we call end-to-end requirements, while low-level policies are associated with the implementation dependent instances level. We believe that a classification tied to the context of system configuration gives a better insight in the different abstraction levels used by system configuration tools.

In conclusion, a system configuration tool automates the deployment of configuration specifications. At the level of bit-configurations, deployment is simply copying bit-sequences to disks, while deploying configurations specified as end-to-end requirements is a much more complex process.

2.1.3 Modularization mechanisms

One of the main reason system administrators want to automate the configuration of their devices is to avoid repetitive tasks. Repetitive tasks are not cost efficient. Moreover, they raise the chances of introducing errors. Repetitive tasks exist in a computer infrastructure because there are large parts of the configuration that are shared between a subset (or multiple overlapping subsets) of devices ([3]). For example, devices need the same DNS client configuration, authentication mechanism, shared file systems, ... A system configuration tool

1. End-to-end requirements
Configure enough mail servers to guarantee an SMTP response time of X seconds
⇓
2. Instance distribution rules
Configure N suitable machines as a mail server for this cluster
⇓
3. Instance configurations
Configure machines X, Y, Z as a mail server
⇓
4. Implementation dependent instances
Put these lines in sendmail.cf on machines X, Y, Z
⇓
5. Configuration files
Put configuration files on machines
⇓
6. Bit-configurations
Copy disk images onto machines

Figure 2: An example of different abstraction levels of configuration specification for an email setup.

that supports the modularization of configuration chunks reduces repetition in the configuration specification.

In its most basic form, modularization is achieved through a grouping mechanism: a device A is declared to be a member of group X and as a consequence inherits all system configuration chunks associated with X. More advanced mechanisms include query based groups, automatic definition of groups based on environmental data of the target device and hierarchical groups.

An additional property of a modularization mechanism is whether it enables third parties to contribute partial configuration specifications. Third parties can be hardware and software vendors or consultancy firms. System administrators can then model their infrastructure in function of the abstractions provided by the third-party modules and reuse the expertise or rely on support that a third party provides on their configuration modules.

2.1.4 Modeling of relations

One of the largest contributors to errors and downtime in infrastructures are wrong configurations [19, 20, 22] due to human error. An error in a configuration is commonly caused by an inconsistent configuration. For example, a DNS service that has been moved to an other server or moving an entire infrastructure to a new IP range. Explicitly modeling relations that exist in the network helps keeping a configuration model consistent.

Modeling relations is, like the modularization property of Section 2.1.3, a mechanism for minimizing redundancy in the configuration specification. When relations are made explicit, a tool can automatically change configurations that depend on each other. For example, when the location of a DNS server changes and the relation between the DNS server and clients is modeled in the configuration specification, a system configuration tool can automatically adapt the client configurations to use the new server. Again, modeling relations reduces the possibility of introducing errors in the configuration specification.

To evaluate how well a tool supports modeling of relations, we describe two orthogonal properties of relations: their granularity and their arity.

1. **granularity**: In Section 2.1.2, we defined an instance as a unit of configuration specification that can be decomposed in a set of parameters. Examples of instances are mail servers, DNS clients, firewalls and web servers. A web server, for example, has parameters for expressing its port, virtual hosts and supported scripting languages. Based on this definition, we can classify relations in three categories: (1) relations between instances, (2) relations between parameters and (3) relations between a parameter and an instance.

   (a) **Instance relations** represent a coarse grained dependency between instances. Instance dependencies can exist between instances on the same device, or between instances on different devices. An example of the former is the dependency between a DNS server instance and the startup system instance on a device: if a startup system instance is not present on a device (for example: /etc/init.d), the DNS server instance will not work. An example of dependencies between instances on different devices is the dependency between DNS servers and their clients.

(b) **Parameter relations** represent a dependency between parameters of instances. An example of this is a CNAME record in the DNS system: every CNAME record also needs an A record.

(c) **Parameter - instance relations** are used to express a relation between an individual parameter and an instance. For example a mail server depends on the existence of an MX record in the DNS server.

Note that it depends on the abstraction level of a tool which dependencies it can support. The two lowest abstraction layers in Figure 2, configuration files and bit-configurations, have no knowledge of parameters and as a consequence, they can only model instance dependencies.

2. **arity**: Relations can range from one-to-one to many-to-many relationships. A simple one-to-one relationship is a middleware platform depending on a language runtime. A many-to-many relationship is for example the relation between all DNS clients and DNS servers in a network. A system configuration tool can also provide support facilities to query and navigate relations in the system configuration specification. An example that motivates such facilities for navigating and querying relations involves an Internet service. For example, a webservice runs on a machine in the DMZ. This DMZ has a dedicated firewall that connects to the Internet through an edge router in the network. The webservice configuration has a relation to the host it is running on and a relation to the "Internet". The model also contains relations that represent all physical network connections. Using these relations, a firewall specification should be able to derive firewall rules for the webservice host, the DMZ router and the edge router [6].

An extra feature is the tool's ability to support the modeling of constraints on relations. We distinguish two types of constraints: validation constraints and generative constraints.

1. **validation constraints** are expressions that need to hold true for your configuration. Because of policy or technical factors, the set of allowable values for a relation can be limited. Constraints allow to express these limitations. Examples of such limitations are:
   - A server can only serve 100 clients.
   - Clients can only use the DNS server that is available in their own subnet.
   - Every server needs to be configured redundantly with a master and a slave server.
2. **generative constraints** are expressions that leave a degree of freedom between a chunk of configuration specification and the device on which this chunk needs to be applied. Languages without support for generative constraints need a 1-1 link between a chunk of configuration specification and the device on which is needs to be applied. Languages with support for generative constraints leave more degrees of freedom for the tool. An example of a generative constraint is: "One of the machines in this set of machines needs to be a mail server".

2.2 Deployment properties

2.2.1 Scalability

Large infrastructures are subject to constant change in their configuration. System configuration tools must deal with these changes and be able to quickly enforce the configuration specification, even for large infrastructures with thousands of nodes, ten thousands of relations and millions of parameters.

Large infrastructures typically get more benefit of using a higher level specification (see Figure 2). However, the higher-level the specification, the more processing power is needed to translate this high level specification to enforceable specifications on all managed devices. System configuration tools must find efficient algorithms to deal with this problem or restrict the expressiveness of the system configuration tool.

2.2.2 Workflow

Workflow management deals with planning and execution of (composite) changes in a configuration specification. Changes can affect services distributed over multiple machines and with dependencies on other services [3, 18].

One aspect of workflow management is state transfer. The behavior of a service is not only driven by its configuration specification, but also by the data it uses. In the case of a mail server, the data are the mail spool and mailboxes, while web pages serve as data for a web server. When upgrading a service or transferring a service to another device, one has to take care that the state (collection of data) remains consistent in the face of changes.

Another aspect of workflow management is the coordination of distributed changes. This has to be done very carefully as not to disrupt operations of the computing infrastructure. A change affecting multiple machines and services has to be executed as a single transaction. For example, when moving a DNS server from one device to another, one has to first activate the new server and make sure that all clients use the new server before deactivating the old server. For some services, characteristics of the managed protocol can be taken into account to make this process easier. For example, the SMTP protocol retries for a finite span of time to deliver a mail when the first attempt fails. A workflow management protocol can take advantage of this characteristic by allowing the mail server to be unreachable during the change.

A last aspect of workflow management is non-technical: if the organizational policy is to use maintenance windows for critical devices, the tool must understand that changes to these critical devices can influence the planning and execution of changes on other devices.

2.2.3 Deployment architecture

The typical setup of a system configuration tool is illustrated in Figure 1. A system configuration tool starts from a central specification for all managed devices. Next, it (optionally) processes this specification to device profiles and distributes these profiles (or the full specification) to every managed device. An agent running on the device then enforces the device's profile. For the rest of this section, we define the processing step from a central specification to device profiles as the translation agent. The agent running on every device is defined as the deployment agent.

System configuration tools differentiate their deployment architecture along two axises: 1. the architecture of the translation agent and 2. whether they use pull or push technology to distribute specifications .

1. **architecture of translation agent**: Possible approaches for the architecture of the translation agent can be classified in three categories, based on the number of translation agents compared to the number of managed devices: centralized management, weakly distributed management and strongly distributed management [15].

   (a) **centralized management** is the central server approach with only one translation agent. When dealing with huge networks, the central server quickly becomes a bottleneck. This is certainly the case when a system configuration tool uses a high-level abstraction, as the algorithm for computing a device's configuration will become complex.

   (b) **weakly distributed management** is an approach where multiple translation agents are present in the network. This approach can be realized for many centralized management tools by replicating the server and providing a shared policy repository for all servers. Another possible realization of this approach is organizing translation agents hierarchically.

   (c) **strongly distributed management** systems use a separate translation agent for each managed device. The difficulty with this approach is enforcing inter-device relations because each device is responsible for translating its own configuration specification. As a consequence, devices need to cooperate with each other to ensure consistency.

2. **push or pull**: In all approaches, each managed device contains a deployment agent that can be push or pull based. In the case of a pull based mechanism, the deployment agent needs to contact the translation agent to fetch the translated configurations. In a push based mechanism, the translation agent contacts the deployment agent. Deployment agents also have to be authenticated and their capabilities for fetching policies or configurations have to be limited. Configurations often contain sensitive information like passwords or keys and exposing this information to all deployment agents introduces a security risk.

2.2.4 Platform support

Modern infrastructures contain a variety of computing platforms: Windows/Unix/Mac OS X servers, but also desktop machines, laptops, handhelds, smartphones and network equipment. Even in relatively homogeneous environments, we can not assume that all devices run the same operating system: operating systems running on network equipment are fundamentally different than those running on servers/desktops and smartphones are yet another category of operating systems.

Good platform support or interaction with other tools is essential for reducing duplication in the configuration specification. Indeed, many relations exist between devices running different operating systems. For example: a server running Unix and a router/firewall running Cisco IOS. If different tools are used to manage the server and router, relations between the router and server need to be duplicated in both tools which in turn introduces consistency problems if one of the relations changes. An example of such a relation is the the firewall rule on a Cisco router that opens port 25 and the SMTP service on a Unix server.

2.3 Specification management properties

2.3.1 Usability

We identify three features concerning usability of a system configuration tool: 1. ease of use of the language, 2. support for testing specifications and, 3. monitoring the infrastructure.

1. **ease of use of the language**: The target audience of a system configuration tool are system administrators. The language of the system configuration tool should be powerful enough to replace their existing tools, which are mostly custom tools. But it should also be easy enough to use, so the average system administrator is able to use it. Good system administrators with a good education [13] are already scarce, so a system configuration tool should not require even higher education.

2. **support for testing specifications**: To understand the impact of a change in the specification, the system configuration tool can provide support for testing specifications through something as trivial as a dry-run mode or more complex mechanisms like the possibility to replicate parts of the production infrastructure in a (virtualized) testing infrastructure and testing the changes in that testing infrastructure first [5].

3. **monitoring the infrastructure**: A system configuration tool can provide an integrated (graphical) monitoring system and/or define a (language-based) interface for other tools to check the state of an infrastructure. A language-based interface has the advantage that multiple monitoring systems can be connected with the system configuration tool. A monitoring system enables the user to check the current state of the infrastructure and the delta with the configuration specification.

2.3.2 Versioning support

Some system configuration tools store their specification in text files. For those tools, a system configuration specification is essentially code. As a consequence, the same reasoning to use a version control system for source code applies. It enables developers and system administrators to document their changes and track them through history. In a configuration model this configuration history can also be used to rollback configuration changes and it makes sure an audit trail of changes exists.

The system configuration tool can opt to implement versioning of configuration specification using a custom mechanism or, when the specification is in text files, reuse an external version control system and make use of the hooks most generic version control systems provide.

2.3.3 Specification documentation

Usability studies [4, 12] show that a lot of time of a system administrator is spent on communication with other system administrators. These studies also show that a lot of time is lost because of miscommunication, where discussions and solutions are based on wrong assumptions. A system configuration tool that supports structured documentation can generate documentation from the system configuration specification itself and thus remove the need to keep the documentation in sync with the real specification.

2.3.4 Integration with environment

The infrastructure that is managed by the system configuration tool is not an island: it is connected to other networks, is in constant use and requires data from other sources than the system configuration specification to operate correctly. As a consequence, a system administrator may need information from external databases in its configuration specification (think LDAP for users/groups) or information about the run-time characteristics of the managed nodes. A system configuration tool that leverages on these existing sources of information integrates better with the environment in which it is operating because it does not require all existing information to be duplicated in the tool.

2.3.5 Conflict management

A configuration specification can contain conflicting definitions, so a system configuration tool should have a mechanism to deal with conflicts. Despite the presence of modularization mechanisms and relations modeling, a configuration specification can still contain errors, because it is written by a human. In case of such an error, a conflict is generated. We distinguish two types of conflicts: application specific conflicts and contradictions in the configuration specification, also called modality conflicts [14].

1. **application specific conflicts**: An example of an application specific conflict is the specification of two Internet services that use the same TCP port. In general, application specific conflicts can not be detected in the configuration specification. Examples of research on application specific protocols can be found in [10] and [7], where conflict management for IPSec and QoS policies is described.

2. **modality conflicts**: An example of a modality conflict is the prohibition and obligation to enable an

instance (for example a mail server) on a device. In general, modality conflicts can be detected in the configuration specifications.

When a configuration specification contains rules that cause a conflict, this conflict should be detected and acted upon.

2.3.6 Workflow enforcement

In most infrastructures a change to the configuration will never be deployed directly on the infrastructure. A policy describes which steps each update need to go through before it can be deployed on the production infrastructure. These steps can include testing on a development infrastructure, going through Q&A, review by a security specialist, testing on a exact copy of the infrastructure and so on. Exceptions on such policies can exist because not every update can go through all stages, updates can be so urgent that they need to be allowed immediately, but only with approval of two senior managers. A system configuration tool that provides support for modeling these existing workflows can adapt itself to the habits and processes of the system administrators and will thus be easier to use than system configuration tools without this support.

2.3.7 Access control

If an infrastructure is configured and managed based on a system configuration specification, control of this specification implies control of the full infrastructure. So it might be necessary to restrict access to the configuration specification. This is a challenge, especially in large infrastructures where a lot of system administrators with different responsibilities need to make changes to this specification. A lot of these large infrastructures are also federated infrastructures, so one specification can be managed from different administrative domains.

Authenticating and authorizing system administrators before they are making changes to the system configuration can prevent a junior system administrator who is only responsible for the logging infrastructure to make changes to other critical software running on the managed devices.

Many version control systems can enforce access control but the level on which the authorisation rules are expressed differs from the abstraction level of the specification itself. In most systems, this is based on the path of the file that contains the code or specification. But in most programming languages and system configuration tools, the relation between the name of the file and the contents of the file is very limited or even non-existing. For example an authorisation rule could express that users of the *logging* group should only set parameters of object from types in the logging namespace. With path-based access control this becomes: users of group logging should only access files in the */config/logging* directory. The latter assumes that every system administrator uses the correct files to store configuration specifications.

2.4 Support

2.4.1 Available documentation

To quickly gain users, tools have to make their barriers to entry as low as possible. A “ten minutes” tutorial is often invaluable to achieve this. When users get more comfortable with the tool, they need extensive reference documentation that describes all aspects of the tool in detail alongside documentation that uses a more process-oriented approach covering the most frequent use cases.

Thus, documentation is an important factor in the adoption process of a tool.

2.4.2 Commercial support

Studies [13] show that the need for commercial support varies amongst users. Unix users don’t call support lines as often as their Window-colleagues. The same holds true for training opportunities. In all cases, the fact that there is a company actively developing and supporting a tool helps to gain trust amongst system administrators and thus increases adoption.

2.4.3 Community

In our online society, community building is integral part of every product or service. Forums, wiki’s and social networks can provide an invaluable source of information that complements the official documentation of a tool and introduces system administrators to other users of their preferred tool.

2.4.4 Maturity

Some organizations prefer new features above stability, and others value stability higher than new features Therefore, it is important to know what the maturity of the tool is: Is it a new tool with some cutting edge features and frequent syntax changes in its language or a well-established tool with infrequent updates?

3 System configuration tools comparison

In this section we provide a summary of our evaluation of eleven tools. These tools consist of commercial and open-source tools. The set of commercial tools is based

Tool	Version
BCFG2	1.0.1
Cfengine 3	3.0.4
Opscode Chef	0.8.8
Puppet	0.25
LCFG	20100503
BMC Bladelogic Server Automation Suite	8
CA Network and Systems Management (NSM)	R11.x
IBM Tivoli System Automation for Multiplatforms	4.3.1
Microsoft Server Center Configuration Manager (SCCM)	2007 R2
HP Server Automation System	2010/08/12
Netomata Config Generator	0.9.1

Table 1: Version numbers of the set of evaluated tools.

on market research reports [8, 11] and consists of BMC Bladelogic Server Automation Suite, Computer Associates Network and Systems Management, IBM Tivoli System Automation for Multiplatforms, Microsoft System Center Configuration Manager and HP Server Automation System. For the open-source tools we selected a set of tools that were most prominently present in discussions at the previous LISA edition and referenced in publications. This set of tools consists of BCFG2, Cfengine3, Chef, Netomata, Puppet and LCFG.

Due to space constraints we limit the results of our evaluation to a summary of our findings for each property. The full evaluation of each tool is available on our website at `http://distrinet.cs.kuleuven.be/software/sysconfigtools`. We intend to keep the evaluations on this website in sync with major updates of each tool. For this paper we based our evaluation on the versions of each tool listed in Table 1.

3.1 Specification properties

3.1.1 Specification paradigm

Language type Cfengine, Puppet, Tivoli, Netomata and Bladelogic use a declarative DSL for their input specification. BCFG2 uses a declarative XML specification. Chef on the other hand uses an imperative ruby DSL. LCFG uses a DSL that instantiates components and set parameters on them. CA NSM, HP Server Automation and MS SCCM are like LCFG limited to setting parameters on their primitives.

User interface As with the language type, the tools can be grouped in open-source and commercial tools. The open-source tools focus on command-line interface while the commercial tools also provide a graphical interfaces. Tools such as Cfengine, Chef and Puppet provide a web-interface that allows to manage some aspects with a graphical interface. In the commercial tools all management is done through coommand-line and graphical interfaces.

3.1.2 Abstraction mechanisms

3.1.3 Modularization mechanisms

Type of grouping All tools provide a grouping mechanism for managed devices or resources. HP Server Automation, Tivoli and Netomata only provide static grouping. CA NSM and BCFG allow static grouping and hierarchies of groups. LCFG supports limited static, hierarchical and query based grouping through the C-preprocessor. Bladelogic supports static, hierarchical and query based groups. Cfengine and Puppet use the concept of classes to group configuration. Classes can include other classes to create hierarchies. Cfengine can assign classes statically or conditionally using expressions. Puppet can assign classes dynamically using external tools. Chef and MS SCCM can define static groups and groups based on queries.

Configuration modules BCFG, HP Server Automation, MS SCCM and Netomata have no support for configuration modules. Bladelogic can parametrise resources based on node characteristics to enable reuse. Tivoli includes sets of predefined policies that can be used to manage IBM products and SAP. LCFG can use third party components that offer a key-value interface to other policies, CA NSM provides a similar approach for third party agents that manage a device or subsystem. Cfengine uses bundles, Chef uses cookbooks and Puppet uses modules to distribute a reusable configuration specification for managing certain subsystems or devices.

3.1.4 Modeling of relations

BCFG, CA NSM, HP Server Automation and MS SCCM have no support for modeling relations in a configuration specification. Bladelogic can model one-to-one dependencies between scripts that need to be executed as a prerequisite, these are instance relations. Cfengine supports one-to-one, one-to-many and many-to-many relations between instances, parameters and between parameters and instances. On these relations generative constraints can be expressed. Chef can express many-to-many dependency relations between instances. Tivoli can also express relations of all arities between instances and parameters and just like Cfengine express generative constraints. LCFG can express one-to-one and many-to-many relations using spanning maps and references between instances and parameters. Netomata can model one-to-one network links and relations between devices. Finally Puppet can define one-to-many dependency relations between instances. The virtual resource functionality can also be used to define one-to-many relations between all instances.

3.2 Deployment properties

3.2.1 Scalability

The only method to evaluate how well a tool scales is to test each tool in a deployment and scale the number of managed nodes. In this evaluation we did not do this. To have an indication of the scalability we searched for cases of real-life deployments and divided the tools in three groups based on the number of managed devices and a group of tools for which no deployment information was available.

less than 1000 BCFG2

between 1000 and 10k LCFG and Puppet

more than 10k Bladelogic and Cfengine,

unknown CA NSM, Chef, HP Server Automation, Tivoli, MS SCCM and Netomata,

3.2.2 Workflow

BMC Bladelogic and HP Server Automation integrate with an orchestration tool to support coordination of distributed changes. Cfengine and Tivoli can coordinate distributed changes as well. MS SCCM and CA NSM support maintenance windows. Distributed changes in Puppet can be sequenced by exporting and collecting resources between managed devices. BCFG2, LCFG, Chef and Netomata have no support for workflow.

3.2.3 Deployment architecture

Translation agent Cfengine uses a strongly distributed architecture where the emphasis is on the agents that run on each managed device. The central server is only used for coordination and for policy distribution. Bladelogic, CA NSM and MS SCCM use one or more central servers. BCFG2, Chef, HP Server Automation, Tivoli, Netomata and Puppet use a central server. Chef and Puppet can also work in a standalone mode without central server to deploy a local specification.

Tool	Platform support
BCFG2	*BSD, AIX, Linux, Mac OS X and Solaris
Cfengine 3	*BSD, AIX, HP-UX, Linux, Mac OS X, Solaris and Windows
Opscode Chef	*BSD, Linux, Mac OS X, Solaris and Windows
Puppet	*BSD, AIX, Linux, Mac OS X, Solaris
LCFG	Linux (Scientific Linux)
BMC Bladelogic Server Automation Suite	AIX, HP-UX, Linux, Network equipment, Solaris and Windows
CA Network and Systems Management (NSM)	AIX, HP-UX, Linux, Mac OS X, Network equipment, Solaris and Windows
IBM Tivoli System Automation for Multiplatforms	AIX, Linux, Solaris and Windows
Microsoft Server Center Configuration Manager (SCCM)	Windows
HP Server Automation System	AIX, HP-UX, Linux, Network equipment, Solaris and Windows
Netomata Config Generator	Network equipment

Table 2: Version information for the set of evaluated tools.

Distribution mechanism The deployment agent of BCFG2, Cfengine, Chef, LCFG, MS SCCM and Puppet pull their specification from the central server. Bladelogic, CA NSM, HP Server Automation and Tivoli push the specification to the deployment agents. The central servers of Chef, MS SCCM and Puppet can notify the deployment agents that a new specification can be pulled. Netomata relies on external tools for distribution.

3.2.4 Platform support

The platforms that each tool supports is listed in Table 2.

3.3 Specification management properties

3.3.1 Usability

Usability Usability is a very hard property to quantify. We categorised the tools in easy, medium and hard. We determined this be assessing how easy a new user would be able to use and learn a tool. We tried to be as objective as possible to determine this but this part of the evaluation is subjective. We found Bladelogic, CA NSM, HP Server Automation, Tivoli and MSCCM easy to start using. The usability of Cfengine, LCFG and Puppet is medium, partially because of the custom syntax. Puppet also has a lot of confusing terminology but tools such as puppetdoc and puppetca make up for it so we did not classify it as hard to use. We found BCFG2 hard to use because of the XML input and the specification is distributed in a lot of different directories because of their plugin system. Chef is also hard to use because of its syntax and the use of a lot of custom terminology. Netomata is also hard to use because of its very concise syntax but powerful language.

Support for testing specifications BCFG2, Cfengine, LCFG and Puppet have a dry run mode. Netomata is inherently dry-run because it has no deployment part. Chef and Puppet support multiple environments such as testing, staging and production.

Monitoring the infrastructure BCFG2, Bladelogic, HP Server Automation, CA NSM, Tivoli, LCFG, Puppet and MS SCCM have various degrees of support for reporting about the deployment and collecting metrics from the managed devices. The commercial tools have more extensive support for this. Chef, LCFG, Puppet and Netomata can automatically generate the configuration for monitoring systems such as Nagios.

3.3.2 Versioning support

BCFG2, Bladelogic, Cfengine, Chef, Tivoli, LCFG, Netomata and Puppet use a textual input to create their configuration specification. This textual input can be managed in an external repository such as subversion or git. CA NSM and MS SCCM have internal support for policy versions. The central Chef server also maintains cookbook version information. For HP Server Automation it is unclear what is supported.

3.3.3 Specification documentation

BCFG2, Bladelogic, Chef, HP Server Automation, Tivoli, LCFG, Netomata and Puppet specifications can include free form comments. Cfengine can include structured comments that are used to generate documentation. Because Chef uses a Ruby DSL, Rdoc can also be used to generated documentation from structured comments. Puppet can generate reference documentation for built-in types from the comments included in the source code. No documentation support is available in CA NSM and MS SCCM.

3.3.4 Integration with environment

BCFG2, Cfengine, Chef, Tivoli, LCFG, MS SCCM and Puppet can discover runtime characteristics of managed devices which can be used when the profiles of each device are generated. Bladelogic can interact with external data sources like Active Directory.

3.3.5 Conflict management

BCFG and Puppet can detect modality conflict such as a file managed twice in a specification. Cfengine3 also detects modality conflicts such as an instable configuration that does not converge. Bladelogic and CA NSM have no conflict management support. Puppet also supports modality conflicts by allowing certain parameters of resources to be unique within a device, for example the filename of file resources.

3.3.6 Workflow enforcement

None of the evaluated tools have integrated support for enforcing workflows on specification updates. Bladelogic can tie in a change management system that defines workflows.

3.3.7 Access control

The tool that support external version repositories can reuse the path based access control of that repository. BMC, CA NSM, HP Server Automation, Tivoli, MS SCCM and the commercial version of Chef allow fine grained access control on “resources” in the specification.

3.4 Support

3.4.1 Available documentation

Bladelogic, CA NSM and HP Server Automation provide no public documentation. IBM Tivoli provides extensive documentation in their evaluation download. BCFG2, Cfengine, Chef, LCFG, MS SCCM and Puppet all provide extensive reference documentation, tutorials and examples on their websites. Netomata provides limited examples and documentation on their website and Wiki.

3.4.2 Commercial support

Not very surprising the commercial tools all provide commercial support. But most open-source tools also have a company behind them that develops the tool and provides commercial support. LCFG and BCFG2 have both been developed in academic institutes and have no commercial support.

3.4.3 Community

Cfengine, Chef, Tivoli, MS SCCM and Puppet have large and active communities. BCFG2 has a small but active community. CA NSM has a community but it is very scattered. BMC, Netomata and LCFG have small and not very active communities. For HP Server Automation we were unable to determine if a community exists.

3.4.4 Maturity

Some of the evaluated tools such as Tivoli and CA NSM are based on tools that exist for more than ten years, while other tools such as Chef and Netomata are as young as two years. However no relation between the feature set of a tool and their maturity seems to exist.

4 Putting the framework to use

4.1 How do I choose a tool for my environment?

Our framework and tool evaluations can help you to quickly trim down the list of tools to the tools that match your requirements. You list your required features, see which tools support these features and you have a limited list of tools to continue evaluating. In fact, our website at `http://distrinet.cs.kuleuven.be/software/sysconfigtools` provides a handy wizard to help you with this process.

The limitation of our framework is that it can not capture all factors that influence the process for choosing a system configuration tool: 1. We limit our evaluation to system configuration and do not include adjacent processes like provisioning, 2. Politics often play an important role when deciding on a tool, 3. your ideal solution might be too pricey, or 4. other, more subjective, factors come into play.

For all these reasons, we see our framework more as an aid that can quickly give you a high-level overview of the features of the most popular tools. Based on our framework, you can decide which tools deserve more time investment in your selection process.

4.2 How do I evaluate another tool using this framework?

We welcome clarifications to our existing evaluations and are happy to add other tool evaluations on the website. Internally, the website defines our framework as a taxonomy and every property is a term in this taxonomy. We associated a description with every term which should allow you to asses whether the property is supported by the tool you want to evaluate. Feel free to contact us for an account on the website so that you can add your evaluated tool.

5 Areas for improvement

Based on our evaluations in Section 3, we identify six areas for improvement in the current generation of tools. We believe that tools who address these areas will have a significant competitive advantage over other tools. The areas are:

1. **Create better abstractions**: Very few tools support creating higher-level abstractions like those mentioned in Figure 2 on page 4. If they do, those capabilities are hidden deep in the tool's documentation and not used often. We believe this is a missed opportunity. Creating higher-level abstractions would enable reuse of configuration specifications and lower the TCO of a computer infrastructure. To realize this, the language needs to (a) support primitives that promote reuse of configuration specifications like parametrization and modularization primitives, (b) support constraints modeling and enforcement, (c) deal with conflicts in the configuration specification and (d) model and enforce relations.

2. **Adapt to the target audience's processes**: A tool that adapts to the processes for system administration that exist in an organization is much more intuitive to work with than a tool that imposes its own processes on a system administrators. A few examples of how tools could support the existing processes better:

   - *structured documentation and knowledge management*: Cfengine3 is the only tool in our study that supports structured documentation in the input specification and has a knowledge management system that uses this structured documentation. Yet, almost all system administrators document their configurations. Some do it in comments in the configuration specification, some do it in separate files or in a fully-fledged content management system. In all cases, documentation needs to be kept in sync with the specification. If you add structured documentation to the configuration specification, the tool can generate the documentation automatically.
   - *integrate with version control systems*: A lot of system administrator teams use a version control system to manage their input specification. It allows them to quickly rollback a configuration and to see who made what changes.

Yet, very few tools provide real integration with those version control systems. A tool could quickly set up a virtualized test infrastructure for a branch that I created in my configuration. I would be able to test my configuration changes before I merge them with the main branch in the version control system that gets deployed on my real infrastructure.

- *semantic access controls*: In a team of system administrators, every admin has his own expertise: some are expert in managing networking equipment, other know everything from the desktop environment the company supports, others from the web application platform, ... As a consequence, responsibilities are assigned based on expertise and this expertise does not always aligns with machine boundaries. The ability to specify and enforce these domains of responsibility will prevent that for example a system administrator responsible for the web application platform modifies the mail infrastructure setup.
- *flexible workflow support*: Web content management systems like Drupal have support for customized workflows: If a junior editor submits an article, it needs to be reviewed by two senior editors, all articles need to be reviewed by one of the senior editors, ... The same type of workflows exist in computer infrastructures: junior system administrators need the approval from a senior to roll out a change, all changes in the DMZ needs to be approved by one of the managers and a senior system administrator, ... Enforcing such workflows would lower the number of accidental errors that are introduced in the configuration and aligns the tool's operation with the existing processes in the organization.

3. **Support true integrated management**: We would like to see a tool that provides a uniform interface to manage all types of devices that are present in a computer infrastructure: desktops, laptops, servers, smartphones and network equipment. Why would this be useful? When you have one tool, with one language that can specify the configuration of all devices, every system administrator speaks the same language and thinks in the same primitives: whether they are responsible for the network equipment, the data center or your desktops. The tool can then also support the specification and enforcement of relationships that cross platform boundaries: the dependencies between your web server farm and your Cisco load balancer, dependencies between desktops and servers, dependencies between your firewall and your DMZ servers, ... The current generation of tools either focuses on a single platform (Windows or Unix), focuses on one type of devices (servers) or needs different products with different interfaces for your devices (one product for network equipment, one for servers and one for desktops).

4. **Become more declarative**: The commercial tools in our study all start from scripting functionality: the system administrator can create or reuse a set of scripts and the tool provides a script-management layer. Research and experience with many open-source tools has shown that declarative specifications are far more robust than the traditional paradigm of imperative scripting. Imperative scripts have to deal with all possible states to become robust which results in a lot of if-else statements and spaghetti-code.

5. **Take the CIO's agenda into account**: Most open-source tools in our study have their origin in academia. As a result, they lag behind on the features that are on the CIO's checklists when deciding on a system configuration tool: (a) easy to use (graphical) user interface, reporting, (b) auditing, compliance, reporting capabilities in nice graphs and (c) access control support.

6. **Know that a system is software + configuration + data**: No tool has support for the data that is on the managed machines. Take a web server as example: the web server is software, that needs configuration files and serves data. System configuration tools can manage the software and configuration but have no support for state transfer: if my tool moves the web server to another node, I need to move the data manually.

6 Conclusion

We believe that this paper and our website can help system administrators make a more informed, and as a consequence better, choice for a system configuration tool. Our framework is not a mechanical tool: you can not check off the things you need and it will give you the perfect tool for you. We see it more as one of the decision factors that will save you a lot of time in the process of researching different tools: it quickly gives you a high-level overview of the features of each tool and enables you to trim down the list of possibilities for your use case. We will keep the website at `http://distrinet.cs.kuleuven.be/software/sysconfigtools` up to date when new versions of tools are released and are open for adding new tool evaluations to our website.

7 Acknowledgements

We would like to thank our shepherd, Eser Kandogan for his comments on the draft versions of the paper and Mark Burgess for sharing his insights in the constraints-part of our framework. We would also like thank our anonymous reviewers and all people who commented on the tool evaluations on the website.

This research is partially funded by the Agency for Innovation by Science and Technology in Flanders (IWT), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

References

[1] ALVA COUCH, JOHN HART, E. G. I., AND KALLAS, D. Seeking closure in an open world: A behavioral agent approach to configuration management. In *Proceedings of the 17th Large Installations Systems Administration (LISA) conference* (Baltimore, MD, USA, 10/2003 2003), Usenix Association, Usenix Association, p. 125–148.

[2] ANDERSON, P., AND COUCH, A. What is this thing called "system configuration"? LISA Invited Talk, November 2004.

[3] ANDERSON, P., AND SMITH, E. Configuration tools: Working together. In *Proceedings of the Large Installations Systems Administration (LISA) Conference* (Berkeley, CA, December 2005), Usenix Association, pp. 31–38.

[4] BARRETT, R., KANDOGAN, E., MAGLIO, P. P., HABER, E. M., TAKAYAMA, L. A., AND PRABAKER, M. Field studies of computer system administrators: analysis of system management tools and practices. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work* (New York, NY, USA, 2004), ACM, ACM, pp. 388–395.

[5] BARRETT, R., MAGLIO, P. P., KANDOGAN, E., AND BAILEY, J. Usable autonomic computing systems: The system administrators' perspective. *Advanced Engineering Informatics 19*, 3 (2005), 213 – 221. Autonomic Computing.

[6] BARTAL, Y., MAYER, A., NISSIM, K., AND WOOL, A. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst. 22*, 4 (2004), 381–420.

[7] CHARALAMBIDES, M., FLEGKAS, P., PAVLOU, G., BANDARA, A. K., LUPU, E. C., RUSSO, A., DULAY, N., SLOMAN, M., AND RUBIO-LOYOLA, J. Policy conflict analysis for quality of service management. In *POLICY '05: Proceedings of the Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 99–108.

[8] COLVILLE, R. J., AND SCOTT, D. Vendor Landscape: Server Provisioning and Configuration Management. Gartner Research, May 2008.

[9] FEYRER, H. g4u homepage. http://www.feyrer.de/g4u/.

[10] FU, Z. J., AND WU, S. F. Automatic generation of ipsec/vpn security policies in an intra-domain environment, 2001.

[11] GARBANI, J.-P., AND O'NEILL, P. The IT Management Software Megavendors. Forrester, August 2009.

[12] HABER, E. M., AND BAILEY, J. Design guidelines for system administration tools developed through ethnographic field studies. In *CHIMIT '07: Proceedings of the 2007 symposium on Computer human interaction for the management of information technology* (New York, NY, USA, 2007), ACM, ACM, p. 1.

[13] HREBEC, D. G., AND STIBER, M. A survey of system administrator mental models and situation awareness. In *SIGCPR '01: Proceedings of the 2001 ACM SIGCPR conference on Computer personnel research* (New York, NY, USA, 2001), ACM, ACM, pp. 166–172.

[14] LUPU, E., AND SLOMAN, M. Conflict analysis for management policies. In *Proceedings of the Vth International Symposium on Integrated Network Management IM'97* (May 1997), Chapman & Hall, pp. 1–14.

[15] MARTIN-FLATIN, J.-P., ZNATY, S., AND HUBAUX, J.-P. A survey of distributed enterprise network andsystems management paradigms. *J. Netw. Syst. Manage. 7*, 1 (1999), 9–26.

[16] MOFFETT, J. D. Requirements and policies. In *Proceedings of the Policy Workshop* (November 1999).

[17] NARAIN, S. Towards a foundation for building distributed systems via configuration. http://www.argreenhouse.com/papers/narain/Service-Grammar-Web-Version.pdf, 2004.

[18] OPPENHEIMER, D. The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop* (April 2003).

[19] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do internet services fail, and what can be done about it? In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems* (Berkeley, CA, USA, 2003), USENIX Association, USENIX Association, p. 1–1.

[20] OPPENHEIMER, D., AND PATTERSON, D. A. Studying and using failure data from large-scale internet services. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), ACM, ACM, p. 255–258.

[21] Partimage homepage. http://www.partimage.org.

[22] PATTERSON, D. A. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX conference on System administration* (Berkeley, CA, USA, 11/2002 2002), USENIX Association, USENIX Association, p. 185–188.

[23] RAYMER, D., STRASSNER, J., LEHTIHET, E., AND VAN DER MEER, S. End-to-end model driven policy based network management. In *Policies for Distributed Systems and Networks, 2006. Policy 2006. Seventh IEEE International Workshop* (2006), p. 4.

[24] SYMANTEC. Norton Ghost Homepage. http://www.symantec.com/ghost.

[25] VERMA, D. Simplifying network administration using policy-based management. *IEEE Network 16*, 2 (Mar/Apr 2002), 20–26.

High Performance Multi-Node File Copies and Checksums for Clustered File Systems*

Paul Z. Kolano, Robert B. Ciotti
NASA Advanced Supercomputing Division
NASA Ames Research Center, M/S 258-6
Moffett Field, CA 94035 U.S.A.
{paul.kolano,bob.ciotti}@nasa.gov

Abstract

Mcp and msum are drop-in replacements for the standard cp and md5sum programs that utilize multiple types of parallelism and other optimizations to achieve maximum copy and checksum performance on clustered file systems. Multi-threading is used to ensure that nodes are kept as busy as possible. Read/write parallelism allows individual operations of a single copy to be overlapped using asynchronous I/O. Multi-node cooperation allows different nodes to take part in the same copy/checksum. Split file processing allows multiple threads to operate concurrently on the same file. Finally, hash trees allow inherently serial checksums to be performed in parallel. This paper presents the design of mcp and msum and detailed performance numbers for each implemented optimization. It will be shown how mcp improves cp performance over 27x, msum improves md5sum performance almost 19x, and the combination of mcp and msum improves verified copies via cp and md5sum by almost 22x.

1 Introduction

Copies between local file systems are a daily activity. Files are constantly being moved to locations accessible by systems with different functions and/or storage limits, being backed up and restored, or being moved due to upgraded and/or replaced hardware. Hence, maximizing the performance of copies as well as checksums that ensure the integrity of copies is desirable to minimize the turnaround time of user and administrator activities. Modern parallel file systems provide very high performance for such operations using a variety of techniques such as striping files across multiple disks to increase aggregate I/O bandwidth and spreading disks across multiple servers to increase aggregate interconnect bandwidth.

To achieve peak performance from such systems, it is typically necessary to utilize multiple concurrent readers/writers from multiple systems to overcome various single-system limitations such as number of processors and network bandwidth. The standard cp and md5sum tools of GNU coreutils [11] found on every modern Unix/Linux system, however, utilize a single execution thread on a single CPU core of a single system, hence cannot take full advantage of the increased performance of clustered file system.

This paper describes mcp and msum, which are drop-in replacements for cp and md5sum that utilize multiple types of parallelism to achieve maximum copy and checksum performance on clustered file systems. Multi-threading is used to ensure that nodes are kept as busy as possible. Read/write parallelism allows individual operations of a single copy to be overlapped using asynchronous I/O. Multi-node cooperation allows different nodes to take part in the same copy/checksum. Split file processing allows multiple threads to operate concurrently on the same file. Finally, hash trees allow inherently serial checksums to be performed in parallel.

This paper is organized as follows. Section 2 presents related work. Section 3 describes the test environment used to obtain performance numbers. Section 4 discusses the various optimization strategies employed for file copies. Section 5 details the additional optimizations employed for file checksums. Section 6 describes how adding checksum capabilities to file copies decreases the cost of integrity-verified copies. Finally, Section 7 presents conclusions and related work.

2 Related Work

There are a variety of efforts related to the problem addressed by this paper. SGI ships a multi-threaded copy program called cxfscp [25] with their CXFS file system [27] that supports direct I/O and achieves significant performance gains over cp on shared-memory systems, but

*This work is supported by the NASA Advanced Supercomputing Division under Task Number ARC-013 (Contract NNA07CA29C) with Computer Sciences Corporation

offers minimal benefit on cluster architectures. Streaming parallel distributed cp (spdcp) [17] has similar goals as mcp and achieves very high performance on clustered file systems using MPI to parallelize transfers of files across many nodes. Like mcp, spdcp can utilize multiple nodes to transfer a single file. The spdcp designers made the conscious decision to develop from scratch, however, instead of using GNU coreutils as a base, whereas mcp started with coreutils to support all available cp options and to take advantage of known reliability characteristics. Mcp can also use a TCP model as well as MPI to support a larger class of systems.

Ong et al. [20] describe the parallelization of cp and other utilities using MPI. The cp command described, however, was designed to transfer the same file to many nodes as opposed to mcp, which was designed to allow many nodes to take part in the transfer of the same file. Desai et al. [9] use a similar strategy to create a parallel rsync utility that can synchronize files across many nodes at once. Peer-to-peer file sharing protocols such as BitTorrent [6] utilize multiple data streams for a single file to maximize network utilization from low bandwidth sources and support parallel hashing where the integrity of each piece may be verified independently.

High performance remote file transfer protocols such as bbFTP [3] and GridFTP [1] use multiple data streams for portions of the same file to overcome single stream TCP performance limitations. GridFTP additionally supports striped many-to-many transfers to aggregate network and I/O bandwidth. HPN-SSH [22] is a high performance version of SSH that achieves significant speedups using dynamically adjusted TCP receive windows. In addition, HPN-SSH incorporates a multi-threaded version of the AES counter mode cipher that increases performance further by parallelizing MAC and cipher operations on both the sender and receiver.

There are several related multi-threaded programs for the Windows operating systems. RichCopy [14] supports multi-threading in addition to the ability to turn off the system buffer, which is similar to mcp's direct I/O option. MTCopy [15] operates in a similar manner as mcp with a single file traversal thread and multiple worker threads. MTCopy also has the ability like mcp to split the processing of large files amongst multiple threads. HP-UX MD5 Secure Checksum [13] is an md5sum utility that uses multi-threading to compute the checksums of multiple files at once. Unlike msum, however, it cannot parallelize the checksum of a single file.

A variety of work uses custom hardware to increase checksum performance. Deepakumara et al. [8] describe a high speed FPGA implementation of MD5 using loop unrolling. Campobello et al. [4] describe a technique to generate high performance parallelized CRC checksums in compact circuits. CRCs are fast but are unsuitable for integrity checks of large files.

In general, checksums are not easily parallelizable since individual operations are not commutative. A general technique, used by mcp and msum, is based on Merkle trees [18], which allow different subtrees of hashes to be computed independently before being consolidated at the root. A similar approach is described by Sarkar and Schellenberg [23] to parallelize any hash function using a predetermined number of processors, which was used to create a parallel version of SHA-256 call PARSHA-256 [21]. Fixing the number of processors limits achievable concurrency, however, so mcp and msum instead use a predetermined leaf size in the hash tree, which allows an arbitrary number of processors to operate on the same file.

The underlying file system and hardware determine the maximum speed achievable by file copies and checksums. High performance file systems such as Lustre [26], CXFS [27], GPFS [24], and PVFS [5] utilize parallel striping across large numbers of disks to achieve higher aggregate performance than can be achieved from a single-disk file system.

3 Test Environment

All performance testing was carried out using dedicated jobs on the Pleiades supercluster at NASA Ames Research Center, which was recently ranked as the sixth fastest computer system in the world [29] with peak performance of 1.009 PFLOPs/s. Pleiades currently consists of 84,992 cores spread over 9472 nodes, which are connected by DDR and QDR Infiniband. There are three types of nodes with different processor and memory configurations. The nodes used for testing consist of a pair of 3.0 GHz quad-core Xeon Harpertown processors with 6 MB cache per pair of cores and 1 GB DDR2 memory per core for a total of 8 GB per node.

All file copies were performed between Lustre file systems, each with 1 Metadata Server (MDS) and 8 Object Storage Servers (OSS) serving 60 Object Storage Targets (OST). Based on the IOR benchmark [12], the source file system has peak read performance of 6.6 GB/s while the destination file system has peak write performance of 10.0 GB/s. Since copies can only progress at the minimum of the read and write speeds, the peak copy performance of this configuration is 6.6 GB/s. Checksums were performed on the same source file system, hence peak achievable checksum performance is also 6.6 GB/s. Both file systems had zero to minimal load during testing.

Two test cases are used throughout the paper. One case consists of 64 1 GB files while the other consists of a single 128 GB file. Both sets of files were generated from an actual 650 GB user data set. Before any tests could be done, it was necessary to choose a Lustre stripe count for the files that determines how many OSTs they are striped across. Table 1 shows the performance of cp for the two cases at the default (4 OSTs) and the maximum (60 OSTs) stripe counts. As can be seen, the 64 file case performs best at the default stripe count while the single file case performs best at the maximum. In the 64 file case, the maximum stripe count yields too much parallelism as every OST has to be consulted for every file. In the single file case, the default stripe count yields too little parallelism as large chunks of the file will reside on the same OST, which limits how much I/O bandwidth is available for the copy.

All operations in the remainder of the paper will use the default stripe count for the 64 file case and the maximum stripe count for the single file case. The corresponding cp performance of 174 MB/s for the 64 file case and 240 MB/s for the single file case represent the baseline that the various optimizations throughout the remainder should be compared against.

tool	stripe count	64x1 GB	1x128 GB
cp	default	174	102
cp	maximum	132	240

Table 1: Copy performance (MB/s) vs. stripe count

4 File Copy Optimization

4.1 Multi-Threaded Parallelism

In general, copying regular files is an embarrassingly parallel task since files are completely independent from one another. The processing of the hierarchy of directories containing the files, however, must be handled with care. In particular, a file's parent directory must exist and must be writable when the copy begins and must have its original permissions and ACLs when the copy completes.

The multi-threaded modifications to the cp command of GNU coreutils [11] utilize three thread types as shown in Figure 1 implemented via OpenMP [7]. A single *traversal thread* operates like the original cp program, but when a regular file is encountered, a copy task is pushed onto a shared task queue instead of performing the copy. Mutual exclusivity of all queues discussed is provided by semaphores based on OpenMP locks. Before setting properties of the file, such as permissions, the traversal thread waits until an open notification is received on a designated open queue, after which it will continue traversing the source tree.

One or more *worker threads* wait for tasks on the task queue. After it receives a task, each worker opens the source and target files, pushes a notification onto the open queue, then reads/writes the source/target until done. When stats are enabled, the worker pushes the task (with embedded stats) onto a designated stat queue and then waits for another task. The stat queue is processed by the *stat thread*, which prints the results of each copy task.

Table 2 shows the performance of multi-threading for varying numbers of threads. As can be seen, multi-threading alone has some benefit in the many file case up to 4 threads, after which the kernel buffer cache most likely becomes a bottleneck. For the single file case, multi-threading alone has no benefit since all but one thread sit idle while the file is being transferred. This case will be addressed in the next section.

tool	threads	64x1 GB	1x128 GB
mcp	1	177	248
mcp	2	271	248
mcp	4	326	248
mcp	8	277	248

Table 2: Multi-threaded copy performance (MB/s)

4.2 Single File Parallelization

As seen in the previous section, a number of files less than the number of threads results in imbalanced utilization and correspondingly lower performance. To evenly distribute workload across threads, mcp supports split processing of a single file so that multiple threads can operate on different portions of the same file. Figure 2 shows the processing by the traversal thread and worker threads when split processing is added. The main difference is that the traversal thread may add a number of tasks up to the size of the file divided by the split size and worker threads will seek to the correct location first and only process up to split size bytes.

Table 3 shows the performance of multi-threaded copies of a single large file when different split sizes are used. As can be seen, performance is increased from the unsplit case, but only minimal speedup is seen as the number of threads increases. In Section 4.5, however, significant benefits will be shown when splitting over multiple nodes. In addition, the table shows very little difference between the performance at different split sizes indicating that overhead from splitting is minimal. Since there is minimal difference, a split size of 1 GB will be used throughout the remainder of the results in the paper.

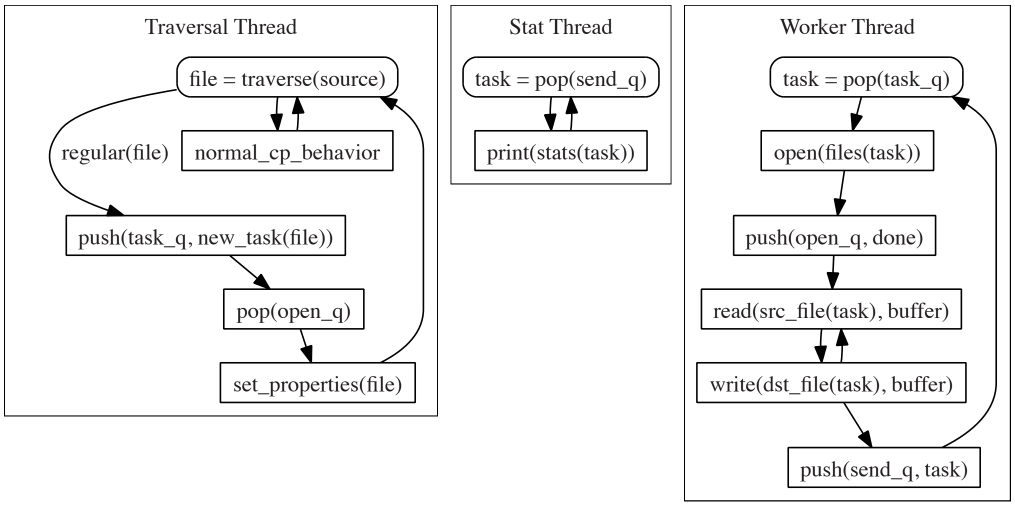


Figure 1: Multi-threaded copy processing

tool	threads	split size	1x128 GB
mcp	2	1 GB	286
mcp	2	16 GB	296
mcp	4	1 GB	324
mcp	4	16 GB	322
mcp	8	1 GB	336
mcp	8	16 GB	336

Table 3: Split file copy performance (MB/s)

4.3 Buffer Management

As witnessed in the Section 4.1, increasing the number of threads yields minimal gains at a certain point. One issue is that file copies generally exhibit poor buffer cache utilization since file data is read once, but then never accessed again. This increases CPU workload by the kernel and decreases performance of other I/O as it thrashes the buffer cache. To address this problem, mcp supports two buffer cache management approaches.

The first approach is to use file advisory information via the posix_fadvise() function, which allows programs to inform the kernel about how it will access data read/written from/to a file. Since mcp only uses data once, it advises the kernel to release the data as soon as it is read/written. The second approach is to skip the buffer cache entirely using direct I/O. In this case, all reads and writes go direct to disk without ever touching the buffer cache.

Table 4 shows the performance of multi-threaded copies when fadvise and direct I/O are utilized with different buffer sizes. As can be seen, performance increases significantly for both cases. Direct I/O achieves about double the performance of fadvise for a single node, but as will be seen in Section 4.5, the performance difference decreases as the number of nodes increases. From this point forward, 128 MB buffers will be used to maximize performance, although this size of buffer is impractical on multi-user systems due to memory limitations. More reasonable 4 MB buffers, however, have been found in testing to achieve a significant fraction of the performance of larger buffers.

4.4 Read/Write Parallelism

In the original cp implementation, a file is copied through a sequence of blocking read and write operations across each section of the file. Through the use of double buffering, it is possible to exploit additional parallelism between reads of one section and writes of another. Figure 3 shows how each worker thread operates in double buffering mode. The main difference is with the write of each file section. Instead of using a standard blocking write, an asynchronous write is triggered via aio_write(), which returns immediately. The read of the next section of the file cannot use the same buffer as it is still being used by the previous asynchronous write, so a second buffer is used. During the read, a write is also being performed, thereby theoretically reducing the original time to read each section from time(read) + time(write) to max(time(read), time(write)). After the read completes, the worker thread blocks until the write is finished (if not already done by that point) and the next cycle begins.

Table 5 shows the copy performance of double buffering for each buffer management scheme across a varying number of threads. As can be seen, double buffering increases the performance of the 64 file case across all

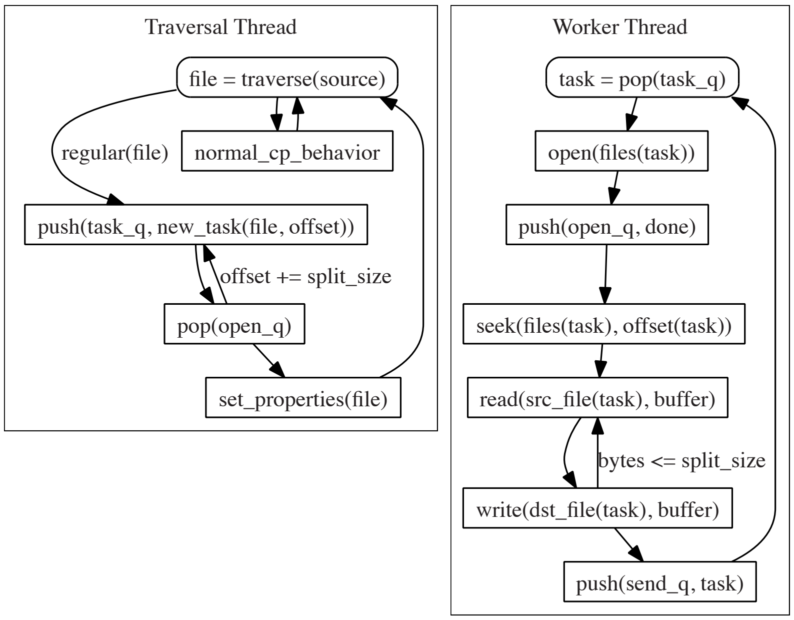


Figure 2: Split file copy processing

numbers of threads. The single file case, however, yields minimal benefit with the exception of the 1 thread case. It is clear a bottleneck exists in the single file case from a single node, but further investigation is needed to determine the exact cause. Double buffering is enabled in all remaining copy results.

4.5 Multi-Node Parallelism

While the results in Table 5 show significant speedup compared to the original cp implementation, it is still a fraction of the peak performance of the file system, hence it is unlikely that a single node can ever achieve the maximum. For this reason, mcp supports multi-node parallelism using both TCP and MPI models. Only the TCP model will be discussed as it is the more portable case and many of the processing details are similar.

In the multi-node TCP model, one node is designated as the *manager node* and parcels out copy tasks to *worker nodes*. The manager node is the only node that runs a traversal thread and stat thread. Both types of nodes have some number of worker threads as in the multi-threaded case. In addition, each node runs a *TCP thread* that is responsible for handling TCP-related activities, whose behavior is shown in Figure 4. The manager TCP thread waits for connections from worker TCP threads. A connection is initiated by a worker TCP thread whenever a worker thread on the same node is idle. If the worker previously completed a task, its stats are forwarded to the manager stat thread via the manager TCP thread. In all cases, the manager thread pops a task from the task queue and sends it back to the worker TCP thread, where it is pushed onto the local task queue for worker threads.

TCP communication introduces security concerns, especially for copies invoked by the root user. Integrity concerns include lost or blocked tasks, where files may not be updated that are supposed to be, replayed tasks where files may have changed between legitimate copies, and/or modified tasks with the source and destination changed arbitrarily. The main confidentiality concern is that contents of normally unreadable directories may be revealed if tasks are intercepted on the network or falsely requested from the manager. Finally, availability can be disrupted by falsely requesting tasks and/or by normal network denials of service.

To protect against TCP-based attacks, all communication is secured by Transport Layer Security (TLS) with Secure Remote Password (SRP) authentication [28]. TLS [10] provides integrity and privacy using encryption so tasks cannot be intercepted, replayed, or modified over the network. SRP [30] provides strong mutual authentication so worker nodes will only perform tasks from legitimate manager nodes and manager nodes will only reveal task details to legitimate worker nodes.

Table 6 shows the copy performance for different numbers of total threads spread across a varying number of nodes. As can be seen, multi-node parallelism achieves significant speedups over multi-threading alone, especially for the single file case. For the same number of total threads, performance increases as the number of

tool	threads	buffer size (MB)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	1	32	216	383	227	408
	1	64	219	401	226	411
	1	128	226	388	204	415
mcp	2	32	360	689	319	670
	2	64	372	723	317	696
	2	128	402	683	313	723
mcp	4	32	541	1065	330	679
	4	64	575	1039	327	699
	4	128	610	1055	331	721
	8	32	653	1185	332	685
	8	64	681	1223	328	718
mcp	8	128	692	1336	328	743

Table 4: Buffer cache managed copy performance (MB/s)

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	1	303	645	329	645
mcp	2	503	1111	329	709
mcp	4	653	1557	327	725
mcp	8	663	1763	325	731

Table 5: Double buffered copy performance (MB/s)

nodes increases as there is greater aggregate bandwidth and less resource contention. Direct I/O achieved the highest performance using 16 nodes and a single thread, while fadvise was best in the cases with the largest number of nodes and threads. While fadvise performed significantly worse than direct I/O in earlier sections, it actually surpasses direct I/O in some of the larger 64 file cases and achieved the fastest overall performance at 4.7 GB/s.

5 File Checksum Optimization

5.1 Multi-Threaded Parallelism

The greater the amount of data copied, the greater the possibility for data corruption [2]. The traditional approach to verifying integrity is to checksum the file at both the source and target and ensure that the values match. Checksums are inherently serial, however, so many of the techniques of the previous sections cannot be applied to any but the most trivial checksum algorithms.

Instead of parallelizing the algorithms themselves, serial algorithms are utilized in parallel through the use of Merkle (hash) trees [18] as mentioned previously. This functionality is implemented in a modification to the md5sum command of GNU coreutils called msum. Note that the use of hash trees makes multi-threaded msum unsuitable for verifying standard hashes. Hence, the main purpose of msum is to verify the integrity of copies within the same organization or across organizations that both use msum. This limitation is necessary for performance, however, as most standard hashes cannot be parallelized.

Msum uses a processing model that is similar to the mcp model shown in Figure 1. The msum traversal thread, however, is based on md5sum functionality with correspondingly less complexity. Figure 5 shows the processing by the msum stat thread (which has become the *stat/hash thread*) and worker threads. After copying their portion of the file, worker threads also create a hash tree of that portion, which is embedded in the task sent back to the stat/hash thread through the TCP threads. The stat/hash thread computes the root of the hash tree when all portions have been received.

Table 7 shows the performance of msum across varying numbers of threads and buffer management schemes. Note that msum utilizes libgcrypt [16] to enable support for many different hash types besides MD5, hence performance is not strictly comparable between the md5sum implementation and msum. As can be seen, significant performance gains are achieved by multi-threading even without buffer management. Direct I/O yields sizable gains while the gains by fadvise are more minimal.

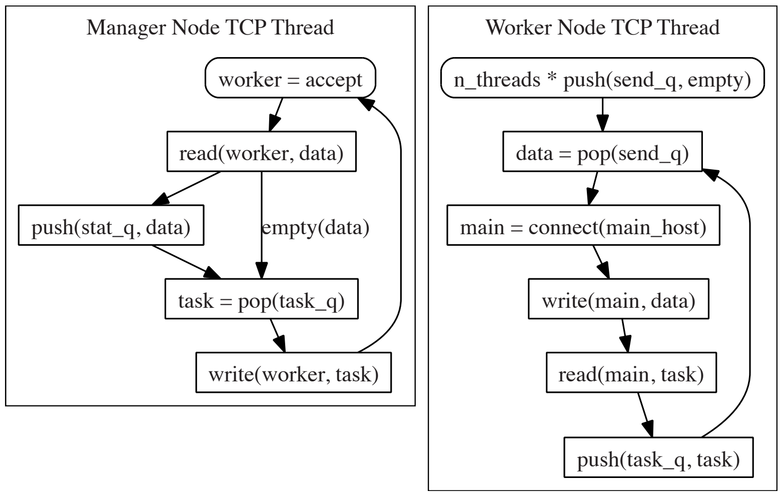


Figure 4: Multi-node copy processing

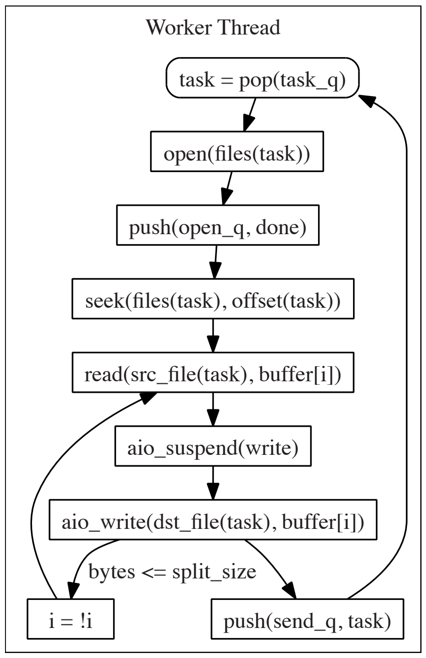


Figure 3: Double buffered copy processing

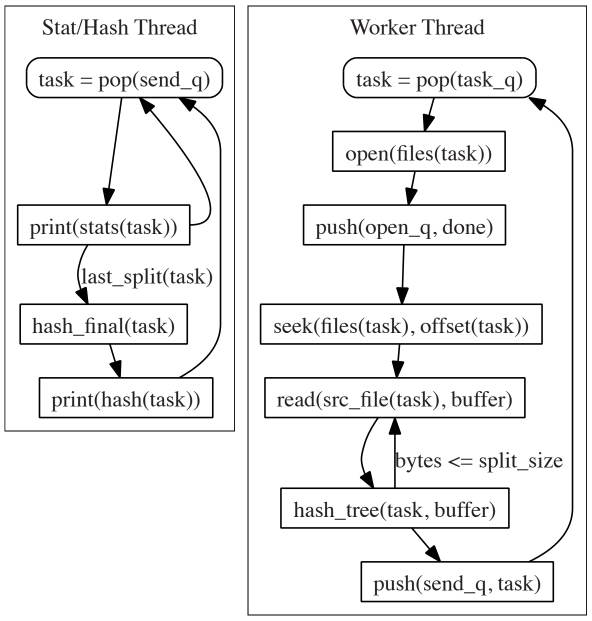


Figure 5: Multi-threaded checksum processing

5.2 Read/Hash Parallelism

Like the original cp implementation, the original md5sum implementation uses blocking I/O during reads of each section of the file. Double buffering can again be used to exploit additional parallelism between reads of one section and the hash computation of another. Figure 6 shows how each worker thread operates in double buffered mode within msum. In this mode, an initial read is used to seed one buffer. When that read completes, an asynchronous read is triggered via aio_read() into the second buffer. During this read, the hash of the first buffer is computed, after which the buffers are swapped and execution proceeds to the next section of the file after blocking until the previous read completes.

Double buffering theoretically reduces the original time to process each section of the file from time(read) + time(hash) to max(time(read), time(hash)) with best performance achieved when the time to read each sec-

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp	2	2	1	578	1161	273	1080
mcp	4	2	2	969	1673	379	1248
mcp	4	4	1	1119	2074	689	2001
mcp	8	2	4	1256	1857	426	1239
mcp	8	4	2	1818	2996	1068	2316
mcp	8	8	1	2058	3213	1289	3196
mcp	16	2	8	1276	2807	451	1226
mcp	16	4	4	2398	3446	1187	2208
mcp	16	8	2	3187	3599	1787	3723
mcp	16	16	1	3474	**4098**	2786	**4501**
mcp	32	4	8	2411	2957	1189	2142
mcp	32	8	4	3430	3459	2257	3706
mcp	32	16	2	4510	4011	3110	3930
mcp	64	8	8	3216	3346	2253	3626
mcp	64	16	4	**4735**	4011	3620	3914
mcp	128	16	8	–	–	**3824**	4400

Table 6: Multi-node copy performance (MB/s)

tool	threads	64x1 GB	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB	1x128 GB (fadvise)	1x128 GB (direct i/o)
md5sum	1	309	–	–	286	–	–
msum	1	278	284	330	263	265	349
msum	2	541	536	625	378	385	483
msum	4	906	903	1092	570	626	698
msum	8	886	908	1355	508	692	711

Table 7: Multi-threaded checksum performance (MB/s)

tion is the same as the time to hash each section. Table 8 shows the performance achieved by double buffering within msum for each buffer management scheme across a varying number of threads. As can be seen, double buffering increases the performance of all the 64 file cases except the 8 thread direct I/O case and all the single file cases except the 8 thread fadvise case. Double buffering is enabled in all remaining checksum results.

5.3 Multi-Node Parallelism

Msum supports the same TCP and MPI models as mcp for multi-node parallelism. TCP threads behave identically to those shown for mcp in Figure 4. Table 9 shows the checksum performance for different numbers of total threads spread across a varying number of nodes. As can be seen, multi-node parallelism achieves significant speedups over multi-threading alone. As was the case with mcp, performance generally increases for the same number of total threads as the number of nodes increases as there is greater aggregate bandwidth and less resource contention.

Both fadvise and direct I/O achieved the highest performance with 16 nodes and 2 threads in the 64 file case and with 16 nodes and 8 threads in the single file case. Once again, fadvise began to yield higher performance than direct I/O in some of the larger cases and once again had the highest overall performance at 5.8 GB/s. Note that this is 88% of peak of the file system and includes hashes as well as reads.

6 Verified File Copy Optimization

6.1 Buffer Reuse

In a typical integrity-verified copy, a file is checksummed at the source, copied, and then checksummed again at the destination to gain assurance that the bits at the source were copied accurately to the destination. This process normally requires two reads at the source since the checksum and copy programs are traditionally separate so each must access the data independently. Adding

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
msum	1	428	489	461	520
msum	2	811	973	462	522
msum	4	926	1647	662	766
msum	8	936	1315	613	776

Table 8: Double buffered checksum performance (MB/s)

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
msum	2	2	1	821	928	471	603
msum	4	2	2	1522	1834	832	939
msum	4	4	1	1487	1744	820	1027
msum	8	2	4	1819	2845	1298	1330
msum	8	4	2	2837	3122	1454	1798
msum	8	8	1	2844	3130	1808	2225
msum	16	2	8	1649	2979	1165	1076
msum	16	4	4	3218	3689	1891	1944
msum	16	8	2	4820	5292	3148	3654
msum	16	16	1	4770	4957	3248	3397
msum	32	4	8	3248	3719	1759	1936
msum	32	8	4	4664	4183	4640	4256
msum	32	16	2	**5812**	**5613**	4533	4856
msum	64	8	8	4114	3680	4256	3579
msum	64	16	4	5543	5131	4595	5114
msum	128	16	8	–	–	**5192**	**5227**

Table 9: Multi-node checksum performance (MB/s)

checksum functionality into the copy portion eliminates one of the reads to increase performance. Mcp incorporates checksums for this reason. This processing is similar to Figure 5 except the buffer is written between the read and the hash computation.

Table 10 shows the performance of copying with checksums for varying numbers of threads and different buffer management schemes. As was the case with the standard copy results in Table 4, direct I/O outperforms fadvise on a single node with the 64 file case achieving better results than the single file case.

6.2 Read/Hash Parallelism

The double buffering improvements of Section 5.2 were incorporated into mcp's checksum functionality with processing similar to Figure 6 with an additional write after the hash. Ideally, both the read of the next section and the write of the current section could be performed while the hash of the current section was being computed. This approach was implemented, but did not behave as expected, possibly due to concurrency controls within the file system. Further investigation is warranted as this would provide an additional increase in performance. Table 11 shows the performance increases achieved with double buffering during copies with checksums. As can be seen, performance increases in all but the 64 file fadvise case.

6.3 Multi-Node Parallelism

Table 12 shows the multi-node performance of copies incorporating checksum functionality. Peak performance of just under 4.0 GB/s was achieved with 8 nodes and 4 threads in the 64 file direct I/O case.

Table 13 is a composite view of Tables 5, 6, 8, 9, 11, and 12 that shows the performance of integrity-verified copies using the traditional checksum + copy + checksum versus a copy with embedded checksum + checksum. As can be seen, performance is better in almost every case with only a few scattered exceptions. Both fadvise and direct I/O achieve verified copies over 2 GB/s with 16 nodes and 2 threads in the 64 file case.

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	1	156	224	92	201
mcp (w/ sum)	2	294	428	152	376
mcp (w/ sum)	4	503	770	216	510
mcp (w/ sum)	8	629	1102	266	602

Table 10: Copy with checksum performance (MB/s)

tool	threads	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	1	190	290	104	222
mcp (w/ sum)	2	356	558	171	400
mcp (w/ sum)	4	561	966	235	560
mcp (w/ sum)	8	626	1498	275	671

Table 11: Double buffered copy with checksum performance (MB/s)

7 Conclusions and Future Work

Mcp and msum provide significant performance improvements over standard cp and md5sum using multiple types of parallelism and other optimizations. Tables 14, 15, and 16 show the maximum speedups obtained at each stage of optimization for copies, checksums, and integrity-verified copies, respectively. The relative effectiveness of each optimization is difficult to discern as they build upon each other and would have different peak speedups if applied in a different order. The total speedups from all improvements, however, is significant. Mcp improves cp performance over 27x, msum improves md5sum performance almost 19x, and the combination of mcp and msum improves verified copies via cp and md5sum by almost 22x. These improvements come in the form of drop-in replacements for cp and md5sum so are easily used and are available for download as open source software [19].

There are a variety of directions for future work. Currently, only optimized versions of cp and md5sum have been implemented from GNU coreutils. Optimized versions of the coreutils install and mv utilities should also be implemented as they would immediately benefit from the same techniques. In general, other common single-threaded utilities should be investigated to see if similar optimizations can be made.

Another area of study is to determine if mcp can be made into a remote transfer utility. While it currently can only be used for copies between local file systems, mcp already contains network authentication processing in the multi-node parallelization. In addition, most of the other techniques would be directly applicable to a high performance multi-node striping transfer utility. The missing component is a network bridge between the local read buffer and remote write buffer. The buffer reuse optimizations to checksums can be used directly to support integrity-verified remote transfers.

Although not discussed, mcp and msum both have the ability to store intermediate hash tree values within file system extended attributes. The purpose of this feature is to allow file corruption to be detected and precisely located over time in persistent files. The use of extended attributes has been found to be impractical, however, when the hash leaf size is small since only some file systems such as XFS support large extended attribute sizes and read/write performance of extended attributes is suboptimal. Further investigation is required to determine if greater generality and higher performance can be achieved using a mirrored hierarchy of regular files that contain the intermediate hash tree values.

References

[1] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster: The Globus Striped GridFTP Framework and Server. ACM/IEEE Supercomputing 2005 Conf., Nov. 2005.

[2] L.N. Bairavasundaram, G.R. Goodson, B. Schroeder, A.C. Arpaci-Dusseau, R.H. Arpaci-Dusseau: An Analysis of Data Corruption in the Storage Stack. 6th USENIX Conf. on File and Storage Technologies, Feb. 2008.

[3] BbFTP. `http://doc.in2p3.fr/bbftp`.

[4] G. Campobello, G. Patane, M. Russo: Parallel CRC Realization. IEEE Trans. on Computers, vol. 52, no. 10, Oct. 2003.

[5] P.H. Carns, W.B. Ligon, R.B. Ross, R. Thakur: PVFS: A Parallel File System for Linux Clusters.

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
mcp (w/ sum)	2	2	1	375	554	197	439
mcp (w/ sum)	4	2	2	682	1028	257	779
mcp (w/ sum)	4	4	1	714	1028	380	833
mcp (w/ sum)	8	2	4	1075	1756	398	1106
mcp (w/ sum)	8	4	2	1304	1815	611	1396
mcp (w/ sum)	8	8	1	1387	1858	722	1545
mcp (w/ sum)	16	2	8	1185	2506	617	1568
mcp (w/ sum)	16	4	4	2000	2716	825	1905
mcp (w/ sum)	16	8	2	2362	3032	1151	2233
mcp (w/ sum)	16	16	1	2439	2858	1319	2274
mcp (w/ sum)	32	4	8	2166	2809	907	2215
mcp (w/ sum)	32	8	4	3124	**3952**	1494	2318
mcp (w/ sum)	32	16	2	3229	3595	1973	3088
mcp (w/ sum)	64	8	8	2139	3147	1525	2693
mcp (w/ sum)	64	16	4	**3275**	3739	2353	**3277**
mcp (w/ sum)	128	16	8	–	–	**2481**	3183

Table 12: Multi-node copy with checksum performance (MB/s)

4th Annual Linux Showcase and Conf., Oct. 2000.

[6] B. Cohen: Incentives Build Robustness in BitTorrent. 1st Wkshp. on Economics of Peer-to-Peer Systems, Jun. 2003.

[7] L. Dagum, R. Menon: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Computational Science and Engineering, vol. 5, no. 1, Jan.-Mar. 1998.

[8] J. Deepakumara, H.M. Heys, R. Venkatesan: FPGA Implementation of MD5 Hash Algorithm. 14th IEEE Canadian Conf. on Electrical and Computer Engineering, May 2001.

[9] N. Desai, R. Bradshaw, A. Lusk, E. Lusk: MPI Cluster System Software. 11th European PVM/MPI Users' Group Meeting, Sept. 2004.

[10] T. Dierks, E. Rescorla: The Transport Layer Security (TLS) Protocol Version 1.2. IETF Request for Comments 5246, Aug. 2008.

[11] GNU Core Utilities. `http://www.gnu.org/software/coreutils/manual/coreutils.html`.

[12] R. Hedges, B. Loewe, T. McLarty, C. Morrone: Parallel File System Testing for the Lunatic Fringe: the care and feeding of restless I/O Power Users. 22nd IEEE / 13th NASA Goddard Conf. on Mass Storage Systems and Technologies, Apr. 2005.

[13] Hewlett Packard: HP-UX MD5 Secure Checksum A.01.01.02 Release Notes. Sept. 2007. `http://docs.hp.com/en/5992-2115/5992-2115.pdf`.

[14] J. Hoffman: Utility Spotlight: RichCopy. TechNet Magazine, Apr. 2009.

[15] Y.S. Li: MTCopy: A Multi-threaded Single/Multi File Copying Tool. CodeProject article, May 2008. `http://www.codeproject.com/KB/files/Lys_MTCopy.aspx`.

[16] Libgcrypt. `http://www.gnupg.org/documentation/manuals/gcrypt`.

[17] K. Matney, S. Canon, S. Oral: A First Look at Scalable I/O in Linux Commands. 9th LCI Intl. Conf. on High-Performance Clustered Computing, Apr. 2008.

[18] R.C. Merkle: Protocols for Public Key Cryptosystems. 1st IEEE Symp. on Security and Privacy, Apr. 1980.

[19] Multi-Threaded Multi-Node Utilities. `http://mutil.sourceforge.net`.

[20] E. Ong, E. Lusk, W. Gropp: Scalable Unix Commands for Parallel Processors: A High-Performance Implementation. 8th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, Sept. 2001.

[21] P. Pal, P. Sarkar: PARSHA-256 – A New Parallelizable Hash Function and a Multithreaded Implementation. 10th Intl. Wkshp. on Fast Software Encryption, Feb. 2003.

[22] C. Rapier, B. Bennett: High Speed Bulk Data Transfer Using the SSH Protocol. 15th ACM Mardi Gras Conf., Jan. 2008.

[23] P. Sarkar, P.J. Schellenberg: A Parallel Algorithm for Extending Cryptographic Hash Functions. 2nd

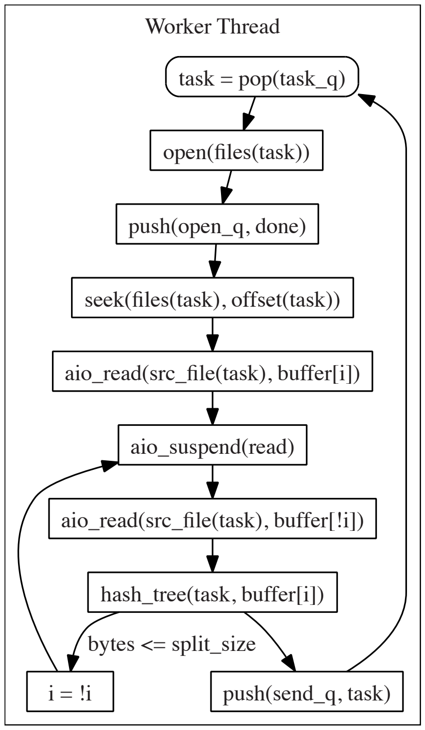

Worker Thread
task = pop(task_q)
open(files(task))
push(open_q, done)
seek(files(task), offset(task))
aio_read(src_file(task), buffer[i])
aio_suspend(read)
aio_read(src_file(task), buffer[!i])
hash_tree(task, buffer[i])
bytes <= split_size
i = !i
push(send_q, task)

Figure 6: Double buffered checksum processing

Intl. Conf. on Cryptology in India, Dec. 2001

[24] F. Schmuck, R. Haskin: GPFS: A Shared-Disk File System for Large Computing Clusters. 1st USENIX Conf. on File and Storage Technologies, Jan. 2002.

[25] Silicon Graphics Intl.: Cxfscp Man Page. `http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=0650&db=man&fname=/usr/share/catman/a_man/cat1m/cxfscp.z`.

[26] P. Schwan: Lustre: Building a File System for 1,000-node Clusters. 2003 Linux Symp., Jul. 2003.

[27] L. Shepard, E. Eppe: SGI InfiniteStorage Shared Filesystem CXFS: A High-Performance, Multi-OS Filesystem from SGI. Silicon Graphics, Inc. white paper, 2004.

[28] D. Taylor, T. Wu, N. Mavrogiannopoulos, T. Perrin: Using the Secure Remote Password (SRP) Protocol for TLS Authentication. IETF Request for Comments 5054, Nov. 2007.

[29] TOP500 Supercomputing Sites, Jun. 2010. `http://www.top500.org/lists/2010/06`.

[30] T. Wu: The Secure Remote Password Protocol. 5th ISOC Network and Distributed System Security Symp., Mar. 1998.

tool	threads (total)	nodes	threads (per node)	64x1 GB (fadvise)	64x1 GB (direct i/o)	1x128 GB (fadvise)	1x128 GB (direct i/o)
md5sum + cp + md5sum	1	1	1	100		90	
msum + mcp + msum	1	1	1	125	177	135	185
mcp (w/ sum) + msum	1	1	1	131	182	84	155
msum + mcp + msum	2	1	2	224	338	135	190
mcp (w/ sum) + msum	2	1	2	247	354	124	226
msum + mcp + msum	2	2	1	240	331	126	235
mcp (w/ sum) + msum	2	2	1	257	346	138	254
msum + mcp + msum	4	1	4	270	538	164	250
mcp (w/ sum) + msum	4	1	4	349	608	173	323
msum + mcp + msum	4	2	2	426	592	198	341
mcp (w/ sum) + msum	4	2	2	470	658	196	425
msum + mcp + msum	4	4	1	446	613	257	408
mcp (w/ sum) + msum	4	4	1	482	646	259	459
msum + mcp + msum	8	1	8	274	478	157	253
mcp (w/ sum) + msum	8	1	8	375	700	189	359
msum + mcp + msum	8	2	4	527	805	257	432
mcp (w/ sum) + msum	8	2	4	675	1085	304	603
msum + mcp + msum	8	4	2	796	1026	432	647
mcp (w/ sum) + msum	8	4	2	893	1147	430	785
msum + mcp + msum	8	8	1	840	1052	531	825
mcp (w/ sum) + msum	8	8	1	932	1165	515	911
msum + mcp + msum	16	2	8	500	973	254	373
mcp (w/ sum) + msum	16	2	8	689	1361	403	638
msum + mcp + msum	16	4	4	962	1201	526	674
mcp (w/ sum) + msum	16	4	4	1233	1564	574	962
msum + mcp + msum	16	8	2	1372	1524	836	1225
mcp (w/ sum) + msum	16	8	2	1585	1927	842	1386
msum + mcp + msum	16	16	1	1414	1544	1025	1233
mcp (w/ sum) + msum	16	16	1	1613	1812	938	1362
msum + mcp + msum	32	4	8	970	1141	505	666
mcp (w/ sum) + msum	32	4	8	1299	1600	598	1033
msum + mcp + msum	32	8	4	1388	1303	1144	1351
mcp (w/ sum) + msum	32	8	4	1870	2032	1130	1500
msum + mcp + msum	32	16	2	1767	1651	1311	1500
mcp (w/ sum) + msum	32	16	2	**2075**	**2191**	1374	1887
msum + mcp + msum	64	8	8	1254	1187	1094	1198
mcp (w/ sum) + msum	64	8	8	1407	1696	1122	1536
msum + mcp + msum	64	16	4	1748	1564	1405	1546
mcp (w/ sum) + msum	64	16	4	2058	2162	1556	**1997**
msum + mcp + msum	128	16	8	–	–	1546	1639
mcp (w/ sum) + msum	128	16	8	–	–	**1678**	1978

Table 13: Multi-node verified copy performance (MB/s)

origin	optimization	peak speedup
cp	multi-threading	1.9
multi-threading	split files	1.4
split files	posix_fadvise	2.5
spit files	direct I/O	4.8
posix_fadvise	double buffering	1.3
direct I/O	double buffering	1.6
double buffering	multiple nodes	7.1
cp	all	27.2

Table 14: Summary of copy optimizations

origin	optimization	peak speedup
md5sum	multi-threading	2.9
multi-threading	split files	2.2
split files	posix_fadvise	1.4
split files	direct I/O	1.5
posix_fadvise	double buffering	1.7
direct I/O	double buffering	1.6
double buffering	multiple nodes	6.2
md5sum	all	18.8

Table 15: Summary of checksum optimizations

origin	optimization	peak speedup
md5sum + cp + md5sum	multi-threaded + split files + buffer management + buffer reuse	6.1
multi-threaded + split files + buffer management + buffer reuse	double buffering	1.2
double buffering	multiple nodes	10.7
md5sum + cp + md5sum	all	21.9

Table 16: Summary of verified copy optimizations

Fast and Secure Laptop Backups with Encrypted De-duplication

Paul Anderson
University of Edinburgh
dcspaul@ed.ac.uk

Le Zhang
University of Edinburgh
zhang.le@ed.ac.uk

Keywords: backup, de-duplication, encryption, cloud computing.

Abstract

Many people now store large quantities of personal and corporate data on laptops or home computers. These often have poor or intermittent connectivity, and are vulnerable to theft or hardware failure. Conventional backup solutions are not well suited to this environment, and backup regimes are frequently inadequate. This paper describes an algorithm which takes advantage of the data which is common between users to increase the speed of backups, and reduce the storage requirements. This algorithm supports client-end per-user encryption which is necessary for confidential personal data. It also supports a unique feature which allows immediate detection of common subtrees, avoiding the need to query the backup system for every file. We describe a prototype implementation of this algorithm for Apple OS X, and present an analysis of the potential effectiveness, using real data obtained from a set of typical users. Finally, we discuss the use of this prototype in conjunction with remote cloud storage, and present an analysis of the typical cost savings.

1 Introduction

Data backup has been an important issue ever since computers have been used to store valuable information. There has been a considerable amount of research on this topic, and a plethora of solutions are available which largely satisfy traditional requirements. However, new modes of working, such as the extensive use of personal laptops, present new challenges. Existing techniques do not meet these challenges well, and many individuals and organisations have partial, ad-hoc backup schemes which present real risks. For example:

- Backups are often made to a local disk and copies are not stored offsite.
- Backups are not encrypted and vulnerable to theft.
- Personal (rather than corporate) information is accidentally stored in plaintext on a corporate service where it can be read by other employees.
- Backups often just include “user files” in the assumption that “system files” can be easily recovered from elsewhere[1].
- The inconvenience of making backups leads to infrequent and irregular scheduling.

Even recent attempts to make backups largely transparent, such as Apple’s Time Machine [10] suffer from the first two of the above problems, and may even lead users into a false sense of data security.

There has recently been a proliferation of ”Cloud” backup solutions [3, 7, 12, 4, 1, 8, 6, 2]. In theory, these are capable of addressing some of the above problems. But, in practice, complete backups are unreasonably slow[2]:

> *”I have a home Internet backup service and about 1TB of data at home. It took me about three months to get all of the data copied off site via my cable connection, which was the bottleneck. If I had a crash before the off-site copy was created, I would have lost data”* [3]

And many organisations may prefer to hold copies of the backup data themselves.

[1] In practice, we found a small but significant number of unique files outside of the ”user space” (figure 3) which means that this may not be such a reasonable assumption.

[2] Home broadband connections usually have upload speeds which are very significantly less than the download speed

[3] Henry Newman, October 9th 2009 - `http://www.enterprisestorageforum.com/technology/features/article.php/3843151`

1.1 De-duplication & Encryption

We observed that there is a good deal of sharing between the data on typical laptops (figure 3). For example, most (*but not all*) of the "system files" are likely to be shared with at least one other user. And it is common for users in the same environment to share copies of the same papers, or software packages, or even music files. Exploiting this duplication would clearly enable us to save space on the backup system. But equally importantly, it would significantly reduce the time required for backups in most cases – upgrading an operating system, or downloading a new music file should not require any additional backup time at all if someone else has already backed-up those same files.

There has been a lot of interest recently in *de-duplication* techniques, using *content-addressable storage* (CAS). This is designed to address exactly the above problem. However, most of these solutions are intended for use in a local filesystem [18, 9, 11] or SAN [20]. This has two major drawbacks: (i) clients must send the data to the remote filesystem before the duplication is detected – this forfeits the potential saving in network traffic and time. And (ii) any encryption occurs on the server, hence exposing sensitive information to the owner of the service – this is usually not appropriate for many of the files on a typical laptop which are essentially "personal", rather than "corporate"[4].

1.2 A Solution

This paper presents an algorithm and prototype software which overcome these two limitations. The algorithm allows data to be encrypted independently without invalidating the de-duplication. In addition, it is capable of identifying shared sub-trees of a directory hierarchy, so that a single access to the backup store can detect when an entire subtree is already present and need not be re-copied. Clearly, this algorithm works for any type of system, but it is particularly appropriate for laptops where the connection speed and network availability are bigger issues than the processing time.

Initial versions of the prototype were intended to make direct use of cloud services such as Amazon S3 for remote storage. However, this has proven to be unworkable, and we discuss the reasons for this, presenting a practical extension to enable the use of such services.

Section 2 describes the algorithm. Section 3 describes the prototype implementation and gives some preliminary performance results. Section 4 presents the data collected from a typical user community to determine the practical extent and nature of shared files. This data is used to predict the performance in a typical environment, and to suggest further optimisations. Section 5 discusses the problems with direct backup to the cloud and presents a practical solution, including an analysis of the cost savings. Section 6 presents some conclusions.

[4] Of course, performing local encryption with personal keys would produce different cipher-text copies of the same file and invalidate any benefits of the de-duplication.

2 The Backup Algorithm

The backup algorithm builds on existing de-duplication and convergent encryption technology:

2.1 De-duplication

A *hashing function* (e.g. [14]) can be used to return a unique key for a block of data, based only on the contents of the data; if two people have the same data, the hashing function will return the same key[5]. If this key is used as the index for storing the data block, then any attempt to store multiple copies of the same block will be detected immediately. In some circumstances, it may be necessary store additional metadata, or a reference count to keep track of the multiple "owners", but it is not necessary to store multiple copies of the data itself.

2.2 Convergent Encryption

Encrypting data invalidates the de-duplication; two identical data blocks, encrypted with different keys, will yield different encrypted data blocks which can no longer be shared. A technique known as *convergent encryption* [16, 21, 19, 23] is designed to overcome this – the encryption key for the data block is derived from the contents of the data using a function which is similar to (but *independent of*) the hash function. Two identical data blocks will thus yield identical encrypted blocks which can be de-duplicated in the normal way. Of course each block now has a separate encryption key, and some mechanism is needed for each owner to record and retrieve the keys associated with "their" data blocks.

Typical implementations (such as [25]) involve complex schemes for storing and managing these keys as part of the block meta-data. This can be a reasonable approach when the de-duplication is part of a local filesystem. But there is considerable overhead in interrogating and maintaining this meta-data, which can be significant when the de-duplication and encryption is being performed remotely – and this is necessary in our case, to preserve the privacy of the data.

[5] Technically, it is possible for two different data blocks to return the same key. However, with a good hash function, the chances of this are sufficiently small to be insignificant - *"if you have something less than 95 EB of data, then your odds don't appear in 50 decimal places"*[5] - i.e. many orders of magnitude less than the chances of failure in any other part of the system.

2.3 The Algorithm

We have developed an algorithm which takes advantage of the hierarchical structure of the filesystem:

- Files are copied into the backup store as *file objects* using the convergent encryption process described above.
- Directories are stored as *directory objects* – these are simply files which contain the normal directory meta-data for the children, *and* the encryption/hash keys for each child.

To recover a complete directory hierarchy, we need only know the keys for the root node – locating this directory object and decrypting it yields the encryption and hash keys for all of the children and we can recursively recover the entire tree. This has some significant advantages:

- Each user only needs to record the keys for the root node. Typically, these would be stored independently on the backup system (one set for each stored hierarchy), and encrypted with the user's personal key.
- The hash value of a directory object acts as a unique identifier for the whole subtree; if the object representing a directory is present in the backup store, then we know that the entire subtree below it is also present. This means that we do not need to query the store for any of the descendants. It not uncommon to see fairly large shared subtrees[6], so this is a significant saving for remote access where the cost of queries is likely to be high. Section 4.3 presents some concrete experimental results.
- No querying or updating of additional metadata is required[7]. This means that updates to the backup store are atomic.

This algorithm does have some disadvantages. In particular, a change to any node implies a change to all of the ancestor nodes up to the root. It is extremely difficult to estimate the impact of this in a production environment, but preliminary testing seems to indicate that this is not a significant problem. There is also some disclosure of information; if a user has a copy of a file, it it possible to tell whether or not some other user also has a copy of the same file. This is an inevitable consequence of any system which supports storage sharing – if a user stores a file, and the size of the stored data does not increase, then there must have been a copy of this file already present.

[6]For example, between successive backups of the same system, or as the same application downloaded to different systems.

[7]If it is necessary to support deletion of shared blocks, then some kind of reference counting or garbage-collection mechanism is necessary, and this may be require additional metadata.

2.4 Implementation

An efficient implementation of this algorithm requires some care. The hash key for a directory object depends on its contents. This in turn depends on the keys for all of the children. Hence the computation of keys must proceed bottom-up. However, we want to prevent the backup from descending into any subtree whose root is already present. And this requires the backup itself to proceed top-down. For example:

```
BackupNode(N) {
  If N is a directory, then let O = DirectoryObjectFor(N)
  Otherwise, let O = contents of N
  Let H = Hash(O)
  if there is no item with index H in the backup store, then {
    Store O in the backup store with index H
    If N is a directory {
      For each entry E in the directory, BackupNode(E)
    }
  }
}

DirectoryObjectFor(D) {
  Create an empty directory object N
  For each entry E in the directory D {
    If E is a directory, then let O = DirectoryObjectFor(E)
    Otherwise, let O = contents of E
    Let H = Hash(O)
    Add the metadata for E, and the hash H to N
  }
  Return N
}
```

Of course, this is still a rather naive implementation – the hash for a particular object will be recomputed once for every directory in its path. This would be unacceptably expensive in practice and the hash function would probably be memoized. A production implementation presents many other opportunities for optimisation; caching of directory objects, parallelisation of compute-intensive tasks (encryption,hashing), and careful detection of files which have been (un-)modified since a previous run.

3 Prototype System

We developed a prototype backup system for Apple OS X as a proof of concept of the proposed algorithm[8]. The purpose is to be able to backup all files on a user's laptop to a central remote storage. The prototype was implemented as a set of command line utilities each performs a single task in the backup process such as scan-

[8]An earlier, simpler proof-of-concept was implemented under Windows as a student project [22].

ning file system changes, uploading files, restoring file system from backup etc. When wrapped in a GUI front-end, our application can be used as a drop-in replacement for the built-in Time Machine backup application [10].

3.1 Architecture

The underlying storage model we adopt is a write once, read many model. This model assumes that once a file is stored on the storage, it will never be deleted or rewritten. This is the common practice employed in enterprise environment where key business data such as electronic business records and financial data are required by law to be kept for at least five and seven years respectively [13]. This kind of storage model, commonly found in DVD-R or tape backup storage, is usually optimised for throughput rather than random access speed. The alternative storage model is write many, read many model that permits deleting older backups to reclaim some space. To achieve that some kind of reference counting mechanism is needed to safely delete un-referred files on the storage. To keep the prototype simple we opt to use the write once, read many storage model.

Our system is architected as a client/server application, where a backup client running on a user's laptop uploads encrypted data blocks to a central server which includes dedicated server side processing. This is in contrast to the thin-cloud approach where only minimal cloud-like storage interface is required on the server end as proposed in the Cumulus system [24]. Cumulus demonstrated that with careful client side data aggregation, backing up to the thin-cloud can be as efficient as integrated client/server approaches. This thin-cloud approach is appealing in a single user environment, however it raises three problems in a multi-user setting with data de-duplication technology:

1. The cloud storage model used by the thin-cloud backup solution does not have a straightforward way of sharing data between different user accounts without posing serious security threat.

2. There is no way to validate the content address of an uploaded object on the server. A malicious user can start an attack by uploading random data with a different content address to corrupt the whole backup.

3. The client side data aggregation used in the thin-cloud approach for performance reason will make data de-duplication very difficult, if not impossible.

In our prototype system we argue for a thin-server approach that addresses all these issues. The majority of computation will happen on the client side (hashing, encryption, data aggregation). A dedicated backup server will handle per-user security required in a multi-user environment. Instead of going for a full-blown client/server backup architecture with custom data transfer protocol, user authentication mechanism and hefty server end software we want to re-use the existing services provided by operating system itself as much as possible. To this end, we used standard services that come with many POSIX systems (Access control list (ACL) mechanism, user account management, Common Internet File System/Server Message Block (CIFS/SMB) server) and deployed a small server application written in Python on the server side to handle data validation. In addition, the whole server, once properly configured with ACL permissions, server application scripts, storage devices, can be packed as a virtual machine image and deployed into an enterprise's existing visualised storage architecture. This means the number of supported clients can be easily scaled to meet the backup demands.

Figure 1 depicts the architecture employed in our prototype. The system consists of several modules described below.

3.1.1 FSEvents Module

FSEvents was introduced as a system service in the OS X Leopard release (version 10.5). It logs file changes at the file system level and can report file system changes since a given time stamp in the past. This service is part of the foundational technologies used by Apple's built-in backup solution Time Machine. Our prototype system utilises the FSEvents service to get a list of changed files for incremental backup.

For efficiency reason, the event reported by the FSEvents API is at directory level only, i.e. it only tells which directory has changes in it but not exactly what was changed. To identify which files are changed we use a local meta database to maintain and compare current files' meta information with their historical values.

3.1.2 Local Meta DB

The local meta DB is used to implement incremental backup. The content of the DB includes: pathname, file size, last modification time, block-level checksums, and a flag indicates if the file has been backed up or not. For each backup session, the local meta DB will produce a list of files to backup which is sorted by directory-depth from bottom up. This is to ensure that a directory will be backed up only after all its children have been backed-up.

3.1.3 Backup Module

The backup module encapsulates the backup logic and interfaces with other modules. For each backup session, it first retrieves a list of files to backup from the local

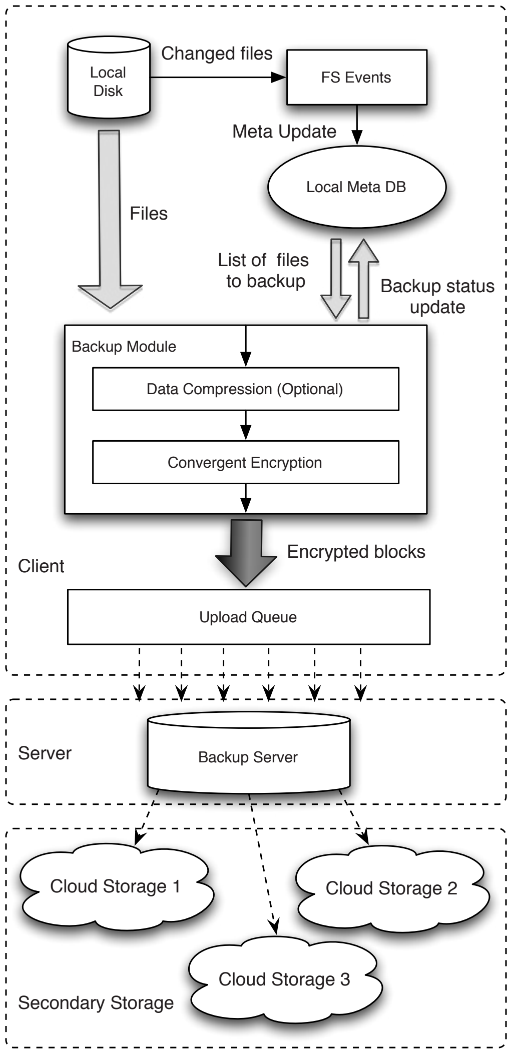


Figure 1: Architecture of the proposed backup system. Optionally data on the backup server can be replicated to multiple cloud storages in background.

meta DB, then it fetches those files from the local disk, calculates file hashs at block level, and encrypts each data block with an encryption key generated from the block's hash value. The final sequence of encrypted data blocks, together with their unique content addresses, are put in an upload queue to be dispatched to the remote storage. Optionally, data compression can be applied prior to the data encryption step to further reduce the size of the data to upload. We use a 256-bit cryptographically strong hash function (SHA2 [14]) for the content address and a 256-bit symmetric-key encryption (Salsa20 [17]) for data encryption.

3.1.4 Upload Queue

The system maintains an upload queue in memory for those data blocks to be uploaded to the backup server via CIFS protocol. First the content address of each data block is checked to see if the same data block is already on the server. If not the block will be scheduled to one of the uploading threads. A typical user machine contains many small files, which are less efficient to transfer over a standard network file system like CIFS compared to a custom upload protocol. We therefore perform data aggregation in the upload queue to pack small blocks into bigger packets of 2 MB before sending them over the network.

3.1.5 Server Application

A Python script on the server periodically checks any new files in a user's upload directory. ACL permission was set up so that the server application can read/write files in both public data pool and users' own upload directories. For each incoming packet, the server will disassemble the packet into original encrypted data blocks. If the block checksum matches its content address, the block will be moved to the public data pool. If a block has incorrect checksum due to network transmission error, it will be discarded and a new copy will be requested from the client.

Optionally, the server can choose to replicate its data to multiple public cloud storage vaults in the background. The use of cloud storage will be discussed in section 5.

4 Laptop Filesystem Statistics

The characteristics of the data, and the way in which it is typically organised and evolved, can have a significant effect on the performance of the algorithm and the implementation. We are aware of several relevant studies of filesystem statistics (e.g. [15]), however these have significant differences from our environment, such as the operating system, or the type of user. We therefore decided to conduct our own small study of laptop users within our typical target environment. We are reasonably confident that this represents "typical usage".

A data collection utility was distributed to voluntary participants in the university to compute a 160-bit cryptographically strong hash for each file on their Mac laptop, along with other meta information such as file size, directory depth etc. Each file was scanned multiple times to get hash values of the following block sizes:128KB, 256KB, 512KB and 1024KB as well as single file hash value. Filenames were not collected to maintain privacy. We grouped the stats gathered from each machine into three categories: `USR`, `APP` and `SYS`. All data within a user's home folder is labelled as `USR`, data in `/Applications` is classified as `APP`, and all the rest are labelled as `SYS`. This is to help us identify where the data duplication occurs. We would expect a high degree of inter-machine data redundancy among Application and System files, but not so much between users' own data. In a real backup system, the amount of data transfer for subsequent incremental backups is typically much smaller than that of the initial uploads. To help estimate the storage request of incremental backup we collected the statistics twice over a two-month period.

4.1 Key Statistics

We gathered filesystem statistics from 24 Mac machines within the university, all of them are running either OS X 10.5 or 10.6. Although this is a small sample, we believe that this is a good representation of a typical target environment. Key statistics of the data are given in table 1. The histogram of file sizes is given in figure 2. The file size distribution follows normal distribution. A further breakdown reveals that the majority (up to 95%) of the files are relatively small (less than 100 KB).

The presence of huge number of small files will likely impose a speed issue when backing up due to I/O overhead and network latency. In addition, a cloud service provider is likely to charge for each upload/download operation. Therefore direct uploading to a cloud storage may not be an economically viable option, as will be seen in section 5.1.

4.2 Backup Simulation

We simulated a backup process by backing up one machine a time to the backup server. After each simulated backup, the projected and actual storage was recorded and the data duplication rate calculated. This was repeated until all machines were added to the backup storage, and this clearly demonstrates the increasing savings (in space per machine) as more machines partici-

Machines	24
Files	20,332,615
Directories	4,607,966
Entries (File + Dir)	24,940,581
File Sizes	
Median	2.4 K
Average	77.9 K
Maximum	32.2 GB
Total	1.94 TB
File Category	
`USR`	1.22 TB (62.94%)
`APP`	149.32 GB (7.68%)
`SYS`	570.8 GB (29.38%)
Harddisk Size	
Average HD Size	290 GB
Average Used HD	115 GB

Table 1: Filesystem statistics from Mac laptops.

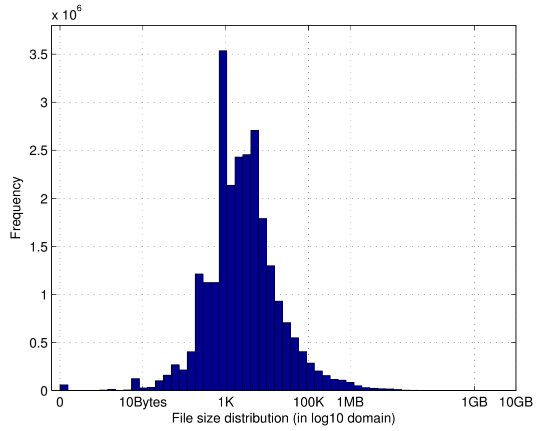


Figure 2: Histogram of file size distribution, the X-Axis is log_{10}(file size).

pate. Figure 3 shows the projected and actual storage for different numbers of machines. As expected, there is greater data sharing among System and Application files, but less so between users' own files.

4.3 Data Duplication

To measure data duplication (or redundancy) we define the data duplication rate as:

$$\frac{\text{Projected Storage Size-Actual Storage Size}}{\text{Projected Storage Size}} \times 100\%$$

where the projected storage size is the sum of all data to backup, actual storage size is the actual amount of storage required due to data de-duplication. For instance, if we need to store a total amount of 100 GB data from two machines A and B, and only 70 GB is needed for actual storage, then the data duplication rate for A and B is 30%.

Backup Block Size

File backup can be performed at whole file level or sub-file level where a file is split into a series of fixed-size blocks. There are two main benefits of performing backup at sub-file block level. First is the increased level of data de-duplication as a result of inter-file block sharing, which will use less storage space. Second, it is more efficient to backup certain type of files where only a small portion of the file content is constantly changing. For instance, database files and virtual machine disk images are usually a few GB in size and are known to subject to frequent updating. Modifications to those files are usually made in-place at various small portions of the file for performance reason. Block-level backup enables us to only backup those changed blocks. Finally, there is also a practical benefit: it is more reliable to upload a small block of data over remote network than a big file. Even the transfer fails due to network glitch, only the last block needs to be resent.

Despite the listed advantages, backup at block-level will incur some overhead that can be significant. Depending on the size of the block, the number of total stored objects could be much higher than whole file backup, which would be an concern when backing up to a cloud storage where the network I/O requests are charged (see section 5.1). Also extra storage is needed to record the block relationship which could offset some of the benefit of data de-duplication.

We tested different block sizes as well as whole file, single block backup in the simulation experiment. The result is plotted in figure 4.

As expected, all the sub-file blocks achieve a higher data duplication rate than single block, shown in figure 4a. The bigger the block size is, the lower data duplication rate due to less inter-file block sharing. For block size of 128KB, we the data duplication rate of 32.08%, which is 9.5% higher than the 29.31% of single block. This transfers into less storage used for all sub-file blocks (Figure 4b). However, the increased number of block objects could be quite substantial: for block size of 128KB, the total number of objects is 38m (million), 64.4% more than that of single block objects which is 24.94m (Figure 4c).

Overall	`USR`	`SYS`	`APP`
29.31%	8.91%	63.34%	63.06%

Table 2: Overall data duplication rate by category.

Directory Tree Duplication

As mentioned in section 2, the directory meta data is stored in directory objects. The size of all directory objects is 16.24 GB. With data de-duplication, the actual required storage is 6.4 GB, or 0.47% of total used storage. The collected stats also reveals that, among all 4,607,966 directory objects, only 1,052,338 unique ones are stored on the server. This suggests that up to 77% directory objects are shared. This strongly supports the value of sub-tree de-duplication.

Finally, we report overall data duplication rate in table 2.

Changes Over Time

Once an initial, full backup of a user machine is made, subsequent backups can be much faster as only changed data needs to be uploaded. To get an estimate of the file change rate, we collected and analysed the data for a second time towards the end of the two-month pilot experiment. We are mainly interested in the sizes of newly added files and changed files that will be included in the incremental backup. Files deleted since last scan were not included. The average daily and monthly per-machine data change rates calculated using a block size of 128KB are presented in table 3. In addition, we observed that the estimated monthly per-machine data change rate would raise from 17.17 GB to 20.61 GB if the backup is performed at file level. This confirms our earlier assumption that backing up at sub-file level can be more efficient in dealing with partial changes in large files. In our scenario it would reduce the amount of data to backup by 3.44 GB for each machine on a monthly basis.

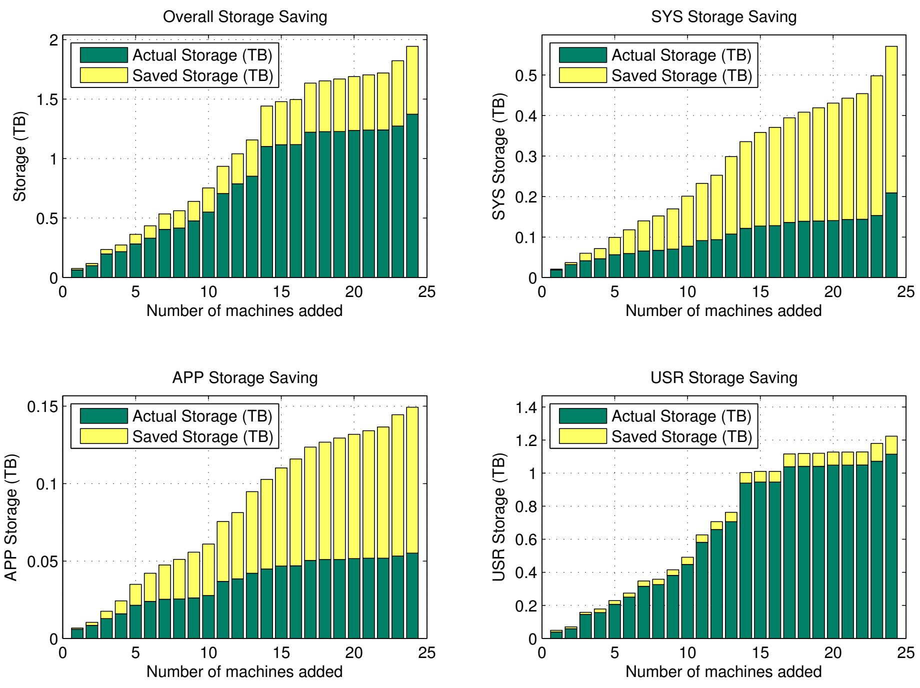


Figure 3: Projected and actual storage by number of machines added during simulated backup.

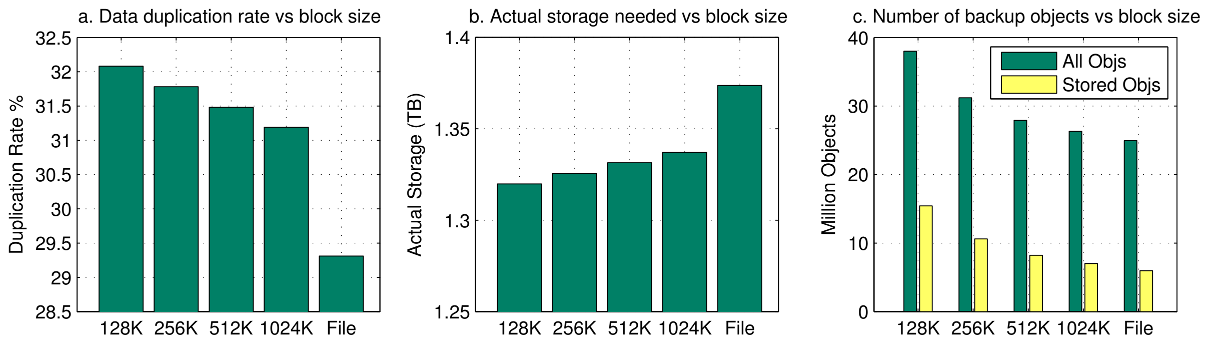


Figure 4: Data duplication rate and actual storage under different block sizes (File means the whole file is treated as a single block).

	Overall	USR	SYS	APP
Daily	0.57 GB	0.31 GB	0.10 GB	0.03 GB
Monthly	17.17 GB	9.20 GB	2.90 GB	0.86 GB

Table 3: Estimated daily and monthly (30 days) per-machine data change rates by category.

5 Using Cloud Storage

Backing up to cloud-based storage becomes increasing popular in recent years. The main benefits of using a cloud storage are lower server maintenance cost, cheaper long term operational cost, and sometimes enhanced data safety via a vendor's own geographically diverse data replication. However, there are still some obstacles to integrating cloud storage into a full-system backup solution:

1. Network bandwidth can be a bottle-neck: uploading data directly to a cloud storage can be very slow while requiring a reliable network connection.
2. The cloud interface has yet be standardised with each vendor offering its own security and data transfer models. In addition, many organisations prefer to have a backup policy that can utilise multiple cloud services to avoid vendor lock-in.
3. A cloud service provider is likely to impose a charge on individual data upload/download operations, which means backing up directly to a cloud can be very costly.

Despite these disadvantages, in the next section we will show that the proposed de-duplicated backup algorithm can effectively reduce the cost of backing up to a cloud storage by a large margin. Furthermore, in our backup architecture it is possible to adopt cloud storage as the secondary storage of the backup server (Figure 1), thereby largely ameliorating the above issues. In particular, the benefits of employing a cloud-based secondary storage are:

1. Backing up to local backup server can still be very fast with all the security features enabled, while the data replication to cloud storage can be performed in the background.
2. New cloud services can be added easily on the backup server to provide enhanced data safety and to reduce the risk of vendor lock-in.
3. Upload cost to cloud storage can be reduced via data aggregation techniques such as employed in [24].

5.1 Cost Saving: Data De-Duplication

In this section we measure the estimated cost of backup to a cloud storage in terms of the bill charged by a typical storage provider: Amazon S3[9]. Its data transfer model is based on standard HTTP protocol requests like `GET` and `PUT`.

For de-duplication backup, the client first needs to check if an object exists on the cloud via an `HEAD` operation. If not, the object is then uploaded via a `PUT` operation. Fortunately, if a file already exists on the server due to data de-duplication, we do not need to use the more expensive `PUT` operation, and no data will be uploaded.

Using the Amazon S3 price model[10] we plot the estimated cost of backing up 1.94 TB data from 24 machines to S3 in terms of US dollars in figure 5. The data de-duplication technology, coupled with data aggregation (see next section), is able to achieve 60% cost reduction for the initial data upload.

5.2 Cost Saving: Data Aggregation

Backing up directly to Amazon S3 can be very slow and costly due to large number of I/O operations and the way Amazon S3 charges for the uploads. Client side data aggregation packs small files into bigger packets before sending them to the cloud. We packed individual files into packets with a size of 2 MB each and compared the estimated cost against that of direct upload (see figure 6). The result shows that an overall of 25% of cost saving can be achieved via data aggregation alone. Moreover, even when cost is not an issue, as is the case when backing up to a departmental server, data aggregation can still speed up the process significantly. In our experiment we observed that the underlying file system (CIFS in this case) did not handle large amount of small I/O requests efficiently between the client and the server, even on a high-speed network. This resulted in a much slower backup speed. Uploading aggregated data to the server and unpacking the data there overcomes this problem.

5.3 Case Study: Six-Month Bill Using S3

Using the gathered information so far, we are able to estimate the cost of backing up all the machines we have to Amazon S3 over a six-month period. The initial upload of 1.94 TB data (25m objects) will cost \$434 without data de-duplication, together with storage cost (\$291) that is \$725. With data de-duplication, only a total of

[9]`https://s3.amazonaws.com/`

[10]As of writing this paper, data upload to Amazon S3 is charged at \$0.1 per GB. Data storage rate is \$0.15 per GB for the first 50 TB/Month of storage. Operating cost is \$0.01 per 1,000 requests for PUT operation and \$0.01 per 10,000 requests for HEAD operation, respectively. All prices quoted are from US Standard tier.

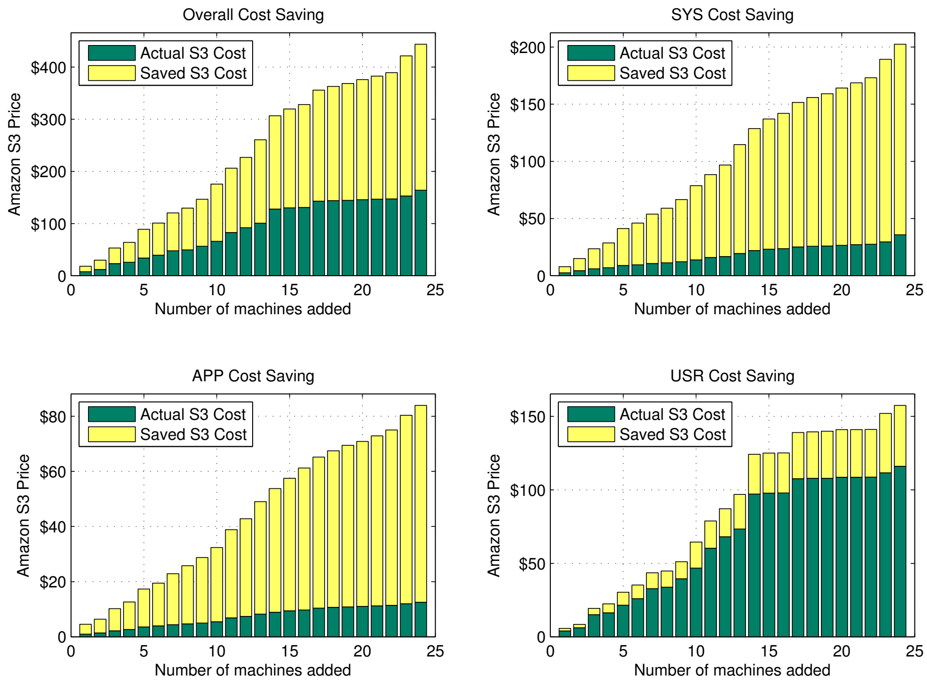


Figure 5: Estimated cost when backing up 24 machines to Amazon S3 (total cost for the initial backup of all machines)

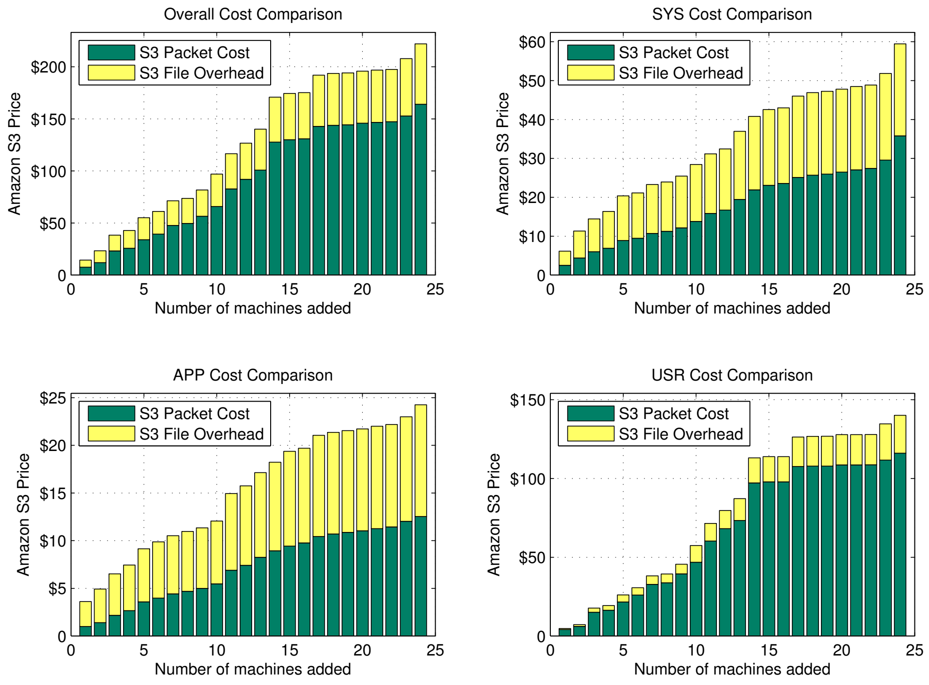


Figure 6: Estimated cost comparison of client side data aggregation (packet) and normal upload (file).

	Conventional		De-Duplicated	
Months	**Storage**	**Cost**	**Storage**	**Cost**
1	1.94 TB	$725	1.37 TB	$370
2	2.35 TB	$445	1.66 TB	$284
3	2.76 TB	$507	1.95 TB	$328
4	3.18 TB	$569	2.25 TB	$372
5	3.59 TB	$630	2.54 TB	$415
6	4.00 TB	$692	2.83 TB	$459
Total		$3,568		$2,228

Table 4: Estimated monthly and accumulated bills for backing up to Amazon S3 over a six-month period without and with the De-Duplicated backup. The monthly data change rate is estimated to be 0.49 TB for 24 machines using figures in table 3 with an estimated duplication rate of 29.31%. The monthly cost is the sum of S3 storage bill (currently $150 per TB) plus the estimated data upload cost from the pilot experiment.

$370 ($164 for upload and $206 for storage) is needed, saving $355 or 49.0% of the initial upload cost. Moreover, as only 1.37 TB of data needs to be uploaded, it is estimated to save 29.4% uploading time. The saving can be even greater over time as less cloud storage is used and billed. The accumulated costs of using Amazon S3 over the course of six months are $3,568 using conventional backup method, and $2,228 using de-deuplicated backup. Monthly estimated bills are presented in table 4.

6 Conclusions

We have shown that a typical community of laptop users share a considerable amount of data in common. This provides the potential to significantly decrease backup times, and storage requirements. However, we have shown that manual selection of the relevant data - for example, backing up only home directories - is a poor strategy; this fails to backup some important files, at the same time as unnecessarily duplicating other files.

We have presented a prototype backup program which achieves an optimal degree of sharing at the same time as maintaining confidentiality. This exploits a novel algorithm to reduce the number of files which need to be scanned and hence decreases backup times.

We have shown that typical cloud interfaces, such as Amazon S3 are not well suited to this type of application, due to the time and cost of typical transfers, and the lack of multi-user authentication to shared data. We have described a implementation using a local server which can avoid these problems by caching and pre-processing data before transmitting to the cloud. This is shown to achieve significant cost savings.

7 Acknowledgements

This project has been funded by the Edinburgh University *Initiating Knowledge Transfer Fund* (iKTF)[11] and the IDEA Lab [12]. Thanks to Toby Blake from the School of Informatics for support with the infrastructure, and to all those who contributed data.

[11]`http://www.eri.ed.ac.uk/commercial/developmentfunding/ikta.htm`

[12]`http://www.idea.ed.ac.uk/IDEA/Welcome.html`

References

[1] Backblaze: online backup [cited 2nd April 2010]. `http://www.backblaze.com/`.

[2] Backjack: online backup [cited 2nd April 2010]. `http://www.backjack.com`.

[3] Carbonite: online backup [cited 2nd April 2010]. `http://www.carbonite.com/`.

[4] Crashplan: online backup [cited 2nd April 2010]. `http://www.crashplan.com`.

[5] Hash collisions: The real odds [cited 31st March 2010]. `http://www.backupcentral.com/content/view/145/47/`.

[6] Humyo: online backup [cited 2nd April 2010]. `http://humyo.com/`.

[7] IDrive: remote data backup [cited 2nd April 2010]. `http://www.idrive.com`.

[8] JungleDisk: online backup [cited 2nd April 2010]. `http://www.jungledisk.com/`.

[9] Lessfs – a high performance inline data deduplicating filesystem for Linux [cited 2nd April 2010]. `http://www.lessfs.com`.

[10] Mac OS X Time Machine [cited 2nd April 2010]. `http://www.apple.com/macosx/what-is-macosx/time-machine.html`.

[11] SDFS: A deduplication file-system for Linux [cited 2nd April 2010]. `http://www.opendedup.org/`.

[12] Soonr: active backup [cited 2nd April 2010]. `http://www.soonr.com/`.

[13] Sarbanes-Oxley act of 2002, February 2002. `http://www.gpo.gov/fdsys/pkg/PLAW-107publ204/content-detail.html`.

[14] Federal information processing standards publications 180-2: Secure hash standard (SHS). Technical report, National Institute of Standards and Technology Gaithersburg, MD 20899-8900, October 2008. `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[15] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, pages 31–45, 2007. `http://www.usenix.org/events/fast07/tech/agrawal.html`.

[16] K. Bennett, C. Grothoff, T. Horozov, and I. Patrascu. Efficient sharing of encrypted data. In *In Proceedings of ASCIP 2002*, pages 107–120. Springer-Verlag, 2002. `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.19.9837`.

[17] D. Bernstein. The Salsa20 family of stream ciphers. *New Stream Cipher Designs*, pages 84–97, 2008. `http://dx.doi.org/10.1007/978-3-540-68351-3_8`.

[18] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in Windows 2000. In *Proceedings of 4th USENIX Windows Systems Symposium*. Usenix, 2000. `http://research.microsoft.com/apps/pubs/default.aspx?id=74261`.

[19] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the international conference on measurement and modeling of computer systems (SIGMETRICS)*, 2007. `http://research.microsoft.com/apps/pubs/default.aspx?id=74262`.

[20] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In *Proceedings of the 2009 Usenix Annual Technical Conference*. Usenix, June 2009. `http://www.usenix.org/events/usenix09/tech/full_papers/clements/clements.pdf`.

[21] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *In Proceedings of 22nd International Conference on Distributed Computing Systems (ICDCS*, 2002. `http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=75E78117EB6C02C4493CA49F28775D65?doi=10.1.1.8.7586&rep=rep1&type=pdf`.

[22] G. Gonsalves. Content addressable storage for encrypted shared backup. Master's thesis, School of Informatics, University of Edinburgh, 2009. `http://homepages.inf.ed.ac.uk/dcspaul/publications/CASFESB.pdf`.

[23] M. W. Storer, K. Greenan, D. D. Long, and E. L. Miller. Secure data deduplication. In *StorageSS '08: Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 1–10, New York, NY, USA, 2008. ACM. `http://portal.acm.org/citation.cfm?id=1456469.1456471#`.

[24] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, February 2009. `http://cseweb.ucsd.edu/~voelker/pubs/cumulus-fast09.pdf`.

[25] D. Wang, L. Wang, and J. Song. SEDBRS: A secure and efficient desktop backup and recovery system. *Data, Privacy, and E-Commerce, International Symposium on*, 0:304–309, 2007. `http://www.computer.org/portal/web/csdl/doi/10.1109/ISDPE.2007.27`.

The Margrave Tool for Firewall Analysis

Timothy Nelson
Worcester Polytechnic Institute
tn@cs.wpi.edu

Christopher Barratt
Brown University
cbarratt@cs.brown.edu

Daniel J. Dougherty
Worcester Polytechnic Institute
dd@cs.wpi.edu

Kathi Fisler
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

Writing and maintaining firewall configurations can be challenging, even for experienced system administrators. Tools that uncover the consequences of configurations and edits to them can help sysadmins prevent subtle yet serious errors. Our tool, Margrave, offers powerful features for firewall analysis, including enumerating consequences of configuration edits, detecting overlaps and conflicts among rules, tracing firewall behavior to specific rules, and verification against security goals. Margrave differs from other firewall-analysis tools in supporting queries at multiple levels (rules, filters, firewalls, and networks of firewalls), comparing separate firewalls in a single query, supporting reflexive ACLs, and presenting exhaustive sets of concrete scenarios that embody queries. Margrave supports real-world firewall-configuration languages, decomposing them into multiple policies that capture different aspects of firewall functionality. We present evaluation on networking-forum posts and on an in-use enterprise firewall-configuration.

1 Introduction

Writing a sensible firewall policy from scratch can be difficult; maintaining existing policies can be terrifying. Oppenheimer, Ganapathi, and Patterson [31] have shown that operator errors, specifically configuration errors, are a major cause of online-service failure. Configuration errors can result in lost revenue, breached security, and even physical danger to co-workers or customers. The pressure on system administrators is increased by the frenetic nature of their work environment [6], the occasional need for urgent changes to network configurations, and the limited window in which maintenance can be performed on live systems.

Many questions arise in checking a firewall's behavior: Does it permit or block certain traffic? Does a collection of policies enforce security boundaries and goals? Does a specific rule control decisions on certain traffic? What prevents a particular rule from applying to a packet? Will a policy edit permit or block more traffic than intended? These questions demand flexibility from firewall-analysis tools: they cover various levels of granularity (from individual rules to networks of policies), as well as reasoning about multiple versions of policies (to check the impact of edits). Margrave handles all these and more, offering more functionality than other published firewall tools.

Margrave's flexibility comes from thinking about policy analysis from an end-user's perspective. The questions that users wish to ask about policies obviously affect modeling decisions, but so does our form of answer. Margrave's core paradigm is *scenario finding*: when a user poses a query, Margrave produces a (usually exhaustive) set of scenarios that witness the queried behavior. Whether a user is interested in the impact of changes or how one rule can override another, scenarios concretize a policy's behavior. Margrave also allows queries to be built incrementally, with new queries refining the results from previous ones.

Margrave's power comes from choosing an appropriate model. Embracing both scenario-finding and multi-level policy-reasoning leads us to model policies in first-order logic. While many firewall-analysis tools are grounded in logic, most use propositional models for which analysis questions are decidable and efficient. In general, one cannot compute an exhaustive and finite set of scenarios witnessing first-order logic formulas. Fortunately, the formulas corresponding to many common firewall-analysis problems do yield such sets. Margrave identifies such cases automatically, thus providing exhaustive analysis for richer policies and queries than other tools. Demonstrating that firewall analyzers can benefit from first-order logic without undue cost is a key contribution of this paper.

Our other key contribution lies in how we decompose IOS configurations into policies for analysis. Single firewall configurations cover many functions, such as access filtering, routing, and switching. Margrave's IOS compiler generates separate policies for each task, thus enabling analysis of either specific functionality or whole-firewall behavior. Task-specific policies aid in isolating causes of problematic behaviors. Our firewall models support standard and most extended ACLs, static NAT, ACL-based and map-based dynamic NAT, static routing, and policy-based routing. Our support for state is limited to reflexive access-lists; it does not include general dynamic NAT, deep packet inspection, routing via OSFP, or adaptive policies. Margrave has an iptables compiler in development; other types of firewalls, such as Juniper's JunOS, fit our model as well.

A reader primarily interested in a tool description can read Sections 2, 6, and 7 for a sense of Margrave and how it differs from other firewall-analysis tools. Section 2 illustrates Margrave's query language and scenario-based output using a multi-step example. Section 3 describes the underlying theory (based on first-order logic), including our notion of policies. Section 4 shows how firewall questions map into Margrave. Section 5 describes the implementation, including the compiler for firewall-configurations and a query-rewriting technique that often improves performance. Section 6 presents experimental evaluation on both network-forum posts and an in-use enterprise firewall. Section 7 describes related work. Section 8 concludes with perspective and future work.

2 Margrave in Action on Firewalls

Margrave presents scenarios that satisfy user-specified queries about firewall behavior. Queries state a behavior of interest and optional controls on which data to consider when computing scenarios. Scenarios contain attributes of packet contents that make the query hold. A separate command language controls how scenarios are displayed. The extended example in this section highlights Margrave's features; Table 1 summarizes which of these features are supported by other available (either free or commercial) firewall analyzers. The Margrave website [22] contains sources for all examples.

In this paper, a *firewall* encompasses filtering (via access-lists), NAT transformation, and routing; we reserve the term *router* for the latter component. The IOS configuration in Figure 1 defines a simple firewall with only filtering. This firewall controls two interfaces (`fe0` and `vlan1`). Each has an IP address and an access-list to filter traffic as it enters the interface; in lines 3 and 7, the number (101 or 102) is a label that associates access rules (lines 9-16) with each interface, while the `in` keyword specifies that the rules should apply on entry. Rules are checked in order from top to bottom; the first rule whose conditions apply determines the decision on a packet. This firewall allows inbound web and mail traffic to the corresponding servers (the `.10` and `.11` hosts), but denies a certain blacklisted IP address (the `10.1.1.2` host). All traffic arriving at the inside-facing interface `vlan1` is allowed. As this filter is only concerned with packets as they arrive at the firewall, our queries refer to the filter as `InboundACL`.

```
1  interface fe0
2  ip address 10.150.1.1 255.255.255.254
3  ip access-group 101 in
4  !
5  interface vlan1
6  ip address 192.128.5.1 255.255.255.0
7  ip access-group 102 in
8  !
9  access-list 101 deny ip host 10.1.1.2 any
10 access-list 101 permit tcp
11             any host 192.168.5.10 eq 80
12 access-list 101 permit tcp
13             any host 192.168.5.11 eq 25
14 access-list 101 deny any
15 !
16 access-list 102 permit any
```

Figure 1: Sample IOS configuration

Basic Queries: All firewall analyzers support basic queries about which packets traverse the firewall. The following Margrave query asks for an inbound packet that `InboundACL` permits:

```
EXPLORE InboundACL:Permit(<req>)

SHOW ONE
```

Query 1

`EXPLORE` clauses describe firewall behavior; here, the behavior is simply to permit packets. `<req>` is shorthand for a sequence of variables denoting the components of a request (detailed in Section 4):

$$\langle \textit{ahostname}, \textit{src-addr-in}, \textit{src-port-in}, \textit{protocol}, \ldots\rangle.$$

Users can manually define this shorthand within Margrave; details and instructions for passing queries into Margrave are in the tool distribution [22]. `SHOW ONE` is an output-configuration command that instructs Margrave to display only a single scenario. The resulting output indicates the packet contents:

```
********* SOLUTION FOUND at size = 15
src-addr-in: IPAddress
protocol: prot-tcp
dest-addr-in: 192.168.5.10
src-port-in: port
exit-interface: interface
entry-interface: fe0
dest-port-in: port-80
length: length
ahostname: hostname-router
src-addr-out: IPAddress
message: icmpmessage
```

Result

	ITVal	Fireman	Prometheus	ConfigChecker	Fang/AlgoSec	Vantage
Which packets	✓	✓	✓	✓	✓	✓
User-defined queries	✓		?	✓	✓	$\checkmark^{\text{nip}}$
Rule Responsibility	✓	?	$\checkmark^{-}$	$\sim$	✓	✓
Rule Relationships	$\sim$	$\checkmark^{-}$	✓	$\checkmark^{-}$	$\checkmark^{\text{nip}}$	✓
Change-impact	?			✓	$\checkmark^{\text{nip}}$	$\checkmark^{-}$
First-order queries	?		?			?
Support NAT	✓		✓	✓	✓	
Support Routing	✓		✓	✓	✓	$\checkmark^{\text{nip}}$
Firewall Networks	✓	✓	✓	✓	✓	$\checkmark^{\text{nip}}$
Language integration						✓
Commercial Tool?	no	no	yes	no	yes	yes

Table 1: Feature comparison between Margrave and other available firewall-analysis tools. In each cell, ✓ denotes included features; $\checkmark^{\text{nip}}$ denotes features reported by the authors in private communication but not described in published papers; $\checkmark^{-}$ denotes included features with more limited scope than in Margrave; $\sim$ denotes features that can be simulated, but aren't directly supported; ? denotes cases for which we aren't sure about support. Section 7 describes nuances across shared features and discusses additional research for which tools are not currently available.

This scenario shows a TCP packet (line 3) arriving on the fast-ethernet interface (line 7), bound for the web server (line 4, with line 11 of Figure 1) on port 80 (line 8). The generic `IPaddress` in lines 2 and 11 should be read as "any IP address not mentioned explicitly in the policy"; lines 5 and 6 are similarly generic. Section 5 explains the `size=15` report on line 1.

A user can ask for additional scenarios that illustrate the previous query via the command `SHOW NEXT`: Once Margrave has displayed all unique scenarios, it responds to `SHOW NEXT` queries with `no results`.

To check whether the filter accepts packets from the blacklisted server, we constrain `src-addr-in` to match the blacklisted IP address and examine only packets that arrive on the external interface. Both `src-addr-in` and `entry-interface` are variable names in `<req>`. The `IS POSSIBLE?` command instructs Margrave to display false or true, rather than detailed scenarios.

```
EXPLORE
InboundACL:Permit(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface

IS POSSIBLE?
```
Query 2

In this case, Margrave returns false. Had it returned true, the user could have inspected the scenarios by issuing a `SHOW ONE` or `SHOW ALL` command.

Rule-level Reasoning: Tracing behavior back to the responsible rules in a firewall aids in both debugging and confirming that rules are fulfilling their intent. To support reasoning about rule effects, Margrave automatically defines two formulas for every rule in a policy (where R is a unique name for the rule):

- R`_matches(<req>)` is true when `<req>` satisfies the rule's conditions, and
- R`_applies(<req>)` is true when the rule both matches `<req>` and determines the decision on `<req>` (as the first matching rule within the policy).

Distinguishing these supports fine-grained queries about rule behavior. Margrave's IOS compiler constructs the R labels to uniquely reference rules across policies. For instance, ACL rules that govern an interface have labels of the form *hostname-interface-*`line`*#*, where *hostname* and *interface* specify the names of the host and interface to which the rule is attached and # is the line number at which the rule appears in the firewall configuration file.

The following query refines query 2 to ask for decision justification: the `EXPLORE` clause now asks for `Deny` packets, while the `INCLUDE` clause instructs Margrave to compute scenarios over the two `Deny` rules as well as the formulas in the `EXPLORE` clause:

```
EXPLORE
InboundACL:Deny(<req>) AND
10.1.1.2 = src-addr-in AND
fe0 = entry-interface
INCLUDE
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)

SHOW REALIZED
InboundACL1:Router-fe0-line9_applies(<req>),
InboundACL1:Router-fe0-line14_applies(<req>)
```
Query 3

The `SHOW REALIZED` command asks Margrave to display the subset of listed facts that appear in some resulting scenario. The following results indicate that the rule at line 9 does (at least sometimes) apply. More telling, however, the absence of the rule at line 14 (the catch-all deny) indicates that that rule *never* applies to any packet from the blacklisted address. Accordingly, we conclude that line 9 processes all blacklisted packets.

```
< InboundACL:line9_applies(<req>) >
```
Result

The INCLUDE clause helps control Margrave's performance. Large policies induce many rule-matching formulas; enabling these formulas only as needed trims the scenario space. SHOW REALIZED (and its dual, SHOW UNREALIZED) controls the level of detail at which users view scenarios. The lists of facts that do (or do not) appear in scenarios often raise red flags about firewall behavior (such as an unexpected port being involved in processing a packet). Unlike many verification tools, Margrave does not expect users to have behavioral requirements or formal security goals on hand. Lightweight summaries such as SHOW REALIZED try to provide information that suggests further queries.

Computing Overshadowed Rules through Scripting: Query 3 checks the relationship between two rules on particular packets. A more general question asks which rules *never* apply to *any* packet; we call such rules *superfluous*. The following query computes superfluous rules:

```
EXPLORE true
UNDER InboundACL
INCLUDE
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)

SHOW UNREALIZED
 InboundACL:router-fe0-line9_applies(<req>),
 InboundACL:router-fe0-line10_applies(<req>),
 InboundACL:router-fe0-line12_applies(<req>),
 InboundACL:router-fe0-line14_applies(<req>),
 InboundACL:router-vlan1-line16_applies(<req>)
```
Query 4

As this computation doesn't care about request contents, the EXPLORE clause is simply true. The heart of this query lies in the INCLUDE clause and the SHOW UNREALIZED command: the first asks Margrave to consider all rules; the second asks for listed facts that are never true in any scenario. UNDER clauses load policies referenced in INCLUDE but not EXPLORE clauses.

While the results tell us which rules never apply, they don't indicate which rules overshadow each unused rule. Such information is useful, especially if an overshadowing rule ascribes the opposite decision. Writing queries to determine justification for each superfluous rule, however, is tedious. Margrave's query language is embedded in a host language (Racket [13], a descendent of Scheme) through which we can write scripts over query results. In this case, our script uses a Margrave command to obtain lists of rules that yield each of *Permit* and *Deny*, then issues queries to isolate overshadowing rules for each superfluous rule. These are similar to other queries in this section. Scripts could also compute hotspot rules that overshadow a large percentage of other rules.

Change-Impact: Sysadmins edit firewall configurations to provide new services and correct emergent problems. Edits are risky because they can have unexpected consequences such as allowing or restricting traffic that the edit should not have affected. Expecting sysadmins to have formal security requirements against which to test policy edits is unrealistic. In the spirit of lightweight analyses that demand less of users, Margrave computes scenarios illustrating packets whose decision or applicable rule changes in the face of edits.

For example, suppose we add the new boldface rule below to access-list 101 (the line numbers start with 14 to indicate that lines 1–13 are identical to those in Figure 1):

```
14 access-list 101 deny tcp
15        host 10.1.1.2 host 192.168.5.10 eq 80
```

If we call the modified filter InboundACL_new, the following query asks whether the original and new InboundACLs ever disagree on Permit decisions:

```
EXPLORE
(InboundACL:Permit(<req>) AND
 NOT InboundACL_new:Permit(<req>)) OR
(InboundACL_new:Permit(<req>) AND
 NOT InboundACL:Permit(<req>)))

IS POSSIBLE?
```
Query 5

Margrave returns false, since the rule at line 9 always overrides the new rule. If instead the new rule were:

```
14 access-list 101 deny tcp
15        host 10.1.1.3 host 192.168.5.10 eq 80
```

Margrave would return true on query 5. The corresponding scenarios show packet headers that the two firewalls treat differently, such as the following:

```
********* SOLUTION FOUND at size = 15
src-addr-in: 10.1.1.3
protocol: prot-tcp
dest-addr-in: 192.168.5.10
src-port-in: port
exit-interface: interface
entry-interface: fe0
dest-port-in: port-80
```
Result

As we might expect, this scenario involves packets from 10.1.1.3. A subsequent query could confirm that no other hosts are affected.

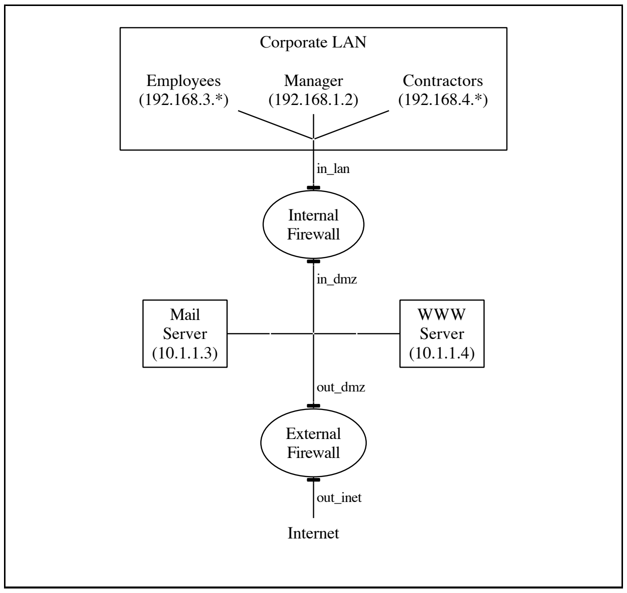


Figure 2: A small-business network-topology

Networks of Firewalls: So far, our examples have considered only single firewalls. Margrave also handles networks with multiple firewalls and NAT. Figure 2 shows a small network with web server, mail server, and two firewalls to establish a DMZ. The internal firewall performs both NAT and packet-filtering, while the external firewall only filters. The firewall distinguishes machines for employees (`192.168.3.*`), contractors (`192.168.4.*`), and a manager (`192.168.1.2`). This example captures the essence of a real problem posted to a networking help-forum.

```
1  hostname int
2  !
3  interface in_dmz
4  ip address 10.1.1.1 255.255.255.0
5  ip nat outside
6  !
7  interface in_lan
8  ip access-group 102 in
9  ip address 192.168.1.1 255.255.0.0
10 ip nat inside
11 !
12 access-list 102 permit tcp any any eq 80
13 access-list 102 deny any
14 !
15 ip nat inside source list 1 interface
16               in_dmz overload
17 access-list 1 permit 192.168.1.1 0.0.255.255
18 !
19 ip route 0.0.0.0 0.0.0.0 in_dmz
20
```
Internal Firewall

Lines 15–17 in the internal firewall apply NAT to traffic from the corporate LAN.[1] Line 11 in the external firewall blacklists a specific external host (`10.200.200.200`). Despite the explicit rule on lines 19–20 in the external firewall, the manager cannot access the web. We have edited the configurations to show only those lines relevant to the manager and web traffic.

```
1  hostname ext
2  !
3  interface out_dmz
4  ip access-group 103 in
5  ip address 10.1.1.2 255.255.255.0
6  !
7  interface out_inet
8  ip access-group 104 in
9  ip address 10.200.1.1 255.255.0.0
10 !
11 access-list 104 deny 10.200.200.200
12 access-list 104 permit tcp any host 10.1.1.4
13                 eq 80
14 access-list 104 deny any
15 !
16 access-list 103 deny ip any
17                 host 10.200.200.200
18 access-list 103 deny tcp any any eq 23
19 access-list 103 permit tcp host 192.168.1.2
20                 any eq 80
21 access-list 103 deny any
22
```
External Firewall

The following query asks "What rules deny a connection from the manager's PC (line 2) to port 80 (line 10) somewhere outside our network (line 8) other than the blacklisted host (line 9)?"

```
1  EXPLORE prot-TCP = protocol AND
2  192.168.1.2 = fw1-src-addr-in AND
3  in_lan = fw1-entry-interface AND
4  out_dmz = fw2-entry-interface AND
5  hostname-int = fw1 AND
6  hostname-ext = fw2 AND
7
8  fw1-dest-addr-in IN 10.200.0.0/255.255.0.0
9  NOT 10.200.200.200 = fw1-dest-addr-in AND
10 port-80 = fw1-dest-port-in AND
11
12 internal-result(<reqfull-1>) AND
13
14 (NOT passes-firewall(<reqpol-1>) OR
15  internal-result(<reqfull-2>) AND
16  NOT passes-firewall(<reqpol-2>))
17
18 UNDER InboundACL
19 INCLUDE
20 InboundACL:int-in_lan-line-12_applies
21   (<reqpol-1>),
22 InboundACL:int-in_lan-line-17_applies
23   (<reqpol-1>),
24 InboundACL:ext-out_dmz-line-19_applies
25   (<reqpol-2>),
26 InboundACL:ext-out_dmz-line-21_applies
27   (<reqpol-2>),
28 InboundACL:ext-out_dmz-line-24_applies
29   (<reqpol-2>)
```
Query 6

Lines 12–16 capture both network topology and the effects of NAT. The `internal-result` and `passes-firewall` formulas capture routing in the face

[1] In this example, we use the 10.200.* private address space to represent the public IP addresses.

of NAT and passing through the complete firewall (including routing, NAT and ACLs) whose hostname appears in the request, respectively; Section 4 describes them in detail. The variables sent to the two `passes-firewall` formulas through `<reqpol-1>` and `<reqpol-2>` encode the topology: for example, these shorthands use the same variable name for *dest-addr-out* in the internal firewall and *src-addr-in* in the external firewall. The `fw1-entry-interface` and `fw2-entry-interface` variables (bound to specific interfaces in lines 3–4) appear as the entry interfaces in `<reqpol-1>` and `<reqpol-2>`, respectively.

A `SHOW REALIZED` command over the `INCLUDE` terms (as in query 3) indicates that line 21 of the external firewall configuration is denying the manager's connection. Asking Margrave for a scenario for the query (using the `SHOW ONE` command) reveals that the internal firewall's NAT is changing the packet's source address:

```
...
fw1-src-addr-out=fw2-src-addr_=
  fw2-src-addr-out: 10.1.1.1
fw1-src-addr_=fw1-src-addr-in: 192.168.1.2
```
Result

The external firewall rule (supposedly) allowing the manager to access the Internet (line 19) uses the internal pre-NAT source address; it never matches the post-NAT packet. Naïvely editing the NAT policy, however, can leak privileges to contractors and employees. Change-impact queries are extremely useful for confirming that the manager, and *only* the manager, gain new privileges from an edit. An extended version of this example with multiple fixes and the change-impact queries, is provided in the Margrave distribution.

Summary: These examples illustrate Margrave's ability to reason about both combinations of policies and policies at multiple granularities. The supported query types include asking which packets satisfy a condition (query 1), verification (query 2), rule responsibility (query 3), rule relationships (query 4) and change-impact (query 5). A formal summary of the query language and its semantics is provided with the Margrave distribution.

3 Defining Scenarios

Margrave views a *policy* as a mapping from requests to decisions. In a firewall, requests contain packet data and some routing data, while decisions include *Permit* and *Deny* (for ACLs), *Drop* (for routing), and a few others. Policies often refer to relationships between objects, such as "permit access by machines *on the same subnet*". Queries over policies often require quantification: "*Every* host on the local subnet can access *some* gateway router". First-order logic extends propositional logic with relational formulas (such as `SameSubnet(121.34.42.133,121.34.42.166)`) and quantifiers (∀ and ∃). For firewall policies, the available relations include the decisions, R`_matches` and R`_applies` (as shown in Section 2) and unary relations capturing sets of IP addresses, ports, and protocols.

Margrave maps both policies and queries into first-order logic formulas. To answer a query, Margrave first conjoins the query formula with the formulas for all policies referenced in the query, then computes solutions to the combined formula. A *solution* to a first-order formula contains a set of elements to quantify over (the *universe*) and two mappings under which the formula is true: one maps each relation to a set of tuples over the universe, and another maps each unquantified variable in the query to an element of the universe.[2] For example, the formula

$$\forall x\ host(x) \implies \exists y\ (router(y) \land CanAccess(x,y))$$

says that "every host can access some router". One solution has a universe of $\{h1, r1, r2\}$ and relation tuples *host(h1)*, *router(r1)*, *router(r2)*, and *CanAccess(h1,r2)* (the formula has no unquantified variables). Other solutions could include more hosts and routers, with more access connections between them. Solutions may map multiple variables to the same universe element. This is extremely useful for detecting corner cases in policy analysis; while humans often assume that different variables refer to different objects, many policy errors lurk in overlaps (such as a host being used a both web server and mail server). *Scenarios* are simply solutions to the formula formed of a query and the policies it references.

In general, checking whether a first-order formula has a solution (much less computing them all) is undecidable. Intuitively, the problem lies in determining a sufficient universe size that covers all possible solutions. This problem is disconcerting for policy analysis: we would like to show users an exhaustive set of scenarios to help them ensure that their policies are behaving as intended in all cases. Fortunately, Margrave can address this problem in most cases; Section 5 presents the details.

4 Mapping Firewalls to the Theory

There is a sizeable gap between the theory in Section 3 and a policy in a real-world language, such as the example in Figure 1. To represent policies in the theory, we must describe the shapes of requests, the available decisions, what relations can appear in formulas, and how policy rules translate into formulas. Section 2 used several relations relevant to firewalls, such

[2]In logical terms, a solution combines a first-order model and an environment binding free variables to universe elements.

```
(Policy InboundACL uses IOS-vocab
 (Rules
  ...
  (Router-fe0-line10 =
   (Permit hostname, ...) :-
   (hostname-Router hostname)
   (fe0 entry-interface)
   (IPAddress src-addr-in)
   (prot-tcp protocol)
   (Port src-port-in)
   (192.168.5.10 dest-addr-in)
   (port-80 dest-port-in))
  ...)
 (RComb FAC))
```

Figure 3: A Margrave policy specification

as `passed-firewall`. Margrave defines these relations and other details automatically via several mechanisms.

Policies: Figure 3 shows part of the result of compiling the IOS configuration in Figure 1 to Margrave's intermediate policy language. The fragment captures the IOS rule on line 10. `(Permit hostname, ...)` specifies the decision and states a sequence of variable names corresponding to a request. The `:-` symbol separates the decision from the conditions of the rule. Formula `(prot-tcp protocol)`, for example, captures that TCP is the expected protocol for this rule. Margrave represents constants (such as decisions, IP addresses, and protocols) as elements of singleton unary relations. A scenario that satisfies this rule will map the *protocol* variable to some element of the universe that populates the `prot-tcp` relation. The other conditions of the original rule are captured similarly. The `(RComb FAC)` at the end of the policy tells Margrave to check the policy rules in order (`FAC` stands for "first applicable"). The first line of the policy ascribes the name `InboundACL`.

Decomposing IOS into policies: Figure 4 shows our high-level model of IOS configurations. Firewalls perform packet filtering, packet transformation, and internal routing; the first two may occur at both entry to and exit from the firewall. Specifically, packets pass through the inbound ACL filter, inside NAT transformation, internal routing, outside NAT transformation, and finally the outbound ACL filter on their way through the firewall. The intermediate stages define additional information about a packet (as shown under the stage names): inside NAT may yield new address and port values; internal routing determines the next-hop and exit interface; outside NAT may yield further address and port values.

Internal routing involves five substages, as shown in Figure 6. Margrave creates policies (à la Figure 3) for each of the five substages. The `-Switching` policies determine whether a destination is directly connected to the firewall; the `-Routing` policies bind the `next-hop` IP address for routing. In addition, Margrave generates four policies called `InboundACL`, `OutboundACL`, `InsideNAT`, and `OutsideNat`. The two `-ACL` policies contain filtering rules for all interfaces.

Requests and Decisions: Margrave automatically defines a relation for each decision rendered by each of the 9 subpolicies (e.g., `InboundACL:Permit` in query 1). Each relation is defined over requests, which contain packet headers, packet attributes, and values generated in the intermediate stages; the boxes in Figure 4 collectively list the request contents. As Margrave is not stateful, it cannot update packet headers with data from intermediate stages. The contents of a request reflect the intermediate stages' actions: for example, if the values of `src-addr_` and `src-addr-out` are equal, then `OutsideNAT` did not transform the request's packet. Currently, Margrave shares the same request shape across all 9 subpolicies (even though `InboundACL`, for example, only examines the packet header portion).

Flows between subpolicies: Margrave encodes flows among the 9 subpolicies through three relations (over requests) that capture the subflows marked in Figure 4.

- Internal routing either assigns an exit interface and a next-hop to a packet or drops the packet internally. Margrave uses a special exit-interface value to mark dropped packets; the `int-dropped` relation contains requests with this special exit-interface value. Any request that is not in `int-dropped` successfully passes through internal routing.
- Unlike internal routing, NAT never drops packets. At most, it transforms source and destination ports and addresses. Put differently, NAT is a function on packets. `internal-result` captures this function: it contains all requests whose `next-hop`, `exit-interface`, and `OutsideNAT` components are consistent with the packet header and `InsideNAT` components (as if the latter were inputs to a NAT function).
- ACLs permit or deny packets. The relation `passes-firewall` contains requests that the two ACLs permit, are in `internal-result` (i.e., are consistent with NAT), and are not in `int-dropped` (i.e., are not dropped in internal routing).

Our IOS compiler automatically defines each of these relations as a query in terms of the 9 IOS subpolicies (capturing topology as in query 6). Margrave provides a `RENAME` command that saves query results under a user-specific name for use in later queries. Users can name any set of resulting scenarios in this manner.

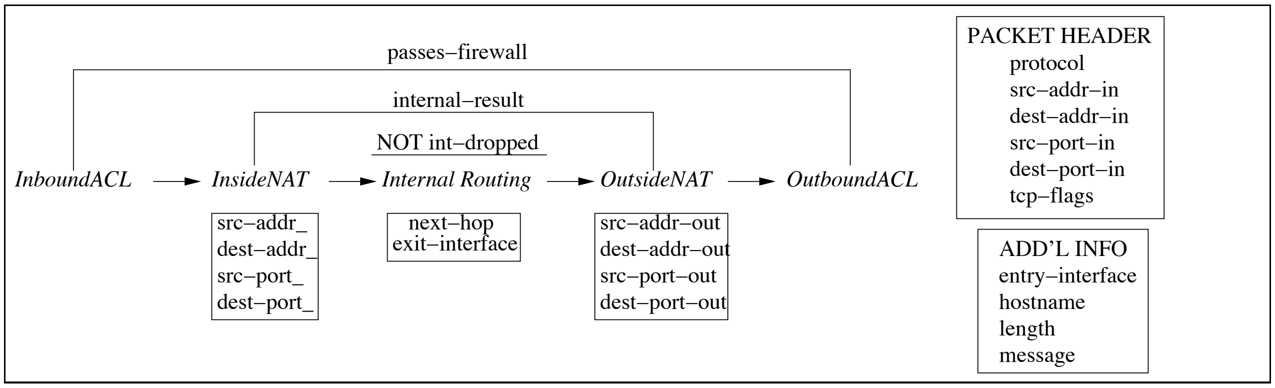


Figure 4: Margrave's decomposition of firewall configurations

```
(PolicyVocab IOS-vocab
 (Types
  (Interface : interf-drop
               (interf-real vlan1 fe0))
  (IPAddress :
    192.128.5.0/255.255.255.0
    10.1.1.0/255.255.255.254
    192.168.5.11
    192.168.5.10
    10.1.1.2)
  (Protocol : prot-ICMP prot-TCP prot-UDP)
  (Port: port-25 port-80)
  (Decisions Permit Deny ...)
  ...
  (disjoint-all Protocol)
  (nonempty Port)
  ...
)
```

Figure 5: A Margrave vocabulary specification

Vocabularies: The 9 subpolicies share ontology about ports and IP addresses. Margrave puts domain-knowledge common to multiple policies in a *vocabulary* specification; the first line of a policy specification references its vocabulary through the `uses` keyword. Figure 5 shows a fragment of the vocabulary for IOS policies: it defines datatypes (such as `Protocol`) and their elements (correspondingly, `prot-ICMP`, `prot-TCP`, `prot-UDP`).

Vocabularies also capture domain constraints such as "all protocols are distinct" or "there must be at least one port" (both shown in Figure 5). While these constraints may seem odd, they support Margrave's scenario-finding model. Some potential "solutions" (as described in Section 3) are nonsensical, such as one which assigns two distinct numbers to the same physical port. Domain constraints rule out nonsensical scenarios.

Generalizing Beyond Firewalls

The policy- and vocabulary-specifications in Figures 3 and 5 show how to map specific domains into Margrave. Datatypes, constraints, and rules capture many other kinds of policies, including access-control policies, hypervisor configurations, and product-line specifications. Indeed, this general-purpose infrastructure is another advantage of Margrave over other firewall-analysis tools: Margrave can reason about interactions between policies from multiple languages for different configuration concerns. For example, if data security depends on a particular interaction between a firewall and an access-control policy, both policies and their interaction can be explored using Margrave. We expect this feature to become increasingly important as enterprise applications move onto the cloud and are protected through the interplay of multiple policies from different sources.

5 Implementation

Margrave consists of a frontend read-eval-print loop (REPL) written in Racket [13] and a backend written in Java. The frontend handles parsing (of queries, commands, policies, and vocabularies) and output presentation. The actual analysis and scenario generation occurs in the backend.

5.1 The Scenario-Finding Engine

Margrave's backend must produce sets of solutions to first-order logic formulas. We currently use a tool called Kodkod [32] that produces solutions to first-order formulas using SAT solving.[3] SAT solvers handle propositional formulas. Kodkod bridges the gap from first-order to propositional formulas by asking users for a finite universe-size; under a finite universe-size, first-order formulas translate easily to propositional ones. Figure 7 shows an example of the rewriting process. Every solution produced using a bounded size is legitimate (in logical terms, our analysis is *sound*). However, analysis will miss solutions that require a universe larger than the given size (in logical terms, it is not *complete*).

[3]Within Kodkod, we use a SAT-solver called SAT4J [8].

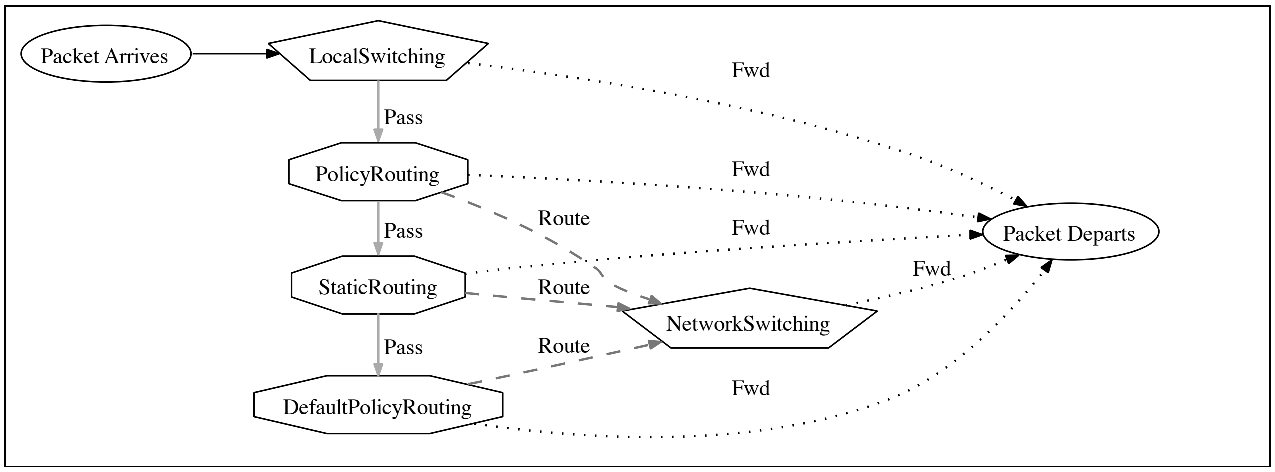


Figure 6: Internal flow of packets within a router. Edges are labeled with decisions rendered by the policies at the source of the edge. Routing policies determine the next-hop IP address, while switching policies send traffic to directly to a connected device.

Fortunately, most firewall queries (including those in this paper) correspond to formulas with no universal ($\forall$) quantifiers. For such formulas, the number of existentially-quantified variables provides a sufficient universe size to represent all solutions. Margrave automatically supplies Kodkod with the universe bound for such formulas. For queries that do not have this form, such as "can *every* host reach some other machine on the network", either Margrave or the user must supply a universe size for the analysis. The query language has an optional `CEILING` clause whose single argument is the desired universe size. If `CEILING` is omitted, Margrave uses a default of 6. Experience with Kodkod in other domains suggests that small universe sizes can yield useful scenarios [15]. If Margrave can compute a sufficient bound but the user provides a lower `CEILING`, Margrave will only check up to the `CEILING` value. Whenever Margrave cannot guarantee that scenario analysis is complete, it issues a warning to the user. The `size=15` statement in the first line of scenarios shown in Section 2 report the universe-size under which Margrave generated the scenario.

`CEILING` settings may impact the results of commands. Margrave includes a `SHOW UNREALIZED` command that reports relations that are not used in any resulting scenario. However, a relation T might be unpopulated at one `CEILING` value yet populated at a higher value. For example, in the formula $\exists x \neg T(x)$, T is never used at `CEILING` 1, but can be realized at `CEILING` 2. Margrave users should only supply `CEILING` values if they appreciate such consequences.

Overall, we believe sacrificing exhaustiveness for the expressive power of first-order logic in policies and queries is worthwhile, especially given the large number of practical queries that can be checked exhaustively.

5.2 Rewriting Firewall Queries

Under large universe sizes, both the time to compute scenarios and the number of resulting scenarios increase. The latter puts a particular burden on the end-user who has to work through the scenarios. Query language constructs like `SHOW REALIZED` summarize details about the scenarios in an attempt to prevent the exhaustive from becoming exhausting. However, query optimizations that reduce universe sizes have more potential to target the core problem.

Most firewall queries have the form $\exists\, \overline{req}\ \alpha$, where α typically lacks quantifiers. Requests have 16 or 20 components (as shown in Figure 4), depending on whether they reference `internal-result`. Margrave therefore analyzes all-existential queries under a universe size of 16 or 20. However, these queries effectively reference a *single request* with attributes as detailed in α. This suggests that we could rewrite this query with a single quantified variable for a request and additional relations that encode the attributes. For example:

$$\exists\, pt\_in\ \exists\, pt\_out : route(pt\_in, pt\_out)$$

becomes

$$\exists\, pkt : is\_ptIn(pkt, i) \wedge is\_ptOut(pkt, o) \wedge route(i, o)$$

Effectively, these new relations lift attributes from the individual packet fields to the packet as a whole.

Formulas rewritten in this way require a universe size of only 1, for which scenario generation stands to be much faster and to yield fewer solutions. The tradeoff, however, lies in the extra relations that Margrave introduces to lift attributes to the packet level. Additional relations increase the time and yield of scenario computations, so the rewriting is not guaranteed to be a net win.

The original sentence:

$$\forall x\ host(x) \implies \exists y\ (router(y) \wedge CanAccess(x, y))$$

Assume a universe of size 2 with elements A and B. Expand the $\forall$-formula with a conjunction over each of A and B for x:

$$\begin{aligned} host(A) &\implies \exists y\ (router(y) \wedge CanAccess(A, y)) \wedge \\ host(B) &\implies \exists y\ (router(y) \wedge CanAccess(B, y)) \end{aligned}$$

Next, expand each $\exists$-formula with a disjunction over each of A and B for y:

$$\begin{aligned} host(A) &\implies (router(A) \wedge CanAccess(A, A)) \vee \\ &\qquad (router(B) \wedge CanAccess(A, B)) \wedge \\ host(B) &\implies (router(A) \wedge CanAccess(B, A)) \vee \\ &\qquad (router(B) \wedge CanAccess(B, B)) \end{aligned}$$

Replace each remaining formula with a propositional variable (e.g., $router(A)$ becomes p_2):

$$\begin{aligned} p_1 &\implies (p_2 \wedge p_3) \vee \\ &\qquad (p_4 \wedge p_5) \wedge \\ p_6 &\implies (p_2 \wedge p_7) \vee \\ &\qquad (p_4 \wedge p_8) \end{aligned}$$

Figure 7: Converting a first-order formula to a propositional one at a bounded universe size

Rules	# Vars	Min Size	Not Tupled	Tupled
100	3	3	694ms	244ms
1000	14	6	7633ms	1221ms
1000	14	10	17659ms	1219ms
1000	14	14	32116ms	1205ms

Table 2: Run-time impact of TUPLING on ACL queries. The first column contains the number of rules in each ACL. The second column lists the number of existentially-quantified variables in the query; we include one 3-variable (non-firewall) query to illustrate the smaller gains on smaller variable counts. The 14-variable ACLs are older firewall examples with smaller request tuples. The "Min Size" column indicates the universe size for the smallest scenario that satisfied the query. Larger minimum sizes have a larger search space.

Table 2 presents experimental results on original versus rewritten queries. In practice, we find performance improves when the query is unsatisfiable or the smallest model is large. A user who expects either of these conditions to hold can enable the rewriting through a query-language flag called `TUPLING`. All performance figures in this paper were computed using `TUPLING`.

6 Evaluation

We have two main goals in evaluating Margrave. First, we want to confirm that our query language and its results support debugging real firewall configuration-problems; in particular, the scenarios should accurately point to root causes of problems. We assume a user who knows enough firewall basics to ask the questions underlying a debugging process (Margrave does not, for example, pre-emptively try queries to automatically isolate a problem). Second, we want to check that Margrave has reasonable performance on large policies, given that we have traded efficient propositional models for richer first-order ones.

We targeted the first goal by applying Margrave to problems posted to network-configuration help-forums (Sections 6.1 and 6.2). Specifically, we phrased the poster's reported problem through Margrave queries and sought fixes based on the resulting scenarios. In addition, we used Margrave to check whether solutions suggested in follow-up posts actually fixed the problem without affecting other traffic. The diversity of firewall features that appear in forum posts demanded many compiler extensions, including reflexive access-lists and TCP flags. That we could do this purely at the compiler level attests to the flexibility of Margrave's intermediate policy- and vocabulary-languages (Section 4).

We targeted the second goal by applying Margrave to an in-use enterprise firewall-configuration containing several rule sets and over 1000 total rules (Section 6.3). Margrave revealed some surprising facts about redundancy in the configuration's behavior. Individual queries uniformly execute in seconds.

Notes on Benchmarking Our figures report Margrave's steady-state performance; they omit JVM warmup time. Policy-load times are measured by loading different copies of the policy to avoid caching bias. All performance tests were run on an Intel Core Duo E7200 at 2.53 Ghz with 2 GB of RAM, running Windows XP Home. Performance times are the mean over at least 100 individual runs; all reported times are $\pm$200ms at the 95-percent confidence level. Memory figures report private (i.e., not including shared) consumption.

6.1 Forum Help: NAT and ACLs

"My servers cannot get access into the internet, even though I will be able to access the website, or even FTP... I don't really know what's wrong. Can you please help? Here is my current configuration..."

In our first forum example [4], the poster is having trouble connecting to the Internet from his server. He believes that NAT is responsible, and has identified the router as the source of the problem. The configuration included with the post appears in Figure 8 (with a slight semantics-preserving modification[4]).

A query (not shown) confirms that the firewall is blocking the connection. Our knowledge of firewalls indicates that packets are rejected either enroute to, or on return from, the webserver. Queries for these two cases are similar; the one checking for response packets is:

```
EXPLORE
NOT src-addr-in
  IN 192.168.2.0/255.255.255.0 AND
FastEthernet0 = entry-interface AND
prot-TCP = protocol AND
port-80 = src-port-in AND
internal-result(<reqfull>) AND
passes-firewall(<reqpol>)

IS POSSIBLE?
```
Query 7

Margrave reports that packets to the webserver are permitted, but responses are dropped. The resulting scenarios all involve source ports 20, 21, 23, and 80 (easily confirmed by re-running the query with a `SHOW REALIZED` command asking for only the port numbers). This is meaningful to a sysadmin: an outgoing web request is always made from an *ephemeral* port, which is never less than 1024. This points to the problem: the router is rejecting all returning packets. ACL 102 (Figure 8, lines 25–29) ensures that the server sees only incoming HTTP, FTP, and TELNET traffic, at the expense of rejecting the return traffic for any connections that the server initiates.

Enabling the server to access other webservers involves allowing packets *coming from* the proper destination ports. Methods for achieving this include:

1. Permit TCP traffic *from* port 80, via the edit:

```
access-list 102 permit tcp
            any host 209.172.108.16 eq 23
access-list 102 permit tcp any eq 80 any
access-list 102 deny tcp
            any host 209.172.108.16
```

2. Allow packets whose *ack* flags are set via the *established* keyword (or, in more recent versions, the *match-all +ack* option). This suggestion guards against spoofing a packet's source port field and allows servers to listen on unusual ports.

3. Use stateful monitoring of the TCP protocol via reflexive access-lists or the *inspect* command. This guards against spoofing of the TCP ACK flag.

[4]We replaced named interface references in static NAT statements with actual IP addresses; our compiler does not support the former.

Follow-up posts in the forum suggested options 1 and 3. Margrave can capture the first two options and the reflexive access-list approach in the third (it does not currently support *inspect* commands). For each of these, we can perform verification queries to establish that the `InboundACL` no longer blocks return packets, and we can determine the extent of the change through a change-impact query.

Space precludes showing the reflexive ACL query in detail. Reflexive ACLs allow return traffic from hosts to which prior packets were permitted. Margrave encodes prior traffic through a series of `connection-` relations over requests. Intuitively, a request is in a `connection-` relation only if the same request with the source- and destination-details reversed would pass through the firewall. Although the connection state is dynamic in practice, its stateless definition enables Margrave to handle it naturally through first-order relations.

Performance: Loading each version of the configuration took between 3 and 4 seconds. The final change-impact query took under 1 second. After loading, running the full suite of queries (including those not shown) required 2751ms. The memory footprint of the Java engine (including all component subpolicies) was 50 MB (19 MB JVM heap, 20 MB JVM non-heap).

6.2 Forum Help: Routing

"there should be a way to let the network `10.232.104.0/22` access the internet, kindly advise a solution for this..."

In our second example [29], the poster is trying to create two logical networks: one "primary" (consisting of `10.232.0.0/22` and `10.232.100.0/22`) and one "secondary" (consisting of `10.232.4.0/22` and `10.232.104.0/22`). These logical networks are connected through a pair of routers (TAS and BAZ) which share a serial interface (Figure 9). Neither logical network should have access to the other, but both networks should have access to the Internet—the primary via `10.232.0.15` and the secondary via `10.232.4.10`.

The poster reports two problems: first, the two components of the primary network—`10.232.0.0/22` and `10.232.100.0/22`—cannot communicate with each other; second, the network `10.232.104.0/22` cannot access the Internet. The poster suspects errors in the TAS router configuration (omitted for sake of space).

We start with the first problem. The following query confirms that network `10.232.0.0/22` cannot reach `10.232.100.0/22` via the serial link. The `hostname` formulas introduce names for each individual router

```
name-server 207.47.4.2
name-server 207.47.2.178
!
interface FastEthernet0
ip address 209.172.108.16 255.255.255.224
ip access-group 102 in
ip nat outside
speed auto
full-duplex
!
interface Vlan1
ip address 192.168.2.1 255.255.255.0
ip nat inside
!
ip route 0.0.0.0 0.0.0.0 209.172.108.1
!
ip nat pool localnet 209.172.108.16 prefix-length 24
ip nat inside source list 1 pool localnet overload
ip nat inside source list 1 interface FastEthernet0
ip nat inside source static tcp 192.168.2.6 80 209.172.108.16 80
ip nat inside source static tcp 192.168.2.6 21 209.172.108.16 21
ip nat inside source static tcp 192.168.2.6 3389 209.172.108.16 3389
!
access-list 1 permit 192.168.2.0 0.0.0.255
access-list 102 permit tcp any host 209.172.108.16 eq 80
access-list 102 permit tcp any host 209.172.108.16 eq 21
access-list 102 permit tcp any host 209.172.108.16 eq 20
access-list 102 permit tcp any host 209.172.108.16 eq 23
access-list 102 deny tcp any host 209.172.108.16
```

Figure 8: The original configuration for the forum post for Section 6.1

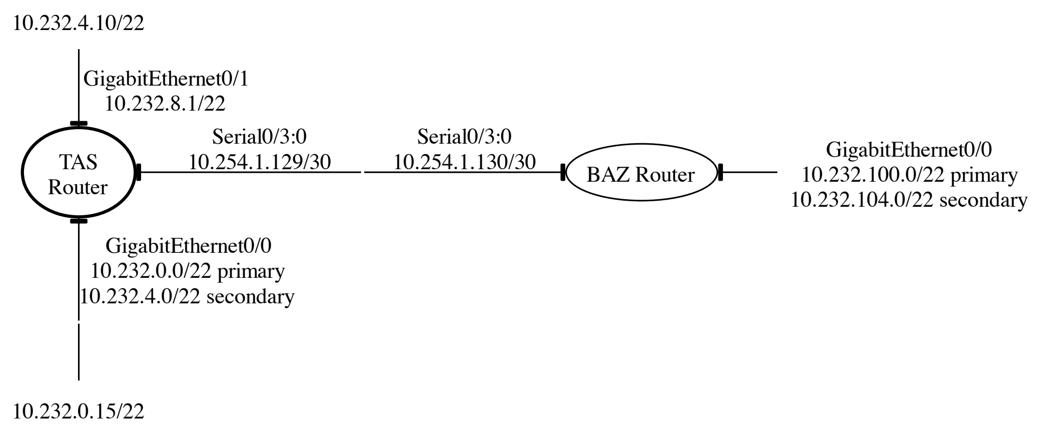


Figure 9: Structure of the network for the forum post for Section 6.2

based on the hostname specification in the IOS configuration; these names appear in the `tasvector-` and `bazvector-` requests. (The `-full-` requests extend the corresponding `-pol-` requests with additional variables needed for `internal-routing`.

```
1  EXPLORE hostname-tas = tas AND
2  hostname-baz = baz AND
3
4  internal-result(<tasvectorfull-fromtas>) AND
5  internal-result(<bazvectorfull-fromtas>) AND
6  passes-firewall(<tasvectorpol-fromtas>) AND
7  passes-firewall(<bazvectorpol-fromtas>) AND
8
9  GigabitEthernet0/0 = tas-entry-interface AND
10 tas-src-addr-in IN
11   10.232.0.0/255.255.252.0 AND
12 tas-dest-addr-in IN 10.232.100.0/255.255.252.0
13 AND "Serial0/3/0:0" = tas-exit-interface AND
14
15 "Serial0/3/0:0" = baz-entry-interface AND
16 GigabitEthernet0/0 = baz-exit-interface
17
18 IS POSSIBLE?
```

Query 8

Margrave returns false, which means that no packets from `10.232.0.0/22` reach `10.232.100.0/22` along this network topology.

By the topology in Figure 9, packets reach the TAS router first. We check whether packets pass through TAS by manually restricting query 8 to TAS (by removing lines 2, 5, 7, 14, and 15); Margrave still returns false. Firewall knowledge suggests three possible problems with the TAS configuration: (1) internal routing could be sending the packets to an incorrect interface, (2) internal routing could be dropping the packets, or (3) the ACLs could be filtering out the packets. Margrave's formulas for reasoning about internal firewall behavior help eliminate these cases: by negating `passed-firewall` on line 6, we determine that the packet does pass through the firewall, so the problem lies in the interface or next-hop assigned during routing. This example highlights the utility of not only having access to these formulas, but also having the ability to negate (or otherwise manipulate) them as any other subformula in a query.

To determine which interfaces the packets are sent on, we relax the query once again to remove the remaining reference to `Serial0/3/0:0` (on line 12) and execute the following `SHOW REALIZED` command:

```
SHOW REALIZED
     GigabitEthernet0/0 = exit-interface,
     "Serial0/3/0:0" = exit-interface,
     GigabitEthernet0/1 = exit-interface
```

Query 9

The output contains only one interface name:

```
{ GigabitEthernet0/0[exit-interface] }
```

Result

According to the topology diagram, packets from `10.232.0.0/22` to `10.232.100.0/22` should be using exit interface `Serial0/3/0:0`; the results, instead, indicate exit interface `GigabitEthernet0/0`. Firewall experience suggests that the router is either switching the correct next-hop address (`10.254.1.130`) to the wrong exit interface, or using the wrong next-hop address. The next query produces the next-hop address:

```
1  EXPLORE hostname-tas = tas AND
2  internal-result(<tasvectorfull-fromtas>) AND
3  passes-firewall(<tasvectorpol-fromtas>) AND
4  GigabitEthernet0/0 = tas-entry-interface AND
5  tas-src-addr-in IN
6    10.232.0.0/255.255.252.0 AND
7  tas-dest-addr-in IN 10.232.100.0/255.255.252.0
8
9  INCLUDE
10 10.232.0.15 = tas-next-hop,
11 10.232.4.10 = tas-next-hop,
12 tas-next-hop IN 10.254.1.128/255.255.255.252,
13 tas-next-hop IN 10.232.8.0/255.255.252.0
14
15 SHOW REALIZED
16 10.232.0.15 = tas-next-hop,
17 10.232.4.10 = tas-next-hop,
18 tas-next-hop IN 10.232.8.0/255.255.252.0,
19 tas-next-hop IN 10.254.1.128/255.255.255.252
```

Query 10

```
{ 10.232.0.15[tas-next-hop] }
```

Result

The next-hop address is clearly wrong for the given destination address. To determine the extent of the problem, we'd like to know whether *all* packets from the given source address are similarly misdirected. That question is too strong, however, as `LocalSwitching` may (rightfully) handle some packets. To ask Margrave for next-hops targeted by some source packet that `LocalSwitching` ignores, we replace line 7 in query 10 with:

```
NOT LocalSwitching:Forward(<routingpol-tas>)
```

Query 11

This once again highlights the value of exposing `LocalSwitching` as a separate relation. The revised query yields the same next-hop, indicating that all non-local packets are routing to `10.232.0.15`, despite the local routing policies. A simple change fixes the problem: insert the keyword **default** into the routing policy:

```
route-map internet permit 10
match ip address 10
set ip default next-hop 10.232.0.15
```

This change ensures that packets are routed to the Internet only as a last resort (i.e., when static destination-based routing fails). Running the original queries against the new specification confirms that the primary subnets now have connectivity to each other. Another query checks that this change does not suddenly enable the primary sub-network `10.232.0.0/22` to reach the secondary sub-network `10.232.4.0/22`.

Now we turn to the poster's second problem: the secondary network `10.232.4.0/22` still cannot access the Internet. As before, we confirm this then compute the next-hop and exit interface that TAS assigns to traffic from the secondary network with an outside destination. The following query (with `SHOW REALIZED` over interfaces and potential next-hops) achieves this:

```
EXPLORE
tas = hostname-tas AND

internal-result2(<tasvectorfull-fromtas>) AND
firewall-passed2(<tasvectorpol-fromtas>) AND

GigabitEthernet0/0 = tas-entry-interface AND
tas-src-addr-in IN
  10.232.4.0/255.255.252.0 AND

NOT tas-dest-addr-in IN
  10.232.4.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.104.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.0.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.232.100.0/255.255.252.0 AND
NOT tas-dest-addr-in IN
  10.254.1.128/255.255.255.252 AND
NOT tas-dest-addr-in IN
  192.168.1.0/255.255.255.0 AND
NOT tas-dest-addr-in IN
  10.232.8.0/255.255.252.0
```

Query 12

```
{   gigabitethernet0/0[tas-exit-interface],
     10.232.4.10[tas-next-hop] }
```

Result

The next-hop for the secondary network's Internet gateway is as expected, but the exit-interface is unexpectedly `GigabitEthernet0/0` (instead of `GigabitEthernet0/1`). In light of this scenario, the network diagram reveals a fundamental problem: the gateway `10.232.4.10` should be "on" the same network as the `GigabitEthernet0/1` interface (address `10.232.8.1/22`); otherwise `LocalSwitching` will send the packet to the wrong exit interface.

This problem can be resolved by changing the address of either the `GigabitEthernet0/1` interface or the next-hop router (`10.232.4.10`). We chose the latter, selecting an arbitrary unused address in the `10.232.8.0/22` network:

```
route-map internet permit 20
match ip address 20
set ip default next-hop 10.232.8.10
```

Re-running the queries in this new configuration confirms that both goals are now satisfied.

Performance: Loading each version of the configuration took between 3 and 4 seconds. Query 12 took 351

Query	Time (ms)
Permit pkt from addr X on interface Y?	1587
Previous with rule responsibility	23317
Change-impact after 1 decision edit	3167
Previous with rule responsibility	24039
Detect all superfluous rules	22578
List overshadows per rule in previous	72178

Table 3: Run-time performance of various queries on the enterprise ACLs. For the change-impact query, we switched the decision from *deny* to *permit* on one non-superfluous rule. The overshadowing-rules computation asked only for overshadows with the opposite decision.

ms. After loading, running the full suite of queries (including those not shown) finished in 8725ms. The memory footprint of the Java engine (including all component subpolicies) was 74 MB (49 MB JVM heap, 21 MB JVM non-heap).

6.3 Enterprise Firewall Configuration

Our largest test case to date is an in-use enterprise iptables configuration. In order to stress-test our IOS compiler, we manually converted this configuration to IOS. The resulting configuration contains ACLs for 6 interfaces with a total of 1108 `InboundACL` rules (not counting routing subpolicies). The routing component of this firewall was fairly simple; we therefore focus our performance evaluation on `InboundACL`.

From a performance perspective, this paper has illustrated three fundamentally different types of queries: (1) computing over a single policy or network with just the default relations (which-packets and verification queries), (2) computing over a single policy or network while including additional relations (rule-responsibility and rule-relationship queries), and (3) computing over multiple, independent policies or networks (change-impact queries). The third type introduces more variables than the first two (to represent requests through multiple firewalls); it also introduces additional relations to capture the policies of multiple firewalls. The second type has the same number of variables, but more relations, than the first type. We therefore expect the best performance on the first type, even under `TUPLING`.

Table 3 reports run-time performance on each type of query over the enterprise firewall-configuration. Loading the policy's `InboundACL` component required 10694ms and consumed 51 MB of memory. Of that, 40 MB was JVM heap and 7 MB was JVM non-heap.

Section 2 described how we compute superfluous rules through scripting. For this example, these queries yielded surprising results: 900 of the 1108 rules in `InboundACL` were superfluous. Even more, 270 of the superfluous rules were (at least partially) overshadowed by a rule with a different decision. The sysadmins who provided the configuration found these figures shocking and subsequently expressed interest in Margrave.

7 Related Work

Studies of firewall-configuration errors point to the need for analysis tools. Oppenheimer, *et al.* [31] survey failures in three Internet services over a period of several months. For two of these services, operator error—predominately during configuration edits—was the leading cause of failure. Furthermore, conventional testing fails to detect many configuration problems. Wool [35] studies the prevalence of 12 common firewall-configuration errors. Larger rule-sets yield a much higher ratio of errors to rules than smaller ones; Wool concludes that complex rule sets are too difficult for a human administrator to manage unaided.

Mayer, Wool and Ziskind [26, 27] and Wool [34] describe a tool called Fang that has evolved into a commercial product called the AlgoSec Firewall Analyzer [3]. AlgoSec supports most of the same analyses as Margrave, covering NAT and routing, but it does not support first-order queries or integration with a programming language. AlgoSec captures packets that satisfy queries through sub-queries, which are a form of abstract scenarios.

Marmorstein and Kearns' [23, 24] ITVal tool uses Multi-way Decision Diagrams (MDDs) to execute SQL-like queries on firewall policies. ITVal supports NAT, routing, and chains of firewall policies. Later work [25] supports a useful query-free analysis: it generates an equivalence relation that relates two hosts if identical packets (modulo source address) from both are treated identically by the firewall. This can detect policy anomalies and help administrators understand their policies. Additional debugging aids in later work includes tracing decisions to rules and showing examples similar to scenarios. Margrave is richer in its support for change-impact and first-order queries.

Al-Shaer *et al.*'s ConfigChecker [1, 2] is a BDD-based tool that analyses networks of firewalls using CTL (temporal logic) queries. Rules responsible for decisions can be isolated manually through queries over sample packets. For performance reasons, the tool operates at the level of policies, rather than individual rules (other of the group's papers do consider rule-level reasoning); Margrave, in contrast, handles both levels.

Bhatt *et al.*'s Vantage tool [5, 9, 10] supports change-impact on rule-sets and other user-defined queries over combinations of ACLs and routing; it does not support NAT. Some of their evaluations [9] exploit change-impact to isolate configuration errors. This work also supports generating ACLs from specifications, which is not common in firewall-analysis tools.

Liu and Gouda [20, 21] introduce Firewall Decision Diagrams (FDDs) to answer SQL-like queries about firewall policies. FDDs are an efficient variant of BDDs for the firewall packet-filtering domain. Extensions of this work by Khakpour and Liu [17] present algorithms for many firewall analysis discussed in this paper, including user-defined queries, rule responsibility, and change-impact, generally in light of NAT and routing. A downloadable tool is under development.

Yuan, *et al.*'s Fireman tool [36] analyzes large networks of firewall ACLs using Binary Decision Diagrams (BDDs). Fireman supports a fixed set of analyses, including whitelist and blacklist violations and computing conflicting, redundant, or correlated rules between different ACLs. Fireman examines all paths between firewalls at once, but does not consider NAT or internal routing. Margrave's combination of user-defined queries and support for NAT and routing makes it much richer. Oliveira, *et al.* [30] extend Fireman with NAT and routing tables. Their tool, Prometheus, can also determine which ACL rules are responsible for a misconfiguration. It does not handle change-impact across firewalls, though it does determine when different paths through the same firewall render different decisions for the same packet. In certain cases, Prometheus suggests corrections to rule sets that guarantee desired behaviors. Margrave's query language is richer.

Verma and Prakash's FACE tool [33] aids both configuration of distributed firewalls and analyzing existing distributed firewalls expressed in iptables. It supports user-defined queries, as well as a form of change-impact over multiple firewalls. Its depth-first-search approach to propagating queries through a network resembles Mayer, Ziskind, and Wool's work. It does not handle routing or NAT. The tool is no longer available.

Gupta, LeFevre and Prakash [14] give a framework for the analysis of heterogeneous policies that is similar to ours. While both works provide a general policy-analysis language inspired by SQL, there are distinct differences. Their tool, SPAN, does not allow queries to directly reference rule applicability and the work does not discuss request-transformations such as NAT. However, SPAN provides tabular output that can potentially be more concise than Margrave's scenario-based output. SPAN is currently under development.

Lee, Wong, and Kim's NetPiler tool [18, 19] analyzes the flow graph of routing policies. It can be used to both simplify and detect potential errors in a network's routing configurations. The authors have primarily applied NetPiler to BGP configurations, which address the propagation of routes rather than the passage of packets. However, their methods could also be applied to firewall policies. Margrave does not currently support BGP, though its core engine is general enough to support them.

Jeffrey and Samak [16] present a formal model and algorithms for analyzing rule-reachability and cyclicity in iptables firewalls. This work does not address NAT or more general queries about firewall behavior.

Eronen and Zitting [11] perform policy analysis on Cisco router ACLs using a Prolog-based Constraint Logic Programming framework. Users are allowed to define their own custom predicates (as in Prolog), which enables analysis to incorporate expert knowledge. The Prolog queries are also first-order. This work is similar to ours in spirit, but is limited to ACLs and does not support NAT or routing information.

Youssef *et al.* [7] verify firewall configurations against security goals, checking both for configurations that violate goals and goals that configurations fail to cover. The work does not handle NAT or routing.

Margrave as described in this paper extends an earlier tool of the same name [12] developed by Tschantz, Meyerovich, Fisler and Krishnamurthi. The original Margrave targeted simple access-control policies, encoding them as propositional formulas that we analyzed using BDDs. Attempts to model enterprise access-control policies inspired the shift to first-order models embodied in the present tool. Not surprisingly, there is an extensive literature on logic-based tools for access-control policies; our other papers [12, 28] survey this literature.

8 Perspective and Future Work

Margrave is a general-purpose policy analyzer. Its most distinctive features lie in and arise from embracing scenario finding over first-order models. First-order languages provide the expressive power of quantifiers and relations for capturing both policies and queries. Expressive power generally induces performance cost. By automatically computing universe bounds for key queries, however, Margrave gets the best of both worlds: first-order logic's expressiveness with propositional logic's efficient analysis. Effectively, Margrave distinguishes between propositional *models* and propositional *implementations*. Most logic-based firewall-analysis tools conflate these choices.

First-order modeling lets Margrave uniformly capture information about policies at various levels of granularity. This paper has illustrated relations capturing policy decisions, individual rule behavior, and the effects of NAT and internal routing. The real power of our first-order modeling, however, lies in building new relations from existing ones. Each of the relations capturing behavior internal to a firewall (`passes-firewall`, `internal-routing`, and `int-dropped`) is defined within Margrave's query language and exported to the user through standard Margrave commands. While our firewall compilers provide these three automatically, users can add their own relations in a similar manner. Technically, Margrave allows users to define their own named views (in a database sense) on collections of policies. Thus, Margrave embraces policy-analysis in the semantic spirit of databases, rather than just the syntactic level of SQL-style queries.

Useful views build on fine-grained atomic information about policies. Margrave's unique decomposition of IOS configurations into subpolicies for nine distinct firewall functions provides that foundation. Our pre-defined firewall views would have been prohibitively hard to write without a clean way to refer to components of firewall functionality. Margrave's intermediate languages for policies and vocabularies, in turn, were instrumental in developing the subpolicies. Both languages use general relational terms, rather than domain-specific ones. Vocabularies allow authors to specify decisions beyond those typically associated with policies (such as *Permit* and *Deny*). Our IOS compiler defines separate decisions for the different types of flows out of internal routing, such as whether packets are forwarded internally or translated to another interface. The routing views are defined in terms of formulas capturing these decisions. The policy language defines the formulas through rules that yield each decision (our rule language is effectively stratified Datalog). Had we defined Margrave as a firewall-specific analyzer, rather than a general-purpose one, we likely would have hardwired domain-specific concepts that did not inherently support this decomposition.

User-defined decisions and views support extending Margrave from within. Integrating Margrave into a programming language supports external extension via scripting over the results of commands. Margrave produces scenarios as structured (XML) objects that can be traversed and used to build further queries. `SHOW REALIZED` produces lists of results over which programs (such as superfluous rule detection in Section 2) can iterate to generate additional queries. Extending our integration with iterators over scenarios would yield a more policy-specific scripting environment.

In separate projects, we have applied Margrave to other kinds of policies, including access-control, simple hypervisors, and product-line configuration. Margrave's general-purpose flexibility supports reasoning about *interactions* between firewalls and other types of policies (increasingly relevant in cloud deployments). This is another exciting avenue for future work.

Margrave's performance is reasonable, but slower than other firewall analyzers. This likely stems partly from additional variables introduced during the encoding into propositional logic. In particular, we expect Margrave will scale poorly to large networks of firewalls, as our formulas grow linearly with the number of firewalls. Our use of SAT-solving instead of BDDs may be another factor, though Jeffrey and Samak's comparisons between these for firewall analysis [16] are inconclusive. Exploring alternative backends—whether based on BDDs or other first-order logic solvers—is one area for future work. However, we believe the more immediate questions lie at the modeling level. For example:

- Firewall languages include stateful constructs such as *inspect*. Existing firewall analysis tools, including Margrave, largely ignore state (we are limited to reflexive ACLs). How do we effectively model and reason about state without sacrificing performance?
- Modeling IP addresses efficiently is challenging. Many tools use one propositional variable per bit; Margrave instead uses one per IP address. This makes it harder to model arithmetic relationships on IP addresses (i.e., subranges), though it provides finer-grained control over which IP addresses are considered during analysis. Where is the sweet-spot in IP-address handling?

Margrave is in active development. We are extending our firewall compilers to support VPN and BGP. We would like to automatically generate queries for many common problems (such as overshadowing rule detection and change-impact). Section 2 also hinted at a problem with reusing queries in the face of policy edits: the compiler names rules by line-numbers, so edits may invalidate existing queries. We need to provide better support for policy-management including regression testing.

Acknowledgments:

Support for this research came from several National Science Foundation grants. Cisco supported an early phase of this project. We thank John Basik, Jeff Coady, Mark Dieterich, Jason Montville and Richard Silverman for sysadmins' perspectives on this project. Craig Wills explained how to report performance data. Our LISA shepherd, Matt Disney, provided useful suggestions. In compiling our related work, we contacted many authors with questions about their projects. We thank them for their prompt and cheerful responses and hope we have represented their work accurately; any errors are our own.

References

[1] Ehab S. Al-Shaer and Hazem H. Hamed. Firewall Policy Advisor for Anomaly Discovery and Rule Editing. In *Integrated Network Management*, pages 17–30, 2003.

[2] Ehab S. Al-Shaer and Hazem H. Hamed. Discovery of Policy Anomalies in Distributed Firewalls. In *IEEE Conference on Computer Communications*, 2004.

[3] The AlgoSec Firewall Analzyer. `www.algosec.com`.

[4] azsquall. "ACL and NAT conflict each other. router stop working". `www.networking-forum.com/viewtopic.php?f=33&t=7635`, August 2008. Access Date: July 20, 2010.

[5] Sruthi Bandhakavi, Sandeep Bhatt, Cat Okita, and Prasad Rao. End-to-end network access analysis. Technical Report HPL-2008-28R1, HP Laboratories, November 2008.

[6] Rob Barrett, Eser Kandogan, Paul P. Maglio, Eben M. Haber, Leila Takayama, and Madhu Prabaker. Field Studies of Computer System Administrators: Analysis of System Management Tools and Practices. In *ACM Conference on Computer Supported Cooperative Work*, pages 388–395, 2004.

[7] Nihel Ben Youssef, Adel Bouhoula, and Florent Jacquemard. Automatic Verification of Conformance of Firewall Configurations to Security Policies. In *IEEE Symposium on Computers and Communications*, pages 526 – 531, July 2009.

[8] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 2010. To appear.

[9] Sandeep Bhatt, Cat Okita, and Prasad Rao. Fast, Cheap, and in Control: A Step Towards Pain-Free Security! In *Large Installation System Administration Conference*, pages 75–90, 2008.

[10] Sandeep Bhatt and Prasad Rao. Enhancements to the Vantage Firewall Analyzer. Technical Report HPL-2007-154R1, HP Laboratories, June 2008.

[11] Pasi Eronen and Jukka Zitting. An expert system for analyzing firewall rules. In *Proceedings of the Nordic Workshop on Secure IT Systems*, pages 100–107, 2001.

[12] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering*, pages 196–205, 2005.

[13] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR2010-1, PLT Inc., June 7, 2010. `racket-lang.org/tr1/`.

[14] Swati Gutpa, Kristen LeFevre, and Atul Prakash. SPAN: A unified framework and toolkit for querying heterogeneous access policies. In *USENIX Workshop on Hot Topics in Security*, 2009.

[15] Daniel Jackson. *Software Abstractions*. MIT Press, 2006.

[16] Alan Jeffrey and Taghrid Samak. Model Checking Firewall Policy Configurations. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2009.

[17] Amir R. Khakpour and Alex X. Liu. Quantifying and querying network reachability. In *Proceedings of the International Conference on Distributed Computing Systems*, June 2010.

[18] Sihyung Lee, Tina Wong, and Hyong S. Kim. Improving Dependability of Network Configuration through Policy Classification. In *IEEE/IFIP Conference on Dependable Systems and Networks*, 2008.

[19] Sihyung Lee, Tina Wong, and Hyong S. Kim. NetPiler: Detection of Ineffective Router Configurations. *IEEE Journal on Selected Areas in Communications*, 27(3):291–301, 2009.

[20] Alex X. Liu and Mohamed G. Gouda. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems*, 19(8), August 2008.

[21] Alex X. Liu and Mohamed G. Gouda. Firewall policy queries. *IEEE Transactions on Parallel and Distributed Systems*, 20(6):766–777, June 2009.

[22] The Margrave Policy Analzyer. `www.margrave-tool.org/v3/`.

[23] Robert Marmorstein and Phil Kearns. A Tool for Automated iptables Firewall Analysis. In *USENIX Annual Technical Conference*, 2005.

[24] Robert Marmorstein and Phil Kearns. An Open Source Solution for Testing NAT'd and Nested iptables Firewalls. In *Large Installation System Administration Conference*, 2005.

[25] Robert Marmorstein and Phil Kearns. Firewall Analysis with Policy-Based Host Classification. In *Large Installation System Administration Conference*, 2006.

[26] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analysis Engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[27] Alain Mayer, Avishai Wool, and Elisha Ziskind. Offline firewall analysis. *International Journal of Information Security*, 2005.

[28] Timothy Nelson. Margrave: An Improved Analyzer for Access-Control and Configuration Policies. Master's thesis, Worcester Polytechnic Institute, April 2010.

[29] oelolemy. "problem with policy based routing-urgent please !". `www.experts-exchange.com/Networking/Network_Management/Q_24113014.html`, February 2009. Access Date: July 20, 2010.

[30] Ricardo M. Oliveira, Sihyung Lee, and Hyong S. Kim. Automatic detection of firewall misconfigurations using firewall and network routing policies. In *DSN Workshop on Proactive Failure Avoidance, Recovery and Maintenance*, 2009.

[31] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[32] Emina Torlak and Daniel Jackson. Kodkod: A Relational Model Finder. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 632–647, 2007.

[33] Pavan Verma and Atul Prakash. FACE: A Firewall Analysis and Configuration Engine. In *Proceedings of the Symposium on Applications and the Internet*, 2005.

[34] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *Proceedings of the USENIX Security Symposium*, 2001.

[35] Avishai Wool. A Quantitative Study of Firewall Configuration Errors. *Computer*, 37(6):62–67, 2004.

[36] L. Yuan, J. Mai, Z. Su, H. Chen, C-N. Chuah, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *IEEE Symposium on Security and Privacy*, 2006.

Towards Automatic Update of Access Control Policy

Jinwei Hu[†‡]*, Yan Zhang*[†]*, and Ruixuan Li*[‡⋆]
[†]*Intelligent Systems Laboratory, School of Computing and Mathematics*
University of Western Sydney, Sydney 1797, Australia
[‡]*Intelligent and Distributed Computing Laboratory, School of Computer Science and Technology*
Huazhong University of Science and Technology, Wuhan 430074, China
jwhu@hust.edu.cn   rxli@hust.edu.cn   yan@scm.uws.edu.au
[⋆] *Corresponding author*

Abstract

Role-based access control (RBAC) has significantly simplified the management of users and permissions in computing systems. In dynamic environments, systems are subject to changes, so that the associated configurations need to be updated accordingly in order to reflect the systems' evolution. Access control update is complex, especially for large-scale systems; because the updated system is expected to meet necessary constraints.

This paper presents a tool, RoleUpdater, which answers administrators' high-level update request for role-based access control systems. RoleUpdater is able to automatically check whether a required update is achievable and, if so, to construct a reference model. In light of this model, administrators could fulfill the changes to RBAC systems. RoleUpdater is able to cope with practical update requests, e.g., that include role hierarchies and administrative rules in effect. Moreover, RoleUpdater can also provide minimal update in the sense that no redundant changes are implemented.

1 Introduction

Role-based access control (RBAC) [11, 35] simplifies access control management. In an RBAC system, users are assigned to roles such as manager and employee, and a role in turn is defined as a set of permissions. The key to RBAC is that users are assigned to roles and thus obtain roles' permissions, instead of being assigned permissions directly. Essentially, an RBAC configuration manages three kinds of relations: a user-role relation, a role-role relation, and a role-permission relation. The *user-role* relation assigns users to roles. The *role-role* relation describes how roles' permissions are inherited by other roles. The *role-permission* relation describes which permissions are accorded to each role. An RBAC system consists of two components, the RBAC configuration and the administration configuration. A running example RBAC system, which is used throughout the paper, is comprised of the RBAC configuration in Figure 1 and the administration configuration in Figure 2.

The role-role relation needs to be a partial order over roles; usually we refer to the role-role relation as a role hierarchy. The role hierarchy embodies two inheritance relationships among roles. Take the RBAC configuration in Figure 1 for example. (r_1, r_7) belongs to the hierarchy and we say r_1 is senior to r_7; it means that r_1 inherits all permissions of r_7 (i.e., p_3 and p_4) and that all members of r_1 are also members of r_7 or in other words, r_7 inherits all users of r_1.

RBAC is able to model a wide range of access control requirements, including discretionary and mandatory access control policies [30]. Hence, RBAC is widely supported in commodity operating systems and database systems [15, 17, 25], and is deployed inside many organizations [37].

We call a snapshot of an RBAC system an *RBAC state*. We denote the current state of the running example RBAC system as γ. Administrators can perform administrative actions to take an RBAC system from one RBAC state to another. Usually, the administration configuration is supposed to be static; that is, only the RBAC configuration may be changed. The actions available to administrators we consider are two types:

- `admin` **`assign`** p to r, and
- `admin` **`revoke`** p from r.

Administrators' powers are regulated by the administration configuration. We support variants of the PRA97 component of the ARBAC97 administrative model for RBAC [34]. The administrative model is instantiated by a set of assignment rules and a set of revocation rules. Figure 2 presents the administration configuration of γ. An *assignment rule* is of the form "ar can assign p to r if p assigned to c", which means an administrator in role ar can assign a permission p to r, if p is also assigned to

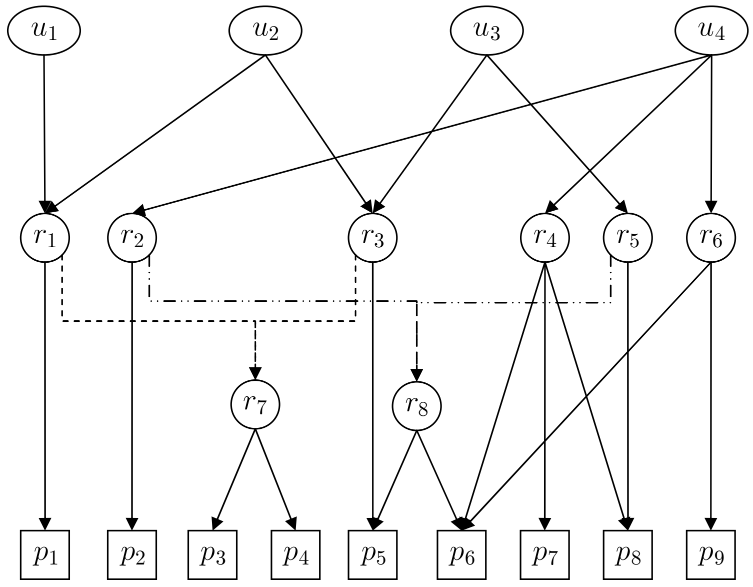


Figure 1: An example RBAC configuration. Users are represented as ellipses, roles as circles, and permissions as rectangles. Arrows between users and roles denote user-role assignments, arrows between roles and permissions denote role-permission assignments, and dashed arrows between roles denote role-role relationships (role hierarchy).

c. The expression c is constructed by roles and the connector $\wedge$. For example, consider the rule "ar_2 can assign p to r_1 if p assigned to $r_2 \wedge r_3$"; then the administrator admin$_2$ can assign a permission p to r_1 if p is assigned to r_2 and r_3.[1] A *revocation rule* is of the form "ar can revoke p from r", expressing that an administrator in role ar can revoke a permission p from r.

Update of RBAC systems is complex and challenging, especially for large-scale RBAC deployments. Existing tools mainly help administrators analyze and manage the RBAC system; they put little emphasis on suggesting to administrators how to configure the system. As shown in Figure 3a, with existing tools, administrators may have to update the system in a manual way. Figure 3b shows a typical process of manual update when one administrator is present. The administrator first determines and specifies, in some language, the update objective and the constraints that the final resulting system should satisfy. Usually, an update objective is initially formulated as high-level objectives (e.g., being able to assign $\{p_5, p_8, p_9\}$ to a user) . Arbitrary update may hinder the security and availability of the RBAC system. For example, revocation of a doctor's permission to write to a patient's medical record as a result of updating is not

[1]Consider, for example, the following situation: an administrator wants to enable an engineer to release the source code of a piece of software; however, the administrator can not do so unless the product manager and the quality manager are authorized to release the source code.

```
assignment rules:
  $ar_1$ can assign $p$ to $r_6$
       if $p$ assigned to $r_1 \wedge r_2$;
  $ar_2$ can assign $p$ to $r_1$
       if $p$ assigned to $r_2 \wedge r_3$;
  $ar_2$ can assign $p$ to $r_1$
       if $p$ assigned to $r_2 \wedge r_4$;

  $ar_0$ can assign $p$ to $r_1$;
              ...
  $ar_0$ can assign $p$ to $r_6$;

revocation rules:
  $ar_1$ can revoke $p$ from $r_4$;
  $ar_1$ can revoke $p$ from $r_6$;
  $ar_2$ can revoke $p$ from $r_1$;
  $ar_2$ can revoke $p$ from $r_2$;
  $ar_2$ can revoke $p$ from $r_3$;
  $ar_3$ can revoke $p$ from $r_5$;
  $ar_3$ can revoke $p$ from $r_6$;

  $ar_0$ can revoke $p$ from $r_1$;
              ...
  $ar_0$ can revoke $p$ from $r_6$;

administrative role assignments:
  admin$_0$ in $ar_0$; admin$_1$ in $ar_1$;
  admin$_2$ in $ar_2$; admin$_3$ in $ar_3$;
```

Figure 2: An example administration configuration.

acceptable.

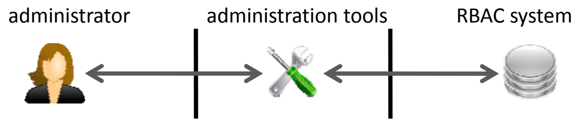
To modify system configurations, an administrator needs to observe the system and the constraints, and devises an update plan, which consists of a sequence of administrative actions. The administrator implements those actions, which take the system to a new state. There is no guarantee that all constraints are met and that this new state is the desired one. Hence, the administrator proceeds to check if these two conditions hold. When either one does not hold, the administrator may need to undo some previous actions and repeat the process. Roughly speaking, this is a trial-and-error approach. For large and complex systems, one can fail to achieve update after several trials; in this case, the question is whether to give up or not. Thus there arises a question: is the update achievable at all? An answer to this question helps the administrator make proper decisions. A positive answer implies that the update can be achieved and that the administrator should persevere in trying, whereas a negative one saves the administrator from continuing with pointless attempts.

On the other hand, suppose that the administrator finally manages to update the system without violating constraints. In this case, how different is the updated system from the original one? The less different it is, the more easier for one to understand and maintain the system, and thus the more preferable the update is. In other words, we may pursue an update that incurs minimal differences.

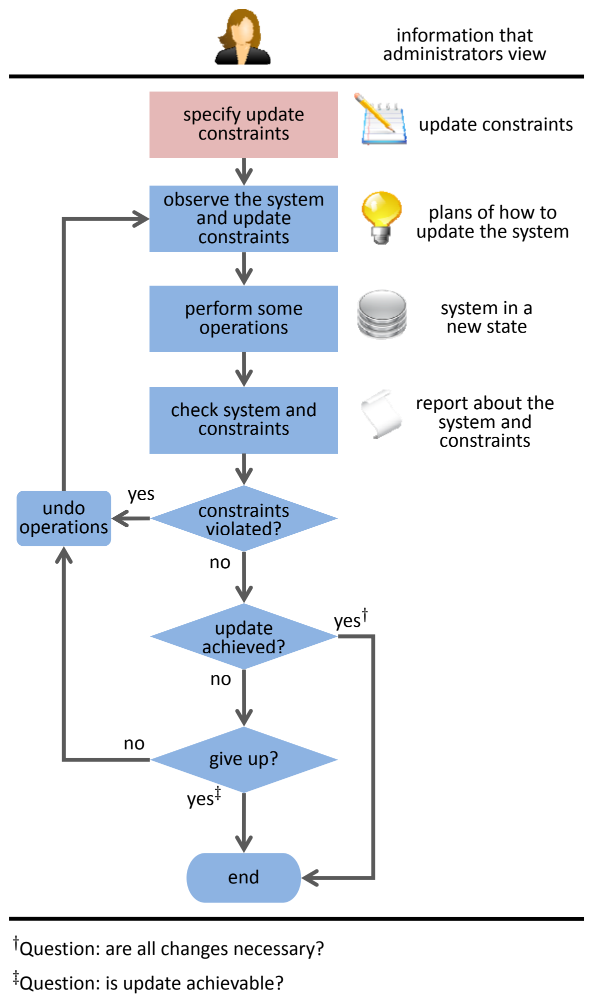
When multiple administrators are involved, the problem become more complicated. The actions an administrator can take might depend on others' actions. That is, administrators have mutual influence on each other in terms of administrative power. Cooperation among administrators is required in this case, which increases the complexity and cost of manual update. In summary, manual administration for update is work-intensive, inefficient and, when the objective is not achievable at all, very frustrating.

Access control update is demanded when security requirements are changed. In addition, RBAC systems may need updating in response to the following developing situations:

Misconfiguration Repair Misconfigurations in access control systems can result in severe consequences [4]. In a health-care situation, for instance, lack of legal authorization could lead to the delay of treatment. Modern access control systems include hundreds of rules, which are managed by different administrators in a distributed manner. The increasing complexity of access control systems gives rise to more likelihood of misconfigurations [2, 3]. As such, correcting misconfigurations is essential to systems' usability and security. Updating is necessary when misconfigurations in RBAC systems are detected.



(a)


information that administrators view
specify update constraints
update constraints
observe the system and update constraints
plans of how to update the system
perform some operations
system in a new state
check system and constraints
report about the system and constraints
undo operations
yes
constraints violated?
no
update achieved?
yes†
no
no
give up?
yes‡
end
†Question: are all changes necessary?
‡Question: is update achievable?

(b) Workflow of manual update.

Figure 3: Illustration of updating without RoleUpdater.

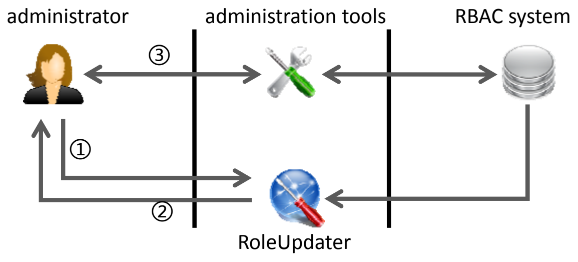
Task Assignments To accomplish a task, a set of permissions should be assigned to a set of users to empower them to perform task operations [13]. For a new task, it is likely that the present RBAC configuration fails to enable exactly the needed user-permission assignments. In this case, administrators may resort to adjusting role configurations.

Property satisfaction An RBAC system should exhibit various properties, including simple availability/safety and containment availability/safety [14, 22, 23, 24]. A simple availability/safety property asks whether a user Alice has a permission, e.g., access to a confidential file. Containment safety properties encode queries such as whether any user who can access printers are members of staff, whereas containment availability properties may ask whether all students have permission to use a library.

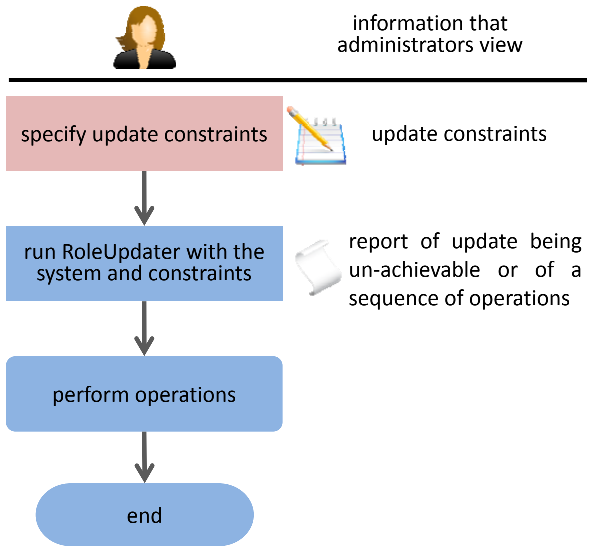
If an RBAC system was not designed with these properties in mind, it is unlikely that all properties would happen to hold. Particularly, for legacy systems, there is no guarantee of automatic establishment of security properties when they are migrated to RBAC management. On the other hand, even if all desired security properties hold currently, requirements are not static. For example, it may be desired that now only managers, instead of employees, have access to an internal document. To assure these properties, one may have to update the RBAC system.

Updating is a key component of maintenance in the RBAC life-cycle [18], and accounts for a great proportion of the total cost of maintenance [29]. RoleUpdater assists administrators with update tasks. As shown in Figure 4a, prior to updating the system, the administrator first interacts with RoleUpdater, and then manipulates the system using suggestions from RoleUpdater. Figure 4b shows the workflow of updating with RoleUpdater. The administrator still needs to specify the update constraints, and invoke RoleUpdater with the request. RoleUpdater checks, in an automatic way, whether the request achievable or not; and if so, a sequence of actions, which take the system to the expected state, is reported. RoleUpdater can also deal with the case where multiple administrators are involved.

RoleUpdater makes novel use of model checking techniques [6]. Figure 5a illustrates the basic idea of model checking. A model checker takes a description of a system and a property as inputs, and examines the system for the property. If the system exhibits the property, the checker reports that the property is true. If the system is found to lack the property, the model checker produces one counter-example. The counter-example, usually a sequence of system state transitions, explains how the system transits to a state where the property fails. Figure 5b illustrates how to use model checking as the basis for update. We check the property that the requested state is never reached; when the property does not hold, one is not only informed of the existence of an update but also a counter-example that corresponds to the update. RoleUpdater transforms update problems into model checking problems, where the failure of the model is synonymous with existence of a solution:



(a)


information that administrators view
specify update constraints
update constraints
run RoleUpdater with the system and constraints
report of update being un-achievable or of a sequence of operations
perform operations
end

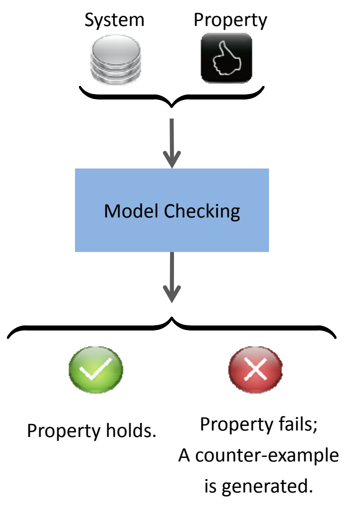
(b) Workflow of update with RoleUpdater.

Figure 4: Illustration of updating with RoleUpdater.

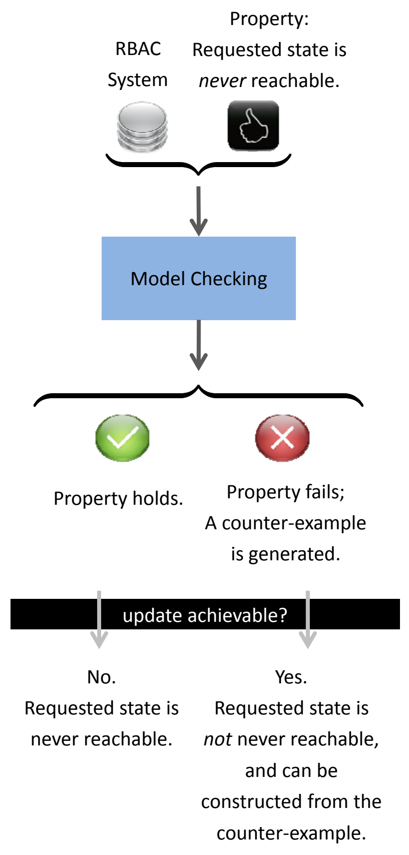
- if the property is determined to be true, the update objective is not achievable;
- otherwise, the model checker returns a counter-example, from which an update is constructed.

RoleUpdater employs NuSMV [5] to perform model checking. NuMSV is a open-source symbolic model checker. For better performance, a collection of reductions and optimization techniques are implemented in RoleUpdater.

The rest of this paper is structured as follows. Related works are given in Section 2. We demonstrate the use of RoleUpdater by showing how it handles a high-level update request specification in Section 3. Section 4 presents the design and implementation of RoleUpdater. We show some experimental results of running RoleUpdater in Section 5, illustrating its effectiveness and efficiency. Section 6 concludes the paper.



(a) The basic illustration of model checking.



(b) Update via model checking.

Figure 5: Illustration of model checking and its usage for updating.

2 Related Work

RBAC administration and analysis Many convenient RBAC administration models (e.g., [8, 21, 34]) are at our disposition. They provide significant advantages in access control management. They define administrative rules, e.g., specifying which administrator can perform what operations. However, high-level update is rarely supported. It is generally difficult and error-prone, because usually the resulting state is expected to meet various constraints.

To help administrators understand RBAC policies, various RBAC policy analysis tools (RPATs) have been invented [4, 14, 23, 38, 39, 44]. RPATs usually answer if an RBAC system satisfies a property. However, little effort has been devoted to answering the question: what if the RBAC system fails to meet the property? When administrators find abnormalities with RPATs, RoleUpdater can assist in correcting them.

Most security analysis problems in literature basically can be stated as: given the current state γ, a query q (e.g., whether accesses to internal documents are only available to employees), and a state-change rule φ, can γ be taken to a state γ' where q evaluates to true? If this is the case, the steps taking γ to γ' may also be reported to administrators so that they can follow them to make γ'. However, as the objectives are different, we believe this kind of reporting could hardly be considered sufficient for the role updating problem. RPATs' objective is to analyze the system. So, their input is just the property to be examined. By contrast, RoleUpdater aims to update the system; the input is the update request. RPATs explore every possible sequence of actions, as long as they are allowed by φ, to test if there is such a γ' where q is true. In this case, administrators do not have any control of the resulting state. By contrast, RoleUpdater seeks a resulting state that complies with administrators' request. In addition, most RPATs focus on user-role assignments. Although it is argued that the role-permission relation might be treated similarly to the user-role relation, the role-permission relation also deserves its own attention [29], especially in terms of role updating.

Various access control properties are proposed and verification schemes are devised to check the satisfiability of properties. In [23], authors propose a tool to answer a set of interesting properties, including simple availability/safety, bounded safety and containment availability/safety. The tool provides a means to guarantee that security requirements are always met as long as trusted users abide by certain behavior patterns [22, 23]. However, an assumption is needed for the usage of security analysis: the properties hold in the current RBAC state [22, 23]. As mentioned above, this is not always the case. Role updating can be used to adjust the current RBAC state so as to exhibit desired properties, while keeping the changes to the customized extent.

```
update
  make $\mathcal{P} = \{p_5, p_8, p_9\}$ available via $\mathcal{T} = \{r_1, r_2, r_3, r_4, r_5, r_6\}$
  with
    administrators $admin_1$, $admin_2$;
    user-permission constraints
      ($u_1$, no-less-than $\{p_1\}$, no-more-than $\{p_1, p_3, p_4\}$),
      ($u_2$, no-less-than $\{p_1, p_3, p_4, p_5\}$, no-more-than $\{p_1, p_3, p_4, p_5\}$),
      ($u_3$, no-less-than $\{p_3, p_4, p_5\}$, no-more-than $\{p_3, p_4, p_5, p_6, p_8\}$),
      ($u_4$, no-less-than $\{p_7, p_8, p_9\}$, no-more-than $\{p_3, p_5, p_6, p_7, p_8, p_9\}$);
    restricted-role constraints
      ($r_4$, no-less-than $\{p_6, p_7\}$, no-more-than $\{p_6, p_7, p_8, p_9\}$),
      ($r_8$, no-less-than $\{p_5, p_6\}$, no-more-than $\{p_5, p_6\}$);
    role-hierarchy = $\{(r_2, r_8), (r_3, r_7)\}$;
    minimal;
```

Figure 6: An example high-level update request specification.

Role engineering Role engineering attracts much research effort [7, 10, 26, 40, 41, 45]. Existing role engineering tools (eRETs) take user-permission assignments as input and output user-role assignments and role-permission assignments. eRETs may take into account some other information such as business meanings, semantics, and users' attributes. Taxonomically, RoleUpdater can be viewed as a role engineering tool. However, role updating works when RBAC states have been defined and possibly deployed, whereas eRETs usually define roles from scratch. The focuses are also different. Role updating aims to answer administrators' question whether an update is achievable with respect to update constraints and how to generate one, if any. By contrast, eRETs put more emphasis on how to define an appropriate set of roles. In the context of a role life cycle, RoleUpdater is for role maintenance, while eRETs help with role design. Thus, one may consider RoleUpdater as a complement to eRETs; RoleUpdater can be used to fine-tune the ideal state generated by eRETs.

RBAC udpate Ni et al. [29] studied the role adjustment problem (RAP) in the context of role-based provisioning via machine-learning algorithms. Though similar, the role updating problem differs from the RAP in several aspects. First, customized constraints on updates are enforced in RoleUpdater, whereas it is unclear if these constraints could be supported in RAP. Second, our role updating is request-driven, whereas RAP is a learning process. RAP and RoleUpdater are both assistant tools for administrators but with different usage and orientation.

Fisler et al. [16] investigated the semantic difference of two XACML policies and the related properties. However, they do not consider how to make a different desired state from the current one. Ray [32] studied the real-time update of access control policies, in the context of database systems. They focused on transaction properties, instead of RBAC policies.

```
admin$_2$ assign $p_8$ to $r_1$;
admin$_2$ assign $p_8$ to $r_2$;
admin$_1$ assign $p_8$ to $r_6$;
admin$_2$ revoke $p_8$ from $r_1$;
admin$_2$ revoke $p_8$ from $r_2$;
admin$_1$ revoke $p_6$ from $r_6$;
admin$_2$ assign $p_5$ to $r_1$;
admin$_1$ assign $p_5$ to $r_6$;
admin$_2$ revoke $p_5$ from $r_1$;
```

Figure 7: The update returned by RoleUpdater when running with the request in Figure 6.

3 High-Level Update Request Specifications

We do not consider the update of user-role assignments, because users' role memberships are determined by their attributes, jobs, titles, etc. When this information is renewed, administrators can accomplish user-role assignments straightforwardly.

Suppose the administrators want to update the RBAC configuration in Figure 1. Suppose further that the administrators specify the update request as in Figure 6. This specification expresses the customized conditions on the potential updated system. In the rest of this section, we illustrate the use of RoleUpdater through this example. Running with this example, RoleUpdater returns the steps, as shown in Figure 7, that the administrators can follow to make the changes; in the updated state, the administrators can assign $\{p_5, p_8, p_9\}$ to users via r_6.

Administrative power Line 4 specifies which administrators are going to update the system. As mentioned before, it is common for administrative rules to regulate administrators' operations; that is, administrators have limited administrative power. A proposed update does not make sense unless the needed changes lie within administrators' capabilities.

RoleUpdater appears more useful when multiple administrators are involved. Observe the five actions by $\texttt{admin}_1$ and $\texttt{admin}_2$: $\texttt{admin}_2$ assigns p_8 to r_1 and r_2, $\texttt{admin}_1$ assigns p_8 to r_6, and $\texttt{admin}_2$ revokes p_8 from r_1 and r_2. These interleaving operations require close cooperation between $\texttt{admin}_1$ and $\texttt{admin}_2$ and careful examination. By contrast, RoleUpdater takes the cooperation among administrators into account automatically.

Suppose that we replace Line 4 with the following.

```
administrators admin3;
```

That is, the administrator $\texttt{admin}_3$, instead of $\texttt{admin}_1$ and $\texttt{admin}_2$, wants to update the system. Then RoleUpdater suggests an alternative: first revoke p_6 from r_5 and then revoke p_6 from r_6. However, $\texttt{admin}_1$ and $\texttt{admin}_2$ are not authorized to perform this alternative. Note that administrators' powers are configured in Figure 2.

Controllable effects Administrators should be able to confine the effects of an update. With RoleUpdater, administrators can specify a certain set of users U and define what changes could happen to users' permissions. For example, Alice at least has access to files under "/foo/bar1" but at most "/foo/bar1" and "/foo/bar2". Line 5 to Line 9 are constraints on users' permissions after update. For example, by Line 6, administrators requires that u_1 have at least permission p_1, but at most p_1, p_3, and p_4 in the potential new state. Note that users still obtain permissions via roles and even that users' role assignments remain the same.

For another example, Line 7 prescribes that u_2's permissions are exactly $\{p_1, p_3, p_4, p_5\}$. Consider the solution in Figure 7; administrators have to revoke p_8 from r_1, for u_2 is assigned to r_1 and cannot have permission p_8, as required by Line 7.

By properly specifying constraints, administrators guarantee the tasks associated with users in U progress smoothly. Suppose that u_2 and u_4 cooperate to finish a task $\texttt{t}$, which requires that u_2 and u_4 are entitled to privileges $\{p_1, p_3, p_4, p_5\}$ and $\{p_7, p_8, p_9\}$, respectively. Then Line 7 and Line 9 guarantee that the updated state, if any, would not disable $\texttt{t}$.

When administrators are specifying U, U often contains those users for whom the administrators are not responsible so that they have to ensure that the potential update does not affect such users, and/or those users whose permissions are designated by the administrators and vary within a range. For users outside the set U, their current role assignments and permissions in γ are neglected by RoleUpdater; that is, updates may change their role-assignments and permission-assignments.

Restricted update The principle of *least privilege* is important in computer security and well supported by RBAC. Users activate only the roles necessary to finish the underlying work, but not all assigned roles. For example, a user Alice may activate the role manager when she wants to evaluate an employee under her department, and activates the employee role for routine works. As a result, upper bounds should be put on roles' permission sets in compliance with the least privilege principle. On the other hand, some roles are designed with expected functions; users should be able to perform a particular job with such a role. If associating with the role a set of permissions less than necessary, administrators may make the role useless. Hence, it would be handy if administrators are able to set the permission sets of certain roles within a range.

Line 10 to Line 12 shows constraints on roles' permissions after update. For each selected role (e.g., r_4), administrators can impose a lower bound (e.g., $\{p_6, p_7\}$) and an upper bound (e.g., $\{p_6, p_7, p_8, p_9\}$) on the role's permissions. RoleUpdater assures that the role is assigned to permissions no less than those in the lower bound and also no more than those in the upper bound.

A requirement is that, the upper bound (or the lower bound) of the range should be a superset (or subset) of the set of all permissions that r is currently assigned in γ. This is reasonable, because the permissions r has currently in γ are enough to make it useful. We also find that, without this requirement, RoleUpdater's efficiency degrades.

Line 12 indicates that r_8's permissions must still be $\{p_5, p_6\}$ after update, because the lower bound equals the upper bound. We call roles like r_8 *invariant roles*. Despite the importance of update, it is likely that administrators demand some roles be invariants in order to, for example, preserve roles' intuitions, business meanings or definitions. In this case, by letting the lower bound of r be its upper bound, administrators request RoleUpdater to find an update which does not change r's permission assignments. In other words, RoleUpdater may change those non-invariant roles' permission assignments in the hope to find an update. In practice, non-invariant roles are usually the ones under administrators' control; otherwise, even though an update is found, administrators would not be able to implement it and thus the update is of little value.

If the administrators impose another restricted-role

```
admin₂ assign p₈ to r₁;
admin₂ assign p₈ to r₂;
admin₁ assign p₈ to r₆;
admin₂ revoke p₈ from r₁;
admin₂ revoke p₈ from r₂;
admin₁ revoke p₆ from r₆;
admin₂ revoke p₃ from r₃;
admin₂ revoke p₄ from r₃;
```

Figure 8: An alternative when the role hierarchy (r_3, r_7) is not required.

constraint besides those in Figure 6.

$$(r_6, \texttt{no-less-than}\ \{p_6, p_9\},\ \texttt{no-more-than}\ \{p_6, p_9\})$$

Then RoleUpdater reports that the requested update does not exist, which is indeed the case.

Role hierarchy Role hierarchy is recognized by the proposed NIST standard for RBAC as one of the fundamental criteria [11]. It further mitigates the burden of security administration and maintenance. Usually, there could be a natural mapping between role hierarchy and organization's structure. It is imprudent to alter a role hierarchy arbitrarily. Administrators can ask RoleUpdater to preserve the whole or part of the original role hierarchy. Line 13 tells that r_2 and r_3 are still senior to r_8 and r_7, respectively, in the updated system.

The requirement that r_3 be senior to r_7 stops RoleUpdater from suggesting another solution, as shown in Figure 8. If following this approach, administrators can assign $\{p_5, p_8, p_9\}$ via r_3 and r_6; however, r_3 is no longer a senior role of r_7.

Minimal update As long as an update is implemented, some changes are made to the system. When two update solutions are available, which one is more preferable? One perspective is to compare the changes they recommend. The fewer changes are needed, the closer the resulting state to the original state. Ideally, we may find an update such that none of its changes is redundant; that is, failure to implement any change thereof gives rise to a disqualified state. We say the update is *minimal*.

Minimal update is valuable in several ways. First of all, minimal update causes few difficulties for administrators to understand the new RBAC state. The administrators are responsible for the maintenance of the RBAC system. It is essential for them to comprehend the system's behavior. We can assume that administrators understand the system well before updating. However, changes to the system configuration have the potential to obfuscate the system. Obviously, a smaller gap between the updated state and the original one usually means a smaller degree to which administrators have to re-examine and re-learn the system.

```
admin₂ assign p₈ to r₁;
admin₂ assign p₈ to r₂;
admin₁ assign p₈ to r₆;
admin₂ revoke p₈ from r₁;

admin₁ revoke p₆ from r₆;
admin₂ assign p₅ to r₁;
admin₁ assign p₅ to r₆;
admin₂ revoke p₅ from r₁;
```

Figure 9: Update in response to the request in Figure 6 but without the minimality requirement.

Secondly, minimal update possibly preserves more previously computed analysis results. It is reasonable to assume that the current RBAC state satisfies necessary properties (otherwise, it should have been adjusted). It is likely that more properties might be preserved with minimal update. Finally, minimal update is also desirable when authorization recycling is deployed in access control implementation [42, 43].[2]

In RoleUpdater, administrators can choose to require each returned update to be minimal in the sense that no change is redundant. However, there is a tradeoff between doing this and incurring extra computing overhead. In Figure 6, Line 14 indicates administrators' willingness to find a minimal update. If turning the minimal requirement off, RoleUpdater would possibly not insist on the revocation of p_8 from r_2, for p_8 being assigned to r_2 does not contradict with the constraints. That is, RoleUpdater returns the update in Figure 9.

4 Design and Implementation

Figure 11 shows the architecture of RoleUpdater. Its interface accepts administrators' input and parses the request. We say a request is *canonical* if (1) all administrative operations are available, (2) users' permissions are required to remain unchanged, and (3) no role hierarchy is required to be preserved. Figure 10 shows an example canonical request, where P_i is the set of permissions that user u_i has prior to updating.

[2] Authorization decision-making is time-consuming and costly. Authorization recycling caches the authorization decisions that are made previously and infer decisions for forthcoming authorization requests. As an important mechanism for access control implementation, authorization recycling makes use of "cache" to enhance performance; there, policy update is a major concern. For details, readers are referred to [9, 42, 43].

```
update
  make $\mathcal{P}$ available via $\mathcal{T}$
  with
    administrators all-administrators
    user-permission constraints
      ($u_1$, no-less-than $P_1$,
                     no-more-than $P_1$),
      ($u_2$, no-less-than $P_2$,
                     no-more-than $P_2$),
      ...;
    restricted-role constraints $\emptyset$;
    role-hierarchy = $\emptyset$;
```

Figure 10: An example canonical request.

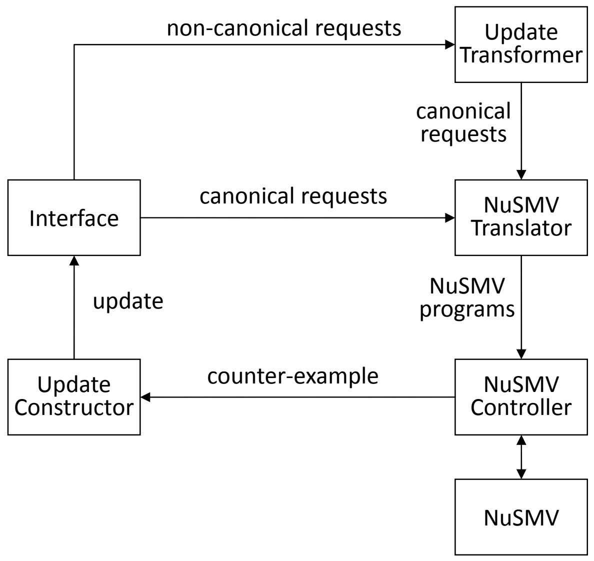


Figure 11: The architecture of RoleUpdater.

If canonical, the request is forwarded to the NuSMV Translator; otherwise, it is first processed by the Update Transformer, where non-canonical requests are transformed into canonical ones. Afterwards, the NuSMV Translator converts requests into NuSMV programs. The NuSMV Controller invokes NuSMV to execute those programs. According to the results returned from NuSMV, the Update Constructor generates an update report, either a sequence of administrative operations which lead to desired RBAC system state or a message that the request is unachievable.

Algorithm 1 presents RoleUpdater's pseudo-code. Line 2 belongs to the Interface module. Line 3 represents the Update Transformer. Line 4 and Line 5 are the main components of the NuSMV Translator module. Normally, the NuSMV Translator would create a set G of NuSMV programs for each canonical request. However, since the execution of the NuMSV programs translated directly from the request easily result in state explosions and memory crashes, some reductions are performed in advance [12]. The set G has the property: an update is found, if and only if, the run of NuSMV with at least one program in G reports a counter-example. As indicated by Line 6, on receiving the NuSMV programs, the NuSMV Controller schedules NuSMV programs in increasing order by the number of variables, because NuSMV's performance highly depends on the number of variables in the input program. The NuSMV Controller proceeds to execute each program with NuSMV; if any execution returns a counter-example, it informs the Update Constructor of the counter-example. The Update Constructor generates the needed update and administrative operations necessary to institute the changes (Line 10 to Line 12). If minimal update is required, further processing (Section 4.3) will be done. In the rest of this section, we give details of each component.

4.1 Handling non-canonical requests

We tried to use the model checking approach directly to evaluate non-canonical update requests. Our experience is that, an extensive number of variables are needed to model complex requests, which often gives rise to state explosions and memory crashes. The reasons are twofold. First, non-canonical requests enable much more potential combinations of role-permission assignments than canonical requests do. Second, some reductions in [12] are not applicable to non-canonical ones. It is not clear how to reduce non-canonical requests effectively.

Consider a non-canonical update request issued against γ in Figure 12. Non-canonical requests are transformed into canonical ones by adding dummy elements (e.g., users, roles, user-role assignments, and role-permission assignments) to γ; these dummy elements simulate those non-canonical conditions on the update. Usually, the obtained RBAC state, against which the canonical request is checked, is more complicated than γ. Fortunately, the construction is polynomial. We trade off the simplicity of RBAC states for the ability to cope with complex updates. By this modeling, we need only to focus on one unified problem: evaluating canonical requests.

4.2 NuSMV program generation

NuMSV is the symbolic model checker that RoleUpdater employs to perform model checking. The NuSMV Translator converts update requests into NuSMV programs. A set of boolean variables are defined to model the RBAC system. To use NuSMV, let ϕ denote the statement that a user could acquire exactly the permissions in $\mathcal{P}$ via roles in $\mathcal{T}$; we ask if $\neg\phi$ is always true in all reach-

Algorithm 1: Algorithm of RoleUpdater.

```
Input: High-level update request $H$, $\gamma$, and NuSMV property $type$: "$AG$" or "$AX\ AG$"
Output: update report
1  begin
       /* Parse($H$,$Q$) parses $H$ and reads information into $Q$; it returns a boolean value
          showing if any error happened.                                                    */
2      if !Parse($H$, $Q$) then show error message;
3      if $Q$ is non-canonical then $Q$ ←TransCanonical($Q$);
       /* perform reductions on $Q$                                                          */
4      Reduce($Q$);
5      $G$ ←TransNuSMV($Q$, $type$);
       /* NuSMV's performance highly depends on the number of variables in the input
          program; so schedule NuSMV programs in increasing order by the number of
          variables.                                                                          */
6      $S\_G$ ←Schedule($G$);
7      foreach $g \in S\_G$ do
8          Invoke $g$ with NuSMV;
9          if a counterexample is returned then
10             construct an update $\gamma'$ from the counter-example;
11             if Minimal update is required then $\gamma'$ ←Minimize($Q$, $\gamma$, $\gamma'$);
               /* compute the needed administrative operations that take the RBAC system
                  from $\gamma$ to $\gamma'$                                                  */
12             $AdminOp$ ←computeAdminOperation($\gamma$, $\gamma'$);
13             show $AdminOp$ and $\gamma'$;
14             return $\gamma'$;
15     show "update unachievable" report;
16     return $\epsilon$;
17 end
```

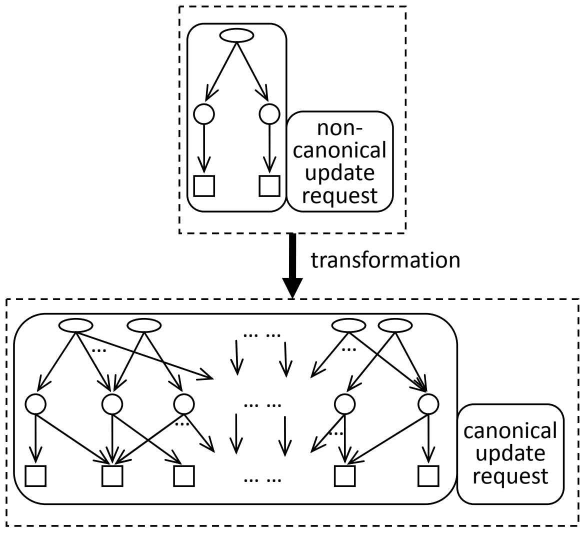


Figure 12: An illustration of the transformation from non-canonical update requests to canonical ones.

able (NuSMV) states;[3] If it evaluates as true, the user can never obtain exactly all permissions in $\mathcal{P}$ via roles in $\mathcal{T}$, indicating that one cannot fulfill the request without violating the update constraints. Otherwise, NuSMV will generate a counter-example, from which RoleUpdater constructs an update.

In the current implementation of RoleUpdater, only boolean variables and `TRANS` declarations are used. An RBAC state is represented by a valuation of boolean variables, whereas `TRANS` declarations capture transitions among RBAC states. Further explorations of NuSMV features and other model checking techniques could improve RoleUpdater's efficiency.

4.3 Minimal Update

Interestingly, the minimal update can be obtained in the same way we seek an update. Once an update is found, denote the RBAC state after update as γ'. As illustrated in Figure 13, if a role-permission assignment appears in exactly one of γ and γ', this assignment is changed

[3] ϕ is defined over the boolean variables. The checked property is $AG\neg\phi$, where A means always and G means globally; $AG\neg\phi$ is a CTL (Computational Tree Logic) formula, which is used to specify properties in NuSMV.

(either removed or added); denote the set of all such changed assignments as CA. Then the minimal update requirement is to determine if all changes in CA are necessary. The basic idea is to ask if the same goal could be achieved with a proper subset of CA. To answer this, we define variables to simulate CA and treat assignments outside CA as constants. This is done by adding dummy elements and imposing new update constraints. A new update request is issued against RoleUpdater; this request is the same as the original one except that new restricted-role constraints are added.

However, the checked property is whether, starting from the next state of γ', all reachable states satisfy $\neg\phi$.[4] If so, then γ' itself is minimal. Otherwise, from the returned counter-example, we could obtain γ''. This γ'' is closer to the minimal update than γ', because only a proper subset of CA is implemented. Note that this is a recursive process; and thus a minimal update could be reached.

Take the request in Figure 6 for example. Figure 14 shows an example calling stack of RoleUpdater. Receiving a request with the minimality requirement, RoleUpdater first removes this requirement and searches for an update. Suppose that RoleUpdater finds the update shown in Figure 9; it proceeds to compute CA, which is $CA = \{(r_6, p_8), (r_2, p_8), (r_6, p_5), (r_6, p_6)\}$. By composing a new update request, RoleUpdater goes on checking if there exists such an update that the resulting changes are a proper subset of CA. This starts a recursive call. Then the same processes are applied. This time, an update shown in Figure 7 is found and CA is computed to be $\{(r_6, p_8), (r_6, p_5), (r_6, p_6)\}$. Again, another round commences. This time, RoleUpdater could not find any update, which implies that the update in Figure 7 is minimal; RoleUpdater returns this update in response to the original request.

5 Experiments

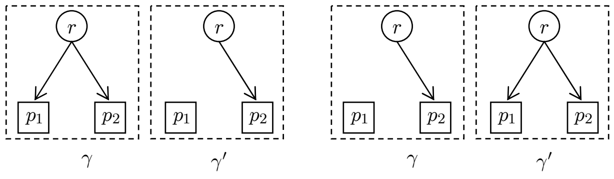
We implemented a prototype of RoleUpdater in Java. Experiments were performed with randomly-generated RBAC systems on a machine with an Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz, and with 2GB of RAM running Microsoft Windows XP Home Edition Service Pack 3.

Data generation

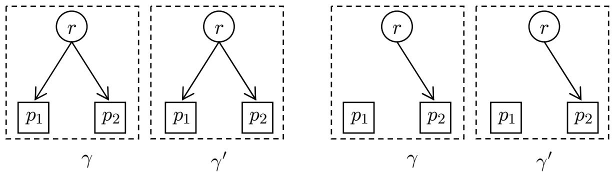
To generate each RBAC system, we adapted algorithms from [41, 45][5]; γ is parameterized by noU (the number of users), noR (the number of roles), noP (the number

[4] In NuSMV, this is expressed by $AXAG\neg\phi$ in CTL.

[5] The latter is accessible via `http://ww2.cs.mu.oz.au/~zhangd/roledata/`.

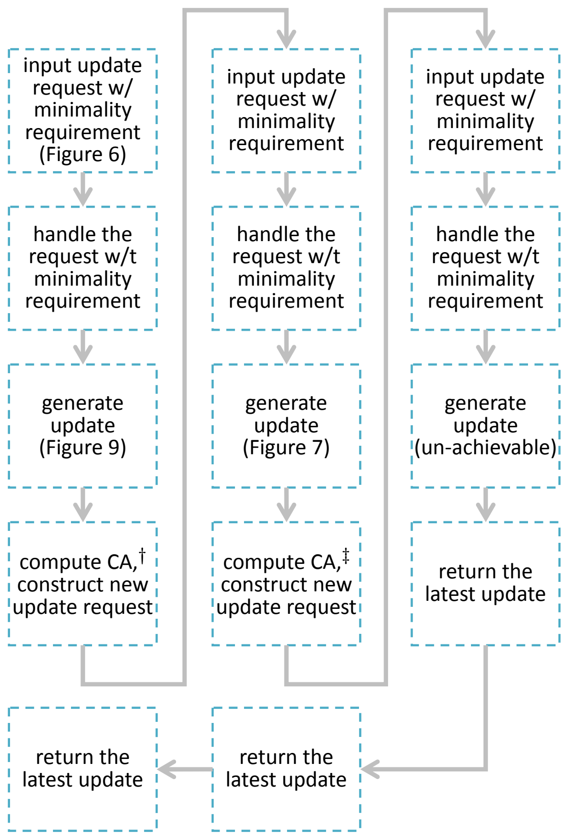


(a) $(r, p_1) \in CA$ (b) $(r, p_1) \in CA$



(c) $(r, p_1) \notin CA$ (d) $(r, p_1) \notin CA$

Figure 13: Examples of assignments in CA.



input update request w/ minimality requirement (Figure 6)
handle the request w/t minimality requirement
generate update (Figure 9)
compute CA,† construct new update request
input update request w/ minimality requirement
handle the request w/t minimality requirement
generate update (Figure 7)
compute CA,‡ construct new update request
input update request w/ minimality requirement
handle the request w/t minimality requirement
generate update (un-achievable)
return the latest update
return the latest update
return the latest update

$^\dagger CA = \{(r_6, p_8), (r_2, p_8), (r_6, p_5), (r_6, p_6)\}$

$^\ddagger CA = \{(r_6, p_8), (r_6, p_5), (r_6, p_6)\}$

Figure 14: The recursive calling procedure.

```
1   update
2     make $\mathcal{P} = input$ available via $\mathcal{T} = \gamma.R$
3     with
4       administrators all-administrators
5       user-permission constraints
6         ($u_1$, no-less-than $P_1$,
7                       no-more-than $P_1$),
8         ($u_2$, no-less-than $P_2$,
9                       no-more-than $P_2$),
10        ...
11      restricted-role constraints $\emptyset$;
12      role-hierarchy = $\gamma.RH$;
```

Figure 15: Experimental update request specification.

of permissions), $noUR$ (the *maximum* number of roles that a user may be assigned to), and $noRP$ (the *maximum* number of permissions that a role may be assigned to). γ's user-role relation (resp. γ's role-permission relation) is generated by associating a number k of roles (resp. permissions) with each user (resp. role), where k is randomly from $[1, noUR]$ (resp. $[1, noRP]$). Without otherwise stated, the parameters used for tests are "$noU = 2000$, $noR = 500$, $noP = 2000$, $noUR = 5$, $noRP = 150$, $noReqps = 200$" and the role hierarchy is empty. One or more parameters are made variable in each group of tests.

Update requests are parameterized by $noReqps$ (the number of requested permissions) and is generated by randomly choosing a number $noReqps$ of permissions from γ's permission set. We let $\mathcal{T}$ be γ's role set. Figure 15 shows the experimental update request, lines of which may be replaced in each group of tests and where P_i is the set of permissions that user u_i has prior to updating.

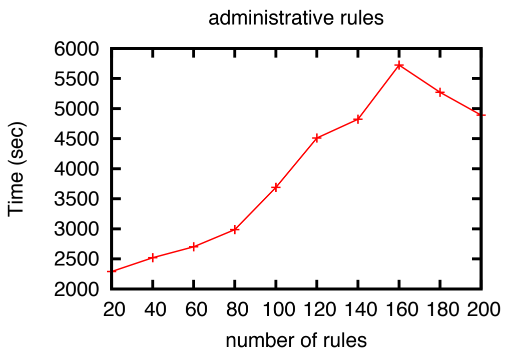
Results

Figure 16 shows the computing time required for each test. Since the data set is randomly created, for each configuration of parameters, we ran the test 5 times. The times in Figure 16 are averaged over the 5 runs.

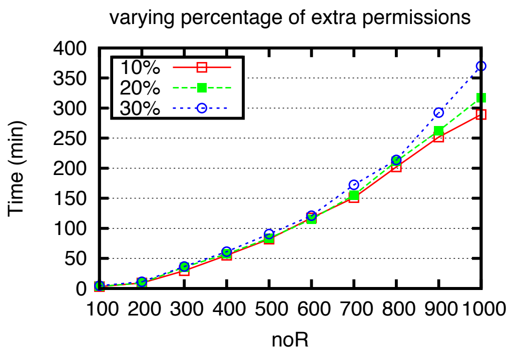
Administrative rules Figure 16a shows performance with respect to varying number of administrative rules ($noRules$). We let an administrator `admin` be a member of role ar and replace Line 4 of Figure 15 with the following.

```
administrators  admin
```

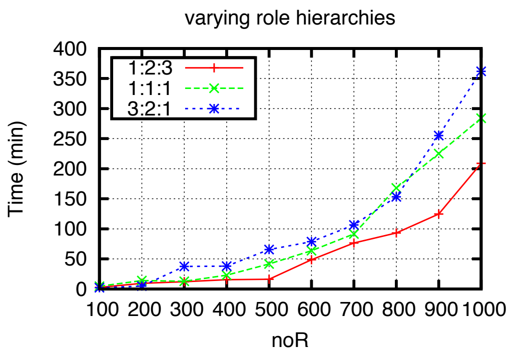
Each assignment rule "ar can assign p with r if p assigned to c" is constructed as follows: (1) denote the number of roles in c as $|c|$ and we let $|c| \in [1, 4]$, and (2) randomly choose roles in c. For revocation rules "ar



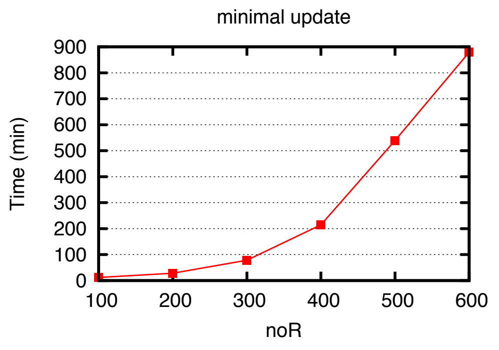
(a)



(b)



(c)



(d)

Figure 16: The computing time for evaluating update requests.

can revoke p from r", r is also randomly chosen. Note that we guarantee that rules have effects on the roles that might be changed. The speed of RoleUpdater is quite good as far as administrative rules are concerned. The reasons are two-fold: (1) The transformation into canonical requests is fast. (2) During the transformation, RoleUpdater only increases noU and noR but not $noUR$; fortunately, RoleUpdater is scalable to noU and noR [12].

Controllable effects To test RoleUpdater's performance with respect to controllable effects, we generated a ratio α of extra permissions. For each user u_i, we define the following constraint and substitute it for the corresponding line

$$(u_i, \texttt{no-less-than}\ P_{l,i},\ \texttt{no-more-than}\ P_{m,i})$$

where $P_{l,i} \subset P_i$ and $|P_{l,i}| = (1 - \alpha) * |P_i|$, and $P_{m,i} \supset P_i$ and $|P_{m,i}| = (1+\alpha)*|P_i|$. Extra permissions were randomly chosen. Recall that P_i is the set of permissions that u_i has prior to updating. Figure 16b shows the results when α takes 10%, 20%, and 30%, respectively. It seems from this experiment that RoleUpdater is not sensitive to α, especially when $noR \leq 800$.

Role hierarchy Figure 16c gives the test results when the RBAC state involves role hierarchies. Role hierarchies were created in the following way. We created three sets of roles R_1, R_2, and R_3 such that $R_i \cap R_j = \emptyset$ for $i, j \in [1, 3]$ and $i \neq j$; we randomly created $\gamma.RH \subset (R_1 \times R_2) \cup (R_2 \times R_3)$ (where $\gamma.RH$ denotes γ's role hierarchy) such that each role may have only a number h of junior roles where $h \in [1, 3]$. This two-level layered role hierarchy is common in practical systems [19, 27, 31]. The x-axis is $|R_1| + |R_2| + |R_3|$. We tested three configurations by varying $|R_1| : |R_2| : |R_3|$. As the RBAC configuration needs to be flattened, $noUR$ is increased by 2 on average. This results in notable overhead. However, the time taken was sensitive to the structures of role hierarchies: almost all runs with $1 : 2 : 3$ were much faster than $3 : 2 : 1$. That is, the less senior roles there were, the faster RoleUpdater dealt with role hierarchies.

Minimal update To evaluate how well RoleUpdater treats minimal update, the minimality requirement is inserted into the specification in Figure 15. Figure 16d reports the computing time when minimal update is pursued. Note that the time was averaged over 5 achievable requests. When $noR = 600$, the computing time could be almost 18 times greater than the case without the minimal update requirement. This is because RoleUpdater has to compute a number of intermediate updates, with the number depending on $|CA|$. It would be interesting and useful to investigate how to reduce the number of intermediate steps.

In real-world large-scale RBAC systems, we expect that only a small portion of users have a number $noUR$ of roles and that the number of roles that are under specified administrators' control will be small. Hence, we conjecture RoleUpdater will be able to handle update requests in these RBAC systems, especially with the advances in model checking.

6 Conclusion

To update an access control system, we have presented a tool RoleUpdater, which accepts and answers high-level update requests. Experiments confirm the effectiveness and efficiency of RoleUpdater. We have reported the theoretical results of RoleUpdater in [12], including the computational complexity, the formal transformation into model checking problem, and the reductions. However, the full-fledged RoleUpdater is first reported here. RoleUpdater is still experimental and we regret that it is not yet available to the public.

There are several avenues for future work. RoleUpdater becomes awkward when dealing with administrative rules with negations, e.g., "ar can assign p if p assigned to r_1 but not r_2". The problem with more sophisticated administrative models, where negative conditions are allowed, deserves further investigation. In addition, *separation-of-duty* (SoD) policies are important in RBAC systems; however, enforcing SoD policies is difficult by itself [20]. The interaction between updating and SoD policies poses new challenges. On the other hand, if a series of update requests are issued, the final updated RBAC state may depend on the order of the requests. These composite requests may take place in distributed RBAC systems. We plan to investigate properties of composite update requests and extend RoleUpdater to address this problem.

7 Acknowledgment

We would like to thank Dr. Alva L. Couch for shepherding this paper and the anonymous reviewers for their helpful comments. This work is supported by National Natural Science Foundation of China under Grant 60873225, 60773191, 70771043, National High Technology Research and Development Program of China under Grant 2007AA01Z403, and Natural Science Foundation of Hubei Province under Grant 2009CDB298. This research is supported in part by an Australian Research Council Discovery Grant (DP0988396). This publication was made possible by a grant from the Qatar National Research Fund under its NPRP Grant No. 09-079-1-013.

Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the Qatar National Research Fund.

Author Biographies

Jinwei Hu is a PhD student in School of Computer Science and Technology at Huazhong University of Science and Technology, when submitting this paper. His current interests are the specification and analysis of access control policies.

Yan Zhang is a professor of School of Computing and Mathematics at University of Western Sydney. His research interests are in the areas of knowledge representation, logic, and model checking.

Ruixuan Li is an associate professor of School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests are in the areas of distributed computing and distributed system security.

References

[1] AHMED, T., AND TRIPATHI, A. R. Static verification of security requirements in role based cscw systems. In *SACMAT'03*, pp. 196–203.

[2] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP* (2009), pp. 123–132.

[3] ALIMI, R., WANG, Y., AND YANG, Y. R. Shadow configuration as a network management primitive. In *SIGCOMM* (2008), pp. 111–122.

[4] BAUER, L., GARRISS, S., AND REITER, M. K. Detecting and resolving policy misconfigurations in access-control systems. In *SACMAT'08*, pp. 185–194.

[5] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)* (2002), LNCS, pp. 359–364.

[6] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. MIT Press, 1999.

[7] COLANTONIO, A., PIETRO, R. D., OCELLO, A., AND VERDE, N. V. A formal framework to elicit roles with business meaning in rbac systems. In *SACMAT'09*, pp. 85–94.

[8] CRAMPTON, J. Understanding and developing role-based administrative models. In *CCS* (Alexandria, VA, USA, Nov. 2005), pp. 158 – 167. CCS'05.

[9] CRAMPTON, J., LEUNG, W., AND BEZNOSOV, K. The secondary and approximate authorization model and its application to bell-lapadula policies. In *ACM Symposium on Access Control Models and Technologies* (2006), pp. 111–120.

[10] ENE, A., HORNE, W. G., MILOSAVLJEVIC, N., RAO, P., SCHREIBER, R., AND TARJAN, R. E. Fast exact and heuristic methods for role minimization problems. In *SACMAT'08*, pp. 1–10.

[11] FERRAIOLO, D. F., SANDHU, R. S., GAVRILA, S. I., KUHN, D. R., AND CHANDRAMOULI, R. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (2001), 224–274.

[12] HU, J., ZHANG, Y., LI, R., AND LU, Z. Role updating for assignments. In *Proceedings of 15th ACM symposium on access control models and technologies (SACMAT 2010)* (Pittsburgh, USA, June 9-11 2010), pp. 89–98.

[13] IRWIN, K., YU, T., AND WINSBOROUGH, W. H. Enforcing security properties in task-based systems. In *SACMAT'08*, pp. 41–50.

[14] JHA, S., LI, N., TRIPUNITARA, M., WANG, Q., AND WINSBOROUGH, W. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable Secur. Comput. 5*, 4 (2008), 242–255.

[15] KARJOTH, G. The authorization service of tivoli policy director. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference* (Washington, DC, USA, 2001), IEEE Computer Society, p. 319.

[16] KATHI FISLER, SHRIRAM KRISHNAMURTHI, L. M., AND TSCHANTZ, M. Verification and change impact analysis of access-control policies. In *ICSE* (May 2005).

[17] KERN, A. Advanced features for enterprise-wide role-based access control. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference* (Washington, DC, USA, 2002), IEEE Computer Society, p. 333.

[18] KERN, A., KUHLMANN, M., SCHAAD, A., AND MOFFETT, J. D. Observations on the role life-cycle in the context of enterprise security management. In *SACMAT* (2002), pp. 43–51.

[19] KERN, A., SCHAAD, A., AND MOFFETT, J. D. An administration concept for the enterprise role-based access control model. In *SACMAT* (2003), pp. 3–11.

[20] LI, N., BIZRI, Z., AND TRIPUNITARA, M. V. On mutually-exclusive roles and separation of duty. In *CCS* (2004), pp. 42–51.

[21] LI, N., AND MAO, Z. Administration in role-based access control. In *ASIACCS* (2007), pp. 127–138.

[22] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Beyond proof-of-compliance: security analysis in trust management. *J. ACM 52*, 3 (2005), 474–514.

[23] LI, N., AND TRIPUNITARA, M. V. Security analysis in role-based access control. In *SACMAT* (2004), pp. 126–135.

[24] Li, N., Tripunitara, M. V., and Wang, Q. Resiliency policies in access control. In *CCS* (2006), pp. 113–123.

[25] McPherson, D. *Role-based access control for multi-tier applications using authorization manager: http://technet.microsoft.com/en-us/library/cc780256(WS.10).aspx*.

[26] Molloy, I., Chen, H., Li, T., Wang, Q., Li, N., Bertino, E., Calo, S. B., and Lobo, J. Mining roles with semantic meanings. In *SACMAT* (2008), pp. 21–30.

[27] Molloy, I., Li, N., Li, T., Mao, Z., Wang, Q., and Lobo, J. Evaluating role mining algorithms. In *SACMAT* (2009), pp. 95–104.

[28] Mondal, S., Sural, S., and Atluri, V. Towards formal security analysis of gtrbac using timed automata. In *SACMAT'09*, pp. 33–42.

[29] Ni, Q., Lobo, J., Calo, S. B., Rohatgi, P., and Bertino, E. Automating role-based provisioning by learning from examples. In *SACMAT* (2009), pp. 75–84.

[30] Osborn, S. L., Sandhu, R. S., and Munawer, Q. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur. 3*, 2 (2000), 85–106.

[31] Park, J. S., Costello, K. P., Neven, T. M., and Diosomito, J. A. A composite rbac approach for large, complex organizations. In *SACMAT* (2004), pp. 163–172.

[32] Ray, I. Applying semantic knowledge to real-time update of access control policies. *IEEE Trans. Knowl. Data Eng. 17*, 6 (2005), 844–858.

[33] Reith, M., Niu, J., and Winsborough, W. H. Toward practical analysis for trust management policy. In *ASIACCS '09*, ACM, pp. 310–321.

[34] Sandhu, R. S., Bhamidipati, V., and Munawer, Q. The ARBAC97 model for role-based administration of roles. *TISSEC 2*, 1 (1999), 105–135.

[35] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. Role-based access control models. *IEEE Computer 29*, 2 (February 1996), 38–47.

[36] Schaad, A., Lotz, V., and Sohr, K. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT'06*, pp. 139–149.

[37] Schaad, A., Moffett, J. D., and Jacob, J. The role-based access control system of a european bank: a case study and discussion. In *SACMAT* (2001), pp. 3–9.

[38] Sohr, K., Drouineaud, M., Ahn, G.-J., and Gogolla, M. Analyzing and managing role-based access control policies. *Knowledge and Data Engineering, IEEE Transactions on 20*, 7 (July 2008), 924–939.

[39] Stoller, S. D., Yang, P., Ramakrishnan, C., and Gofman, M. I. Efficient policy analysis for administrative role based access control. In *CCS'07*.

[40] Vaidya, J., Atluri, V., and Guo, Q. The role mining problem: Finding a minimal descriptive set of roles. In *SACMAT* (2007), pp. 175–184.

[41] Vaidya, J., Atluri, V., and Warner, J. Roleminer: mining roles using subset enumeration. In *CCS* (2006), pp. 144–153.

[42] Wei, Q., Crampton, J., Beznosov, K., and Ripeanu, M. Authorization recycling in rbac systems. In *SACMAT* (2008).

[43] Wei, Q., Ripeanu, M., and Beznosov, K. Cooperative secondary authorization recycling. In *HPDC* (2007).

[44] Xu, W., Shehab, M., and Ahn, G.-J. Visualization based policy analysis: case study in selinux. In *SACMAT'08*, pp. 165–174.

[45] Zhang, D., Ramamohanarao, K., Ebringer, T., and Yann, T. Permission set mining: Discovering practical and useful roles. In *ACSAC* (2008), pp. 247–256.

First Step Towards Automatic Correction of Firewall Policy Faults

Fei Chen Alex X. Liu
Dept. of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824-1266, U.S.A.
Email: {feichen, alexliu}@cse.msu.edu

JeeHyun Hwang Tao Xie
Dept. of Computer Science
North Carolina State University
Raleigh, North Carolina 27695, U.S.A.
Email: {jhwang4, txie}@ncsu.edu

Abstract

Firewalls are critical components of network security and have been widely deployed for protecting private networks. A firewall determines whether to accept or discard a packet that passes through it based on its policy. However, most real-life firewalls have been plagued with policy faults, which either allow malicious traffic or block legitimate traffic. Due to the complexity of firewall policies, manually locating the faults of a firewall policy and further correcting them are difficult. Automatically correcting the faults of a firewall policy is an important and challenging problem. In this paper, we make three major contributions. First, we propose the first comprehensive fault model for firewall policies including five types of faults. For each type of fault, we present an automatic correction technique. Second, we propose the first systematic approach that employs these five techniques to automatically correct all or part of the misclassified packets of a faulty firewall policy. Third, we conducted extensive experiments to evaluate the effectiveness of our approach. Experimental results show that our approach is effective to correct a faulty firewall policy with three of these types of faults.

1 Introduction

1.1 Motivation

Firewalls serve as critical components for securing the private networks of business, institutions, and home networks. A firewall is often placed at the entrance between a private network and the outside Internet so that it can check all incoming and outgoing packets and decide whether to accept or discard a packet based on its policy. A firewall policy is usually specified as a sequence of rules that follow the first-match semantics where the decision for a packet is the decision of the first rule that the packet matches. However, most real-life firewall policies are poorly configured and contain faults (*i.e.*, misconfigurations) [21]. A policy fault either creates security holes that allow malicious traffic to sneak into a private network or blocks legitimate traffic and disrupts normal business processes. In other words, a faulty firewall policy evaluates some packets to unexpected decisions. We call such packets *misclassified packets* of a faulty firewall policy. Therefore, it is important to develop an approach that can assist firewall administrators to automatically correct firewall faults.

1.2 Technical Challenges

There are three key challenges for automatic correction of firewall policy faults. First, it is difficult to determine the number of policy faults and the type of each fault in a faulty firewall. The reason is that a set of misclassified packets can be caused by different types of faults and different number of faults. Second, it is difficult to correct a firewall fault. A firewall policy may consist of a large number of rules (*e.g.*, thousands of rules) and each rule has a predicate over multi-dimensional fields. Locating a fault in a large number of rules and further correcting it by checking the field of each dimension are two difficult tasks. Third, it is difficult to correct a fault without introducing other faults Due to the first-match semantics of firewall policies, correcting a fault in a firewall rule affects the functionality of all the subsequent rules, and hence may introduce other faults into the firewall policy.

1.3 Limitations of Prior Art

To the best of our knowledge, no approach has been proposed for automatic correction of firewall policy faults. The closest work to us is the technique of firewall fault localization proposed by Marmorstein *et al.* [17]. Their technique first finds failed tests that violate some security requirements and further uses the failed tests to locate two or three faulty rules in a firewall policy. However, many types of faults cannot be located by their technique, *e.g.*, wrong order of firewall rules, which is a common type of fault in firewall policies [21]. Furthermore, even if a faulty rule is located, it may not be corrected by just changing the faulty rule. For example, if a firewall policy misses one rule, we cannot single out a faulty rule in the policy to correct.

Another piece of related work is fault localization/fixing, which has been studied in software engineering for years (*e.g.*, [1, 10, 19, 23]). The state-of-the-art techniques in that field focus on locating/fixing a single fault in a program. While the proposed approach in this paper can effectively correct multiple faults in a faulty firewall policy for three types of faults. Our work serves as a good starting point towards policy-fault fixing.

1.4 Our Approach

To correct a faulty firewall policy, essentially we need to correct all *misclassified packets* of the policy such that all these packets will be evaluated to expected decisions. However, it is not practical to manually find every misclassified packet and then correct it due to the large number of misclassified packets of the faulty policy.

The idea of our approach is that we first find some samples of all the misclassified packets and then use these samples to correct all or part of the misclassified packets of the faulty policy. We propose the first comprehensive fault model for firewall policies. The proposed fault model includes five types of faults, *wrong order*, *missing rules*, *wrong decisions*, *wrong predicates*, and *wrong extra rules*. For each type of fault, we propose a correction technique based on the passed and failed tests of a firewall policy. Passed tests are packets that are evaluated to expected decisions. Failed tests are packets that are evaluated to unexpected decisions. Note that the failed tests are samples of all misclassified packets.

To generate passed and failed tests, we first employ automated packet generation techniques [8] to generate test packets for a faulty firewall policy. The generated packets can achieve high structural coverage, *i.e.*, covering all or most rules [8]. Second, administrators classify these packets into passed and failed tests by checking whether their evaluated decisions are correct. Identifying passed/failed tests can be automated in some situations, *e.g.*, when policy properties are written, or multiple implementations of the policy are available. Even if this operation cannot be done automatically, manual inspection of passed/failed tests is also common practice for ensuring network security in industry. For example, applying some existing vulnerability testing tools, such as Nessus [18] and Satan [20], does need manual inspection. In this paper, our goal is to automatically correct policies after we have passed/failed packets. Identifying passed/failed tests is out of scope of this paper.

Given passed and failed tests, correcting a faulty firewall policy is still difficult because it is hard to identify the number of faults and the type and the location of each fault in the firewall policy. To address this problem, we propose a greedy algorithm. In each step of the greedy algorithm, we try every correction technique and choose one technique that can maximize the number of passed tests (or minimize the number of failed tests). We then repeat this step until there are no failed tests.

Our proposed approach cannot guarantee to correct all faults in a firewall policy because it is practically impossible unless the formal representation of the policy is available. However, in practice, most administrators do not have such formal representations of their firewall policies. To correct a faulty firewall policy without its formal representation, administrators need to examine the decisions of all 2^{104} packets[1] and manually correct each of misclassified packets; doing so is practically impossible. This paper represents the first step towards automatic correction of firewall policy faults. We hope to attract more attention from the research community on this important and challenging problem.

1.5 Key Contributions

Our major contributions can be summarized as below:

1. We propose the first comprehensive fault model for firewall polices, including five types of faults, wrong order, missing rules, wrong decisions, wrong predicates, and wrong extra rules.

2. We propose the first systematic approach that can automatically correct all or part of the misclassified packets of a faulty firewall policy.

3. We conduct extensive experiments on real-life firewall policies to evaluate the effectiveness of our approach.

1.6 Summary of Experimental Results

We generated a large number of faulty firewall policies from 40 real-life firewalls, and then applied our approach over each faulty policy and produced the fixed policy. Faulty policies with k faults ($1 \leq k \leq 5$) were tested. These faults in a faulty policy were of the same type. The experimental results show that for three types of faults, wrong order, wrong decisions, and wrong extra rules, our approach can effectively correct misclassified packets. When $k \leq 4$, our approach can correct all misclassified packets for over 53.2% faulty policies. This result is certainly encouraging and we hope that this paper will attract more attention from the research community to this

[1]A packet typically includes five fields, source IP (32 bits), destination IP (32 bits), source port (16 bits), destination port (16 bits), and protocol type (8 bits). Thus, the number of possible packets is $2^{32+32+16+16+8} = 2^{104}$.

important problem. For two other types of faults, missing rules and wrong predicates, our approach does not achieve satisfactory results, deserving further study.

2 Related Work

2.1 Firewall Policy Fault Localization

Fault localization for firewall policies has drawn attention recently [9, 17]. Marmorstein *et al.* proposed a technique to find failed tests that violate the security requirement of a firewall policy and further use the failed tests to locate two or three faulty rules in a firewall policy [17]. However, they did not provide a systematic methodology to identify faulty rules according to different types of firewall faults, *e.g.*, wrong order of firewall rules. Furthermore, they applied their approach only to a simple firewall policy (with 5 rules), which cannot strongly demonstrate the effectiveness of their approach.

Our previous work proposed a technique to locate a fault in a firewall policy [9]. The approach first analyzes a faulty firewall policy and its failed tests and then finds the potential faulty rules based on structural coverage metrics[2]. However, this work has three limitations: (1) it considers only two types of faults, which are wrong decisions and wrong predicates, while a firewall policy may contain other types of faults; (2) it considers only a firewall policy with a single fault, while a firewall policy may contain multiple faults; (3) it does not propose a technique to correct the faults in a firewall policy.

2.2 Firewall Policy Analysis and Testing

Firewall policy analysis tools have been proposed in prior work (*e.g.*, [2, 3, 7, 12, 22]). Tools for detecting potential firewall policy faults by conflict detection were proposed in [3, 7]. Similar to conflict detection, some other tools were proposed for detecting anomalies in a firewall policy [2, 22]. Detecting conflicts or anomalies is helpful for finding faults in a firewall policy. However, the number of conflicts or anomalies could be too large to be manually inspected. Therefore, correcting a faulty policy is difficult by using these firewall policy analysis tools. Change impact analysis of firewall policies has also been studied [12]. Such tools are helpful to analyze the impact after changing a firewall policy, but no algorithm has been presented for correcting a faulty firewall policy.

Firewall policy testing tools have also been explored in prior work (*e.g.*, [4, 11, 14, 16]). Such tools focus on injecting packets as tests into a firewall to detect faults in the firewall policy. If the evaluated decision of a packet is not as expected, faults in the firewall policy are exposed. However, because a firewall policy may have a large number of rules and the rules often conflict, it is difficult to manually locate faults and correct them based on the passed and failed tests.

2.3 Software Fault Localization and Fixing

Fault localization and fixing have been studied for years in the software engineering and programming language communities (*e.g.*, [1, 10, 19, 23]). Such research focuses on locating and fixing a fault in a software program. Four main techniques have been proposed for locating/fixing faults in software programs: dynamic program slicing [1], delta debugging [23], nearest neighbor [19], and statistical techniques [10]. These techniques typically analyze likely fault locations based on dynamic information collected from running the faulty program. Firewall polices and general programs are fundamentally different in terms of structure, semantics, and functionality, *etc*. Therefore, fault localization and fixing techniques of software programs are not suitable for locating faults in firewall policies.

3 Background

3.1 Firewall Policies

A firewall policy is a sequence of *rules* $\langle r_1, \cdots, r_n \rangle$ and each *rule* is composed of a *predicate* over d $fields$, $F_1, \cdots, F_d$ and a *decision* for the packets that match the predicate. Figure 1 shows a firewall policy, whose format follows Cisco Access Control Lists [5].

A *field* F_i is a variable of finite length (*i.e.*, of a finite number of bits). The domain of field F_i of w bits, denoted as $D(F_i)$, is $[0, 2^w-1]$. Firewalls usually check five fields, source IP (32 bits), destination IP (32 bits), source port (16 bits), destination port (16 bits), and protocol type (8 bits). For example, the domain of the source IP is $[0, 2^{32}-1]$.

A *packet* p over the d fields $F_1, \cdots, F_d$ is a d-tuple $(x_1, \cdots, x_d)$ where each x_i $(1 \le i \le d)$ is an element of $D(F_i)$. An example packet over these five fields is (1.2.3.5, 192.168.1.1, 78, 25, TCP).

A *predicate* defines a set of packets over the fields $F_1, \cdots, F_d$, and is specified as $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d$. Each S_i is a subset of $D(F_i)$ and is specified as either a prefix or a range. A *prefix* $\{0,1\}^k\{*\}^{w-k}$ (with k leading 0s or 1s) denotes the range $[\{0,1\}^k\{0\}^{w-k}, \{0,1\}^k\{1\}^{w-k}]$. For example, prefix 01** denotes the range $[0100, 0111]$.

A *decision* is an action for the packets that match the predicate of the rule. For firewalls, the typical decisions include *accept* and *discard*.

A packet $(x_1, \cdots, x_d)$ *matches* a rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ if and only if the condition

[2]Firewall policy coverage is measured based on which entities (e.g., rules or fields) are involved (called "covered") during packet evaluation.

$x_1 \in S_1 \wedge \cdots \wedge x_d \in S_d$ holds. For example, the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP) matches the rule r_1 in Figure 1.

A rule $F_1 \in S_1 \wedge \cdots \wedge F_d \in S_d \rightarrow \langle decision \rangle$ is called a *singleton rule* if and only if each S_i has only one element.

Rule	Src. IP	Dest. IP	Src. Port	Dest. Port	Prot.	Dec.
r_1	1.2.3.*	192.168.1.1	*	25	TCP	accept
r_2	*	*	*	*	*	discard

Figure 1: An example firewall

A firewall policy $\langle r_1, \cdots, r_n \rangle$ is *complete* if and only if for any packet p, there is at least one rule that p matches. To ensure that a firewall policy is complete, the predicate of the last rule is usually specified as $F_1 \in D(F_1) \wedge \cdots \wedge F_d \in D(F_d)$, *i.e.*, the last rule r_2 in Figure 1.

Two rules in a firewall policy may *overlap*; that is, there exists at least one packet that matches both rules. Two rules may *conflict*; that is, the two rules not only overlap but also have different decisions. For example, in Figure 1, two rules r_1, r_2 overlap and conflict because the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP) matches r_1 and r_2, and the decisions of r_1 and r_2 are different.

Firewalls typically resolve conflicts by employing the *first-match semantics* where the decision for a packet p is the decision of the first (i.e., highest priority) rule that p matches in the firewall policy. Thus, for the packet (1.2.3.5, 192.168.1.1, 78, 25, TCP), the decision of the firewall policy in Figure 1 is accept.

3.2 Packet Generation

To check the correctness or detect faults in a firewall policy, administrators need to generate test packets to evaluate that each entity (*e.g.*, each rule) is correct. In our previous work [8], we developed automated packet generation techniques to achieve high structural coverage. One cost-effective technique is packet generation based on local constraint solving. In this paper, we use this technique to generate packets for firewall policies. This technique statically analyzes rules to generate test packets. Given a policy, the packet generator analyzes the predicate in an individual rule and generates packets to evaluate the constraints (i.e., rule fields) to be true or false. The generator first constructs constraints to evaluate each field in a rule to be either false or true, and then it generates a packet based on the concrete values derived by constraint solving. For example, given rule r_1 in Figure 1, the generator analyzes r_1 and generates a packet (e.g., packet (1.2.3.5, 192.168.1.1, 23447, 25, TCP)) to cover r_1; this packet evaluates each of r_1's fields to be true during evaluation. Then, the generator analyzes r_2 and generates a packet (e.g., packet (2.2.3.5, 192.168.1.1, 23447, 26, UDP)) to cover r_2; this packet evaluates each of r_2's fields to be true during evaluation. When firewall policies do not include many conflicts, this technique can effectively generate packets to achieve high structural coverage.

4 A Fault Model of Firewall Polices

A fault model of firewall policies is an explicit hypothesis about potential faults in firewall policies. Our proposed fault model includes five types of faults.

1. *Wrong order*. This type of fault indicates that the order of rules is wrong. Recall that the rules in a firewall policy follow the first-match semantics due to conflicts between rules. Misordering firewall rules can misconfigure a firewall policy. Wrong order of rules is a common fault caused by adding a new rule at the beginning of a firewall policy without carefully considering the order between the new rule and the original rules. For example, if we misorder r_1 and r_2 in Figure 1, all packets will be discarded.

2. *Missing rules*. This type of fault indicates that administrators need to add new rules to the original policy. Usually, administrators add a new rule regarding a new security concern. However, sometimes they may forget to add the rule to the original firewall policy.

3. *Wrong predicates*. This type of fault indicates that predicates of some rules are wrong. When configuring a firewall policy, administrators define the predicates of rules based on security requirements. However, some special cases may be overlooked.

4. *Wrong decisions*. This type of fault indicates that the decisions of some rules are wrong.

5. *Wrong extra rules*. This type of fault indicates that administrators need to delete some rules from the original policy. When administrators make some changes to a firewall policy, they may add a new rule but sometimes forget to delete old rules that filter a similar set of packets as the new rule does.

In this paper, we consider faults in a firewall policy that can be represented as a set of misclassified packets. Under this assumption, given a set of misclassified packets, we can always find one or multiple faults in our fault model that can generate the same set of misclassified packets. One simple way to find such faults is that for each misclassified packet, we consider that the faulty policy misses a singleton rule for this misclassified packet. Therefore, we can always find multiple *missing rules* faults that can generate the same set of misclassified packets.

The correction techniques for these five types of faults are called *order fixing*, *rule addition*, *predicate fixing*, *decision fixing*, and *rule deletion*, respectively. Each operation in these five techniques is called a *modification*.

5 Automatic Correction of Firewall Policy Faults

Normally, a faulty firewall policy is detected when administrators find that the policy allows some malicious packets or blocks some legitimate packets. Because the number of these observed malicious packets or legitimate packets is typically small, these packets cannot provide enough information about the faults in the firewall policy, and hence correcting the policy with these packets is difficult. Therefore, after finding a faulty firewall policy, we first employ the automated packet generation techniques [9], which can achieve high structural coverage, to generate test packets for the faulty policy. Second, administrators identify passed/failed tests automatically or manually. According to security requirements for the firewall policy, if the decision of a packet is correct, administrators classify it as a passed test; otherwise, administrators classify it as a failed test. In some situations, *e.g.*, when policy properties are written, or multiple implementations of the policy are available, this operation can be automated. Manual inspection is also a common practice for ensuring network security in industry. For example, applying some existing vulnerability testing tools, such as Nessus [18] and Satan [20], does need manual inspection. Our goal is to automatically correct policies after we have passed/failed packets. Identifying passed/failed tests is out of the scope of this paper.

Figure 2 shows a faulty firewall policy and its passed and failed tests. This policy includes 5 rules over two fields F_1 and F_2, where the domain of each field is [1,10]. The rule r_1 means that accept packets whose value of the first field is in the range [1, 5] and whose value of the second field is in the range [1, 10]. We use a as a shorthand for "accept" and d as a shorthand for "discard". For the passed and failed tests, we use a and d to denote expected decisions. We assign each test a distinct ID p_i ($1 \leq i \leq 8$).

Given passed and failed tests, it is difficult to automatically correct a faulty firewall policy for three reasons. First, it is difficult to locate the faults because a firewall policy may consist of a large number of rules, and the rules often conflict. Second, before correcting a fault, we need to first determine the type of the fault and then use the corresponding correction technique to fix this fault. However, it is difficult to determine the type of a fault because the same misbehavior of a firewall policy, *i.e.*, the same set of misclassified packets, can be caused by different types of faults. Third, it is difficult to correct a

$r_1 : F_1 \in [1,\ 5] \wedge F_2 \in [1, 10] \rightarrow a$
$r_2 : F_1 \in [1,\ 6] \wedge F_2 \in [3, 10] \rightarrow a$
$r_3 : F_1 \in [6, 10] \wedge F_2 \in [1,\ 3] \rightarrow d$
$r_4 : F_1 \in [7, 10] \wedge F_2 \in [4,\ 8] \rightarrow a$
$r_5 : F_1 \in [1, 10] \wedge F_2 \in [1, 10] \rightarrow d$

(a) An example faulty firewall policy

$p_1 : (3,\ 2) \rightarrow a$	
$p_2 : (5,\ 7) \rightarrow a$	$p_6 : (6,\ 3) \rightarrow d$
$p_3 : (6,\ 7) \rightarrow a$	$p_7 : (7,\ 9) \rightarrow a$
$p_4 : (7,\ 2) \rightarrow d$	$p_8 : (8,\ 5) \rightarrow d$
$p_5 : (8, 10) \rightarrow d$	
(b) A set of passed tests	(c) A set of failed tests

Figure 2: An example faulty firewall policy with its failed and passed tests

fault. Due to the first-match semantics, changing a rule can affect the functionality of all the subsequent rules. Without thorough consideration, correcting a fault may introduce a new fault into the firewall policy.

In this paper, we formalize the problem of correcting a faulty firewall policy as follows:

Given a faulty firewall policy FW, a set of passed tests PT, and a set of failed tests FT, where $|PT| \geq 0$ and $|FT| > 0$, find a sequence of modifications $\langle M_1, \cdots, M_m \rangle$, where M_j $(1 \leq j \leq m)$ denotes one modification, such that the following two conditions hold:

1. *After applying $\langle M_1, \cdots, M_m \rangle$ to FW, all tests in $PT \cup FT$ become passed tests.*
2. *No other sequence that satisfies the first condition has a smaller number of modifications than m.*

Correcting a faulty firewall policy with the minimum number of modifications is a global optimization problem and hard to solve because the policy may consist of a large number of rules, and different combinations of modifications can be made. We propose a greedy algorithm to address this problem. For each step, we correct one fault in the policy such that the number of passed tests increases (or the number of failed tests decreases). To determine which correction technique should be used at each step, we try the five correction techniques. Then, we calculate the number of passed tests for each type of modifications and choose the correction technique that corresponds to the maximum number of passed tests. We then repeat the preceding step until there are no failed tests. Figure 3 illustrates our approach for automatic correction of firewall policy faults.

Our greedy algorithm can guarantee to find a sequence of modifications that satisfies the first condition. For each step, the greedy algorithm can increase at least one passed test because of the *rule addition* technique. Using this technique, we can at least convert each failed test

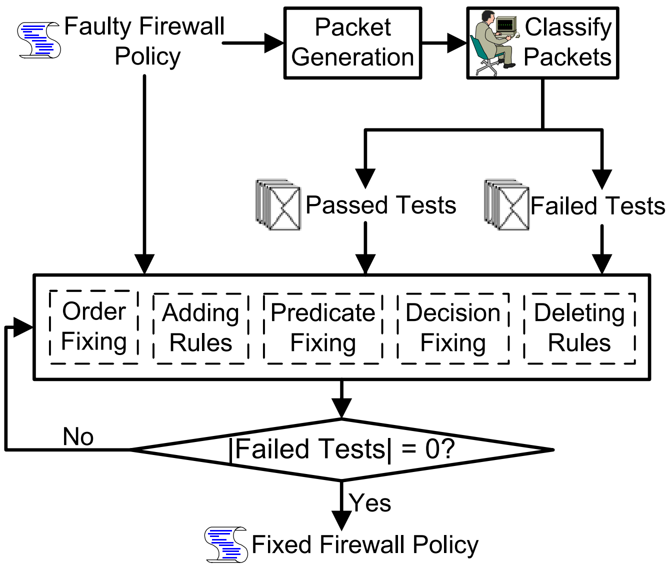


Figure 3: Overview of automatically correcting a faulty firewall policy

to a singleton rule and then add these singleton rules at the beginning of the faulty firewall policy. For example, convert the failed test $(6,\ 3) \rightarrow d$ in Figure 2(c) to a singleton rule $F_1 \in [6,6] \wedge F_2 \in [3,3] \rightarrow d$. However, the greedy algorithm cannot guarantee to find the global optimization solution that satisfies the second condition.

Note that administrators can supervise this process. For each step, administrators can choose their preferred technique for correcting a fault in the policy. If administrators do not want to supervise the process, our greedy algorithm can automatically produce the fixed policy.

Further note that without any restriction, our automatic approach for correcting firewall policy faults could introduce potential faults in the firewall policy. However, an administrator typically has some critical requirements when he/she designs the firewall policy. These critical requirements define that some packets should be accepted or discarded. The administrator can restrict the proposed approach not to violate the critical requirements. Consider a critical requirement that a data server in an organization should not be accessed by any outside connection. For each step of our greedy algorithm, if the modification generated in this step violates the requirement, the approach can simply choose the next modification that does not violate the requirement.

In the next five sections, we discuss our scheme for each correction technique, respectively. Recall that the last rule of a firewall policy is usually specified as $F_1{\in}D(F_1) \wedge \cdots \wedge F_d{\in}D(F_d) \rightarrow \langle decision \rangle$. Checking whether the last rule is correct is trivial. Therefore, we assume that the last rule of a firewall policy is correct in our discussion.

6 Order Fixing

Due to the first-match semantics, changing the order of two rules in a firewall policy (*i.e.*, swapping two rules) affects its functionality. Therefore, after swapping two rules of a firewall policy, we need to test and reclassify all passed tests and failed tests. It is computationally expensive to directly swap every two rules in a faulty firewall policy and then find the two rules such that swapping them can maximize the increased number of passed tests. Given a firewall policy with n rules, without considering the last rule, there are $(n-1)(n-2)/2$ pairs of rules that can be swapped. Furthermore, for each swapping, we need to reclassify all passed and failed tests. Assume that the number of passed tests is m_1 and the number of failed tests is m_2. The computational cost of this brute-force way is $(n-1)(n-2)(m_1+m_2)/2$.

To address this challenge, we use all-match firewall decision diagrams (all-match FDDs) [15] as the core data structure. An all-match FDD is a canonical representation of a firewall policy such that any firewall policy can be converted to an equivalent all-match FDD. Figure 4 shows the all-match FDD converted from the faulty firewall policy in Figure 2. An all-match FDD for a firewall policy $FW{:}\langle r_1, \cdots, r_n \rangle$ over attributes $F_1, \cdots, F_d$ is an acyclic and directed graph that has the following five properties:

1. There is exactly one node that has no incoming edges. This node is called the *root*. The nodes that have no outgoing edges are called *terminal* nodes.

2. Each node v has a label, denoted as $F(v)$. If v is a nonterminal node, then $F(v) \in \{F_1, \cdots, F_d\}$. If

v is a terminal node, then $F(v)$ is a list of integer values $\langle i_1, \cdots, i_k\rangle$ where $1 \leq i_1 < \cdots < i_k \leq n$.

3. Each edge $e{:}u \rightarrow v$ is labeled with a nonempty set of integers, denoted as $I(e)$, where $I(e)$ is a subset of the domain of u's label (*i.e.*, $I(e) \subseteq D(F(u))$). The set of all outgoing edges of a node v, denoted as $E(v)$, satisfies two conditions: (1) *consistency*: $I(e) \cap I(e') = \emptyset$ for any two distinct edges e and e' in $E(v)$; (2) *completeness*: $\bigcup_{e \in E(v)} I(e) = D(F(v))$.

4. A directed path from the root to a terminal node is called a *decision path*. No two nodes on a decision path have the same label. Given a decision path $\mathcal{P}{:}(v_1 e_1 \cdots v_d e_d v_{d+1})$, the matching set of $\mathcal{P}$ is defined as the set of all packets that satisfy $F(v_1) \in I(e_1) \wedge \cdots \wedge F(v_d) \in I(e_d)$. We use $C(\mathcal{P})$ to denote the matching set of $\mathcal{P}$.

5. For any decision path $\mathcal{P} : (v_1 e_1 \cdots v_d e_d v_{d+1})$ where $F(v_{d+1}) = \langle i_1, \cdots, i_k\rangle$, if $C(\mathcal{P}) \cap C(r_j) \neq \emptyset$, $C(\mathcal{P}) \subseteq C(r_j)$ and $j \in \{i_1, \cdots, i_k\}$.

For ease of presentation, we use $\{\mathcal{P}_1, \cdots, \mathcal{P}_h\}$ to denote the all-match FDD of the firewall policy FW. Based on this definition, we can draw the following theorem, the proof of which is in Appendix A.

Theorem 6.1 *Given two firewall policies* $FW_1{:}\langle r^1_1, \cdots, r^1_n\rangle$ *and* $FW_2{:}\langle r^2_1, \cdots, r^2_n\rangle$, *and their all-match FDDs* $\{\mathcal{P}^1_1, \cdots, \mathcal{P}^1_{h_1}\}$ *and* $\{\mathcal{P}^2_1, \cdots, \mathcal{P}^2_{h_2}\}$, *if* $\{r^1_1, \cdots, r^1_n\} = \{r^2_1, \cdots, r^2_n\}$, *without considering terminal nodes,* $\{\mathcal{P}^1_1, \cdots, \mathcal{P}^1_{h_1}\} = \{\mathcal{P}^2_1, \cdots, \mathcal{P}^2_{h_2}\}$.

According to Theorem 6.1, for swapping two rules, we only need to swap the sequence numbers of the two rules in the terminal nodes of the all-match FDD. For finding two rules such that swapping them maximizes the number of passed tests, our correction technique includes five steps:

(1) Convert the policy to an equivalent all-match FDD.

(2) For each failed test p, we find the decision path $\mathcal{P}$: $(v_1 e_1 \cdots v_d e_d v_{d+1})$ that matches p (*i.e.*, $p \in C(\mathcal{P})$). Let $\langle i_1, \cdots, i_k\rangle$ $(1 \leq i_1 < \cdots < i_k \leq n)$ denote $F(v_{d+1})$. Note that the decision of r_{i_1} is not the expected decision for the failed test p; otherwise, p should be a passed test.

(3) Find the rules in $\{r_{i_2}, \cdots, r_{i_k}\}$ whose decisions are the expected decision of p. Suppose $\{r_{j_1}, \cdots, r_{j_g}\}$ are those rules that we find for p, where $\{r_{j_1}, \cdots, r_{j_g}\} \subseteq \{r_{i_2}, \cdots, r_{i_k}\}$. Because the decision of rules in $\{r_{j_1}, \cdots, r_{j_g}\}$ is the expected decision for p, swapping r_{i_1} with any rule in $\{r_{j_1}, \cdots, r_{j_g}\}$ changes p to a passed test. Note that because the last rule of a firewall is a default rule, we cannot swap it with any preceding rule. If r_{j_g} is the last rule of the faulty firewall (*i.e.*, $j_g = n$), we delete r_{j_g} from $\{r_{j_1}, \cdots, r_{j_g}\}$.

(4) For all failed tests, we find out all rule pairs such that swapping two rules in a rule pair may increase the number of passed tests. Then we swap two rules in each rule pair. Note that swapping two rules in a rule pair changes the corresponding failed test to a passed test. However, this modification may change some passed tests to failed tests. Therefore, after swapping two rules in each rule pair, we reclassify all tests and calculate the number of passed tests.

(5) Find a rule pair such that swapping the two rules in this pair can maximize the number of passed tests.

Note that if there are more than one rule pair such that swapping two rules in each pair can maximize the increased number of passed tests, we choose the rule pair that affects the functionality of the minimum number of original firewall rules. Let $(r_{i_1}, r_{j_1}), \cdots, (r_{i_g}, r_{j_g})$ denote these rule pairs, where $i_k \leq j_k$ $(1 \leq k \leq g)$. Due to the first-match semantics, we choose the rule pair (r_i, r_j) where i is the maximum integer in $\{i_1, \cdots, i_g\}$.

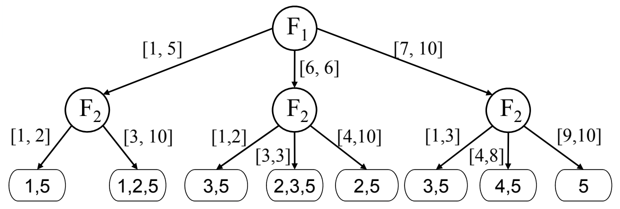


Figure 4: All-match FDD converted from the faulty firewall policy in Figure 2

For the faulty firewall policy in Figure 2, we first convert the faulty firewall policy to an all-match FDD, which is shown in Figure 4. Second, for each failed test, we find the corresponding rule pairs. In the example, we find only one rule pair (r_2, r_3) for the failed test $(6,\ 3) \rightarrow d$. Third, after swapping r_2 and r_3, $(6,\ 2) \rightarrow d$ becomes a passed test and no passed test changes to a failed test. Therefore, swapping r_2 and r_3 increases the number of passed tests by 1.

7 Rule Addition

There are two challenges for adding a rule to a faulty firewall policy. First, given a faulty firewall policy with n rules, there are n positions where we can add a rule. Determining which position is the best for adding a rule is a challenge. Second, because the predicate of a firewall rule is composed of multiple fields and the number of possible values in each field is typically large, brute-force addition of every possible rule for each position is computationally expensive. Considering a firewall rule with five fields (*i.e.*, 32-bit source IP, 32-bit destination IP, 16-bit source port, 16-bit destination port, and 8-bit protocol type) and two possible decisions (*i.e.*, *accept* and *discard*), the number of possible firewall rules that we can add for each position is $O(2^{204})$, because for each field with d-bit length, the number of possible ranges is $\binom{2}{2^d}=O(2^{2d-1})$. Furthermore, after adding a rule, we still need to reclassify all passed and failed tests.

The basic idea of our solution is that for each position, we first find all possible failed tests that can be corrected by adding a rule at this position, and then compute a rule that matches the maximum number of failed tests. To avoid changing a passed test to a failed test, the rule that we compute does not match any possible passed test. More formally, given a faulty firewall policy with n rules $\langle r_1, \cdots, r_n\rangle$, let position i $(1\leq i\leq n)$ denote the position between r_{i-1} and r_i. Note that we cannot add a rule after r_n because r_n is the default rule. Our correction technique for adding a rule includes five steps:

(1) For each position i, find a set of passed tests $PT(i)$ and a set of failed tests $FT(i)$ such that any test p in $PT(i) \cup FT(i)$ does not match any rule r_j $(1 \leq j \leq i-1)$. Note that when $i = 1$, r_j does not exist. In such case, $PT(1) = PT$ and $FT(1) = FT$. Due to the first-match semantics, if a failed test p matches a rule r_i, adding a rule after rule r_i cannot change the decision of p and hence cannot correct p. Therefore, the set $FT(i)$ includes all possible failed tests that we can correct by adding a rule at position i.

(2) Based on the expected decisions of tests, divide $PT(i)$ into two sets $PT(i)_a$ and $PT(i)_d$ where $PT(i)_a$ consists of all passed tests with expected decision *accept* and $PT(i)_d$ consists of all passed tests with *discard*. Similarly, we divide $FT(i)$ into two sets $FT(i)_a$ and $FT(i)_d$. The purpose is that adding a rule cannot correct two failed tests with different expected decisions.

(3) For set $FT(i)_a$, compute a rule with decision *accept*, denoted as $r'_{i,a}$, that satisfies two conditions:

   (a) No passed test in $PT(i)_d$ matches $r'_{i,a}$.

   (b) Under Condition (a), $r'_{i,a}$ matches the maximum number of failed tests in $FT(i)_a$.

   The algorithm for computing rule $r'_{i,a}$ based on $FT(i)_a$ and $PT(i)_d$ is discussed in Section 7.1.

(4) Similar to Step 3, for set $FT(i)_d$, compute a rule with decision *discard*, denoted as $r'_{i,d}$, that satisfies two conditions:

   (a) No passed test in $PT(i)_a$ matches $r'_{i,d}$.

   (b) Under Condition (a), $r'_{i,d}$ matches the maximum number of failed tests in $FT(i)_d$.

(5) Find a rule $r'_{j,\ decision}$ $(1\leq j\leq n)$ that corrects the maximum number of failed tests and then add $r'_{j,\ decision}$ to position j.

Note that if there is more than one rule that can correct the maximum number of failed tests, we choose rule $r'_{j,\ decision}$ where j is the maximum integer among these rules such that adding this rule affects the functionality of the smallest number of original rules in a firewall policy.

For the faulty policy in Figure 2, Figure 5 shows the four sets $PT(i)_a$, $PT(i)_d$, $FT(i)_a$, and $FT(i)_d$ for each rule of the policy.

	$PT(i)_a$	$PT(i)_d$	$FT(i)_a$	$FT(i)_d$
r_1	p_1, p_2, p_3	p_4, p_5	p_7	p_6, p_8
r_2	p_3	p_4, p_5	p_7	p_6, p_8
r_3	–	p_4, p_5	p_7	p_8
r_4	–	p_5	p_7	p_8
r_5	–	p_5	p_7	–

Figure 5: $PT(i)_a$, $PT(i)_d$, $FT(i)_a$, and $FT(i)_d$ for each rule in Figure 2

7.1 Computing Rules $r'_{i,a}$ and $r'_{i,d}$

Without loss of generality, in this section, we discuss the algorithm for computing $r'_{i,a}$ based on a set of failed tests $FT(i)_a$ and a set of passed tests $PT(i)_d$. First, we generate a rule that can match all failed tests in $FT(i)_a$. Suppose that the predicate of a firewall rule is composed of d fields. For each field j $(1\leq j\leq d)$, assume that x_j is the minimum value of all failed tests in $FT(i)_a$ and y_j is the maximum value. Therefore, the rule r:$F_1\in[x_1, y_1]\wedge\cdots\wedge F_d\in[x_d, y_d]\rightarrow a$ matches all failed tests in $FT(i)_a$. Second, we use the passed tests in $PT(i)_d$ to split the rule to multiple rules, each of which does not match any passed test. Let $(z_1, \cdots, z_d)\rightarrow d$ denote the first passed test p in $PT(i)_d$. If rule r matches p, for each field j, we generate two rules by using z_j to split $[x_j, y_j]$ into two ranges $[x_j, z_j - 1]$ and $[z_j + 1, y_j]$. The resulting two rules for field j are as follows.

$$F_1\in[x_1, y_1]\wedge\cdots\wedge F_{j-1}\in[x_{j-1}, y_{j-1}]\wedge \boldsymbol{F_j\in[x_j, z_j - 1]} \\ \wedge F_{j+1}\in[x_{j+1}, y_{j+1}]\wedge\cdots\wedge F_d\in[x_d, y_d]\rightarrow a$$

$$F_1\in[x_1, y_1]\wedge\cdots\wedge F_{j-1}\in[x_{j-1}, y_{j-1}]\wedge \boldsymbol{F_j\in[z_j + 1, y_j]} \\ \wedge F_{j+1}\in[x_{j+1}, y_{j+1}]\wedge\cdots\wedge F_d\in[x_d, y_d]\rightarrow a$$

Note that if $x_j>z_j-1$ (or $z_j+1>y_j$), the rule that includes $[x_j, z_j - 1]$ (or $[z_j + 1, y_j]$) is meaningless and it should be deleted from the resulting rules. If rule r does not match p, p cannot split r. Then, we use the second test in $PT(i)_d$ to split the resulting rules generated from p. Repeat this step until we check all the passed tests in $PT(i)_d$. Finally, we choose one rule that matches the maximum number of failed tests.

Take two sets $PT(2)_a$ and $FT(2)_d$ in Figure 5 as an example, rule $r'_{2,d}$ can be computed as $F_1 \in [6,8] \wedge F_2 \in [3,5] \rightarrow d$, which can correct two failed tests p_6 and p_8.

8 Predicate Fixing

There are two challenges for fixing a predicate in a faulty firewall policy. First, for a faulty firewall policy with n rules, there are $n-1$ possible predicates that we can correct. Note that the last rule r_n is the default rule. Second, similar to adding rules, brute-force fixing of the predicate for each rule is computationally expensive. The number of possible predicates for each rule is $O(2^{203})$.

The basic idea for predicate fixing is similar to adding rules. We first find all possible failed tests that can be corrected by fixing a predicate, and then compute a rule that matches the maximum number of failed tests. However, there are two major differences. First, for fixing the predicate of r_i, we compute only a rule with the same decision of r_i. Second, after fixing the predicate of rule r_i, the original rule r_i does not exist in the firewall policy. Therefore, the passed tests whose first-matching rule is r_i may become failed tests. The set of these passed tests for r_i can be computed as $PT(i)-PT(i+1)$ (shown in Figure 7). The passed tests whose first-matching rule is not r_i should be prevented from changing to failed tests. Therefore, the set of all possible failed tests that we can correct by fixing r_i's predicate is $FT(i) \cup (PT(i)-PT(i+1))$. Our correction technique for predicate fixing includes five steps:

(1) For each position i $(1 \leq i \leq n)$, find a set of passed tests $PT(i)$ and a set of failed tests $FT(i)$ such that any test p in $PT(i) \cup FT(i)$ does not match any rule r_j $(1 \leq j \leq i-1)$.

(2) For each rule r_i $(1 \leq i \leq n-1)$, compute the set of all possible failed tests $FT(i) \cup (PT(i)-PT(i+1))$ that we can correct by fixing r_i's predicate. Let $\widehat{FT(i)}$ denote $FT(i) \cup (PT(i)-PT(i+1))$. The complementary set of $FT(i) \cup (PT(i)-PT(i+1))$ is $PT(i+1)$, which is the set of passed tests that we cannot change to failed tests by fixing r_i's predicate.

(3) Based on the expected decisions of tests, divide $PT(i+1)$ into two sets $PT(i+1)_a$ and $PT(i+1)_d$, and divide $\widehat{FT(i)}$ into two sets $\widehat{FT(i)}_a$ and $\widehat{FT(i)}_d$.

(4) Without loss of generality, assume that r_i's decision is *accept*. For set $\widehat{FT(i)}_a$, we compute $r''_{i,a}$ that satisfies two conditions:

   (a) No passed test in $PT(i+1)_d$ matches $r''_{i,a}$.

   (b) Under condition (a), $r''_{i,a}$ matches the maximum number of failed tests in $\widehat{FT(i)}_a$.

   The algorithm for computing rule $r''_{i,a}$ based on $\widehat{FT(i)}_a$ and $PT(i+1)_d$ is the same as that in Section 7.1. Let r''_i denote the resulting rule.

(5) Find a rule r''_j $(1 \leq j \leq n-1)$ that can correct the maximum number of failed tests and then replace rule r_j.

Note that if there is more than one rule that can correct the maximum number of failed tests, we choose rule r''_j where j is the maximum integer among these rules.

For the faulty policy in Figure 2, Figure 6 shows the four sets $PT(i+1)_a$, $PT(i+1)_d$, $\widehat{FT(i)}_a$, and $\widehat{FT(i)}_d$ for each rule. Rule $r''_{2,a}$ can be computed as $F_1 \in [6,7] \wedge F_2 \in [7,9] \rightarrow a$, which can correct one failed test p_7.

	$PT(i+1)_a$	$PT(i+1)_d$	$\widehat{FT(i)}_a$	$\widehat{FT(i)}_d$
r_1	p_3	p_4, p_5	p_1, p_2, p_7	p_6, p_8
r_2	–	p_4, p_5	p_3, p_7	p_6, p_8
r_3	–	p_5	p_7	p_4, p_8
r_4	–	p_5	p_7	p_8

Figure 6: $PT(i+1)_a$, $PT(i+1)_d$, $\widehat{FT(i)}_a$, and $\widehat{FT(i)}_d$ for each rule in Figure 2

9 Decision Fixing

The idea of fixing a decision is that for each rule r_i, we first find the passed tests and failed tests whose first-matching rule is r_i. The set of the passed tests for r_i can be computed as $PT(i)-PT(i+1)$ and the set of the failed tests for r_i can be computed as $FT(i)-FT(i+1)$. If we change the decision of r_i, the passed tests in $PT(i)-PT(i+1)$ become failed tests and the failed tests in $FT(i)-FT(i+1)$ become passed tests. Then, we can calculate the increased number of passed tests by fixing r_i's decision. Finally, we fix the decision of the rule that corresponds to the maximum increased number of passed tests. Our correction technique for fixing a decision includes three steps:

(1) For each rule r_i $(1 \leq i \leq n-1)$, compute two sets $PT(i)-PT(i+1)$ and $FT(i)-FT(i+1)$.

(2) Calculate the increased number of passed tests by fixing r_i's decision, which is $|FT(i)-FT(i+1)|-|PT(i)-PT(i+1)|$.

(3) Fix the decision of a rule that can maximize the increased number of passed tests.

Note that if there is more than one rule such that fixing the decision of each of them can maximize the increased number of passed tests, we choose the rule with the maximum sequence number.

For the faulty policy in Figure 2, Figure 7 shows the two sets $PT(i) - PT(i+1)$ and $FT(i) - FT(i+1)$ for each rule. Clearly, fixing the decision of r_4 can change the failed test p_8 to a passed test.

	$PT(i) - PT(i+1)$	$FT(i) - FT(i+1)$
r_1	p_1, p_2	$-$
r_2	p_3	p_6
r_3	p_4	$-$
r_4	$-$	p_8

Figure 7: $PT(i){-}PT(i{+}1)$ and $FT(i){-}FT(i{+}1)$ for each rule in Figure 2

10 Rule Deletion

The idea of deleting a firewall rule is that we use the all-match FDD to calculate the increased number of passed packets by deleting each rule, and then delete the rule that can maximize the increased number of passed packets. Given a faulty policy with n rules and its all-match FDD, our correction technique for deleting a rule includes three steps:

(1) For each rule r_i $(1{\leq}i{\leq}n{-}1)$, find every decision path $\mathcal{P}{:}(v_1e_1 \cdots v_de_dv_{d+1})$ such that $C(\mathcal{P}){\subseteq}C(r_i)$ and i is the first rule id in $F(v_{d+1})$. Let $\{\mathcal{P}^i_1, \cdots, \mathcal{P}^i_h\}$ denote the set of such decision paths.

(2) For each decision path $\mathcal{P}^i_g{:}(v_1e_1 \cdots v_de_dv_{d+1})$ $(1 \leq g \leq h)$, find the set of passed tests $PT(\mathcal{P}^i_g)$ and the set of failed tests $FT(\mathcal{P}^i_g)$, where any test in $PT(\mathcal{P}^i_g)$ or $FT(\mathcal{P}^i_g)$ matches $\mathcal{P}^i_g$. Let $\langle i_1, \cdots, i_k\rangle$ $(1{\leq}i_1{<}{\cdots}{<}i_k{\leq}n)$ denote $F(v_{d+1})$. Note that $i_1 = i$ because of the first-match semantics. Let l_g denote the increased number of passed tests that match $\mathcal{P}^i_g$ after deleting rule r_i. To calculate l_g, we need to check whether r_i and r_{i_2} have the same decision. If r_i and r_{i_2} have the same decision, deleting r_i does not change two sets $PT(\mathcal{P}^i_g)$ and $FT(\mathcal{P}^i_g)$. In this case, $l_g{=}0$. Otherwise, the passed tests in $PT(\mathcal{P}^i_g)$ become failed tests and the failed tests in $FT(\mathcal{P}^i_g)$ become passed tests. In this case, $l_g = |FT(\mathcal{P}^i_g)| - |PT(\mathcal{P}^i_g)|$. Therefore, the increased number of passed packets after deleting rule r_i can be computed as $\sum_{g=1}^{h} l_g$.

(3) Delete the rule that can maximize the number of passed packets.

Note that if rule r_i is not the first-matching rule for any failed test, $|FT(\mathcal{P}^i_g)|{=}0$ $(1{\leq}g{\leq}h)$ and hence $\sum_{g=1}^{h} l_g \leq 0$. In this case, deleting r_i cannot increase the number of passed packets. We can easily find such rules by computing the set $FT(i){-}FT(i+1)$ for each rule r_i. Further note that if there is more than one rule such that deleting each of them can maximize the increased number of passed tests, we choose the rule with the maximum sequence number.

For the faulty firewall policy in Figure 2, by checking $FT(i){-}FT(i{+}1)$ in Figure 7, we find that deleting rule r_1 or r_3 cannot increase the number of passed packets. In the all-match FDD of the faulty policy (shown in Figure 4), for rule r_2, there are two paths, $F_1{\in}[6,6]{\wedge}F_2{\in}[3,3]$ and $F_1{\in}[6,6]{\wedge}F_2{\in}[4,10]$, where 2 is the first integer in their terminal nodes. Because the failed test p_6 matches the first path, and r_2 and r_3 have different decisions, deleting r_2 changes p_6 to a passed test. Because the passed test p_3 matches the second path, and r_2 and r_5 have different decisions, deleting r_2 changes p_3 to a failed test. Therefore, deleting r_2 does not increase the number of passed tests. Similarly, deleting r_4 changes p_8 to a passed test, and hence increases the number of passed tests by 1.

11 Experimental Results

11.1 Evaluation Setup

In our experiments, faulty firewall policies were generated from 40 real-life firewall policies that we collected from universities, ISPs, and network device manufacturers. The 40 real-life policies were considered as correct policies with respect to these faulty policies. Each firewall examines five fields, source IP, destination IP, source port, destination port, and protocol type. The number of rules for each policy ranges from dozens to thousands.

To evaluate the effectiveness and efficiency of our approach, we first employed the technique of mutation testing [6] to create faulty firewall policies. The technique for injecting synthetic faults with mutation testing is a well-accepted mechanism for carrying out testing experiments in both testing academia and industry. Particularly, each faulty policy contains one type of fault, and the number of faults in a faulty firewall policy ranges from 1 to 5. Given a real-life firewall with n rules, for each type of fault and each number of faults, we created $n{-}1$ faulty policies. Note that we did not change the last rule of a real-life policy. For example, to create a faulty firewall policy with k *wrong decisions* faults, we randomly chose k rules in a real-life firewall policy and then changed the decisions of the k rules. For each type of fault and each number of faults, we generated 35618 faulty firewall policies. Second, for each faulty policy, we employed a firewall testing tool [8] to generate test packets. Note that we generated test packets based on the faulty policy rather than its corresponding real-life policy. For each faulty policy, on average, the total number of passed and failed tests is about $3n$, where n is the number of rules in the policy. Third, we classified them into passed and failed tests. For each test packet, we compared two decisions evaluated by the faulty policy and its corresponding real-life policy. If the two decisions were the same, we classified the test packet as a passed test; otherwise, we classified it as a failed test. Note that in practice this step should be done by administrators. Finally, we implemented and applied our greedy algorithm over the faulty firewall policy and produced the fixed policy. For each step of the greedy algorithm, if different techniques increase the same number of passed tests, we randomly choose one technique.

11.2 Methodology

In this section, we define the metrics to measure the effectiveness of our approach. First, we define the *difference* between two firewall policies. Given two policies FW_1 and FW_2, the *difference* between FW_1 and FW_2, denoted as $\Delta(FW_1, FW_2)$, is the total number of packets each of which has different decisions evaluated by FW_1 and FW_2. To compute $\Delta(FW_1, FW_2)$, we first use a firewall comparison algorithm [13] to find the functional discrepancies between FW_1 and FW_2, where each discrepancy denotes a set of packets and each packet has different decisions evaluated by the two policies. Then, we compute the number of packets included by all discrepancies. Let FW_{real} denote a real-life firewall policy and FW_{faulty} denote a faulty policy created from FW_{real}. Let FW_{fixed} denote the fixed policy by correcting FW_{faulty} and $m(FW_{faulty})$ denote the number of modifications. Let $S(t,k)$ denote a set of faulty policies, where t denotes the type of fault and k denotes the number of faults in each faulty policy. We define two metrics for evaluating the effectiveness of our approach:

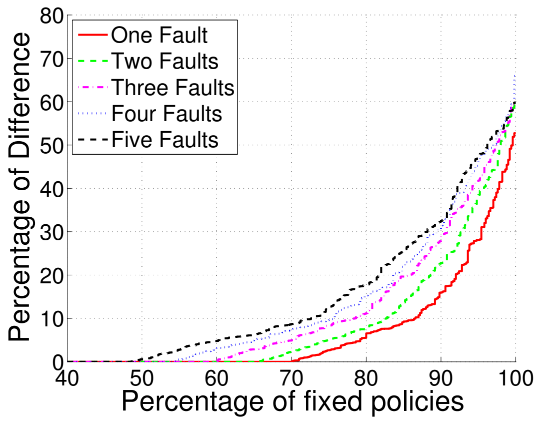
1. The *difference ratio* over FW_{real}, FW_{faulty}, and FW_{fixed}:
$$\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})}$$
2. The *average number of modifications* over $S(t,k)$:
$$\frac{\sum_{FW_{faulty} \in S(t,k)} m(FW_{faulty})}{|S(t,k)|}$$

Note that $\Delta(FW_{real}, FW_{faulty})$ is the total number of misclassified packets in the faulty firewall policy. For the example policy in Figure 1, if we generate a faulty firewall policy by changing r_1's decision to *discard*, the difference between these two policies is $2^8 \times 2^{16} = 2^{24}$. Hence, the total number of packets that are misclassified by the faulty firewall policy is 2^{24}. In fact, for a faulty policy, one failed test is a misclassified packet. But the number of failed tests is typically much smaller than the number of misclassified packets. For example, we may generate only one failed test (1.2.3.5, 192.168.1.1, 23447, 25, TCP)$\rightarrow$*accept* for the preceding faulty policy. After applying our approach over a faulty policy, the fixed policy FW_{fixed} not only corrects all failed tests, but also may correct other misclassified packets. The difference ratio $\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})}$ measures the percentage of misclassified packets after correcting FW_{faulty}. If $\frac{\Delta(FW_{real}, FW_{fixed})}{\Delta(FW_{real}, FW_{faulty})} = 0$, FW_{fixed} corrects all misclassified packets, which means that FW_{fixed} is equivalent to FW_{real} in terms of functionality.

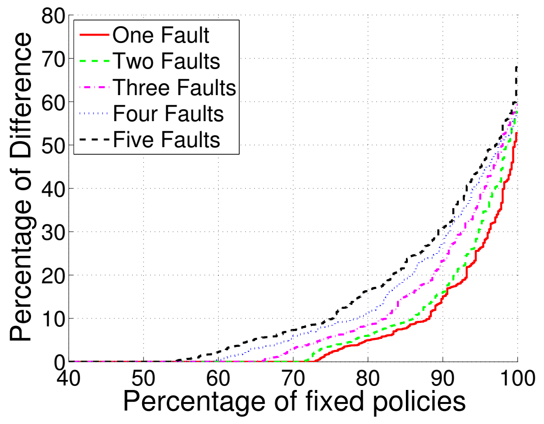
11.3 Effectiveness of Our Approach

Figures 8(a)-8(e) show the cumulative distribution of difference ratios over FW_{real}, FW_{faulty}, and FW_{fixed} for each type of fault. In Figures 8(a)-8(e), we use "One Fault", $\cdots$, "Five Faults" to denote the number of faults in faulty firewall policies. We observe that for three types of faults, wrong order, wrong decisions, and wrong extra rules, fixed policies can significantly reduce the number of misclassified packets. For faulty policies with k faults, where k faults are one of these three types and $k \leq 4$, over 53.2% fixed policies are equivalent to their corresponding real-life policies. For faulty policies with 1 to 5 *wrong decisions* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 73.5%, 68.8%, 63.7%, 59.3%, and 53.8%, respectively. For faulty policies with 1 to 5 *wrong order* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 69.7%, 64.2%, 59.7%, 54.3%, and 48.9%, respectively. For faulty policies with 1 to 5 *wrong extra rules* faults, the percentages of fixed policies that are equivalent to their corresponding real-life policies are 68.3%, 63.5%, 59.3%, 53.2%, and 47.3%, respectively.

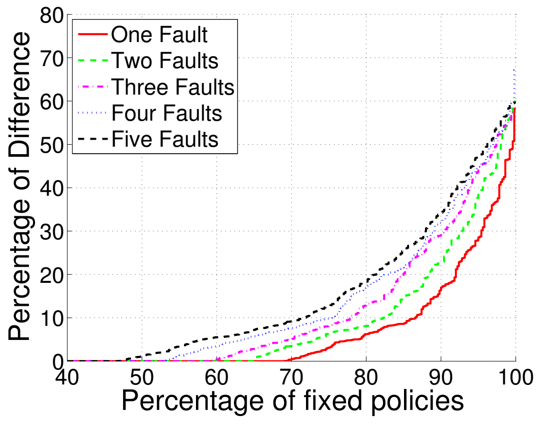
We also observe that fixed policies can reduce only a small number of misclassified packets for two types of faults, missing rules and wrong predicates. For faulty policies with 1 to 2 *missing rules* faults, the percentages of fixed policies that have 50% difference ratio with their corresponding real-life policies are 15.7% and 8.32%, respectively. For faulty policies with 1 to 2 *wrong predicates faults*, the percentages of fixed policies that have 50% difference ratio with their corresponding real-life policies are 17.3% and 9.1%, respectively. The reason is that in most cases, the information provided by failed tests is not enough to recover the missing rule (or correct predicate). A firewall rule (or predicate) with 5 fields can be denoted as a hyperrectangle over a 5-dimensional space, and failed tests are only some points in the hyperrectangle. To recover the missing rule (or correct the



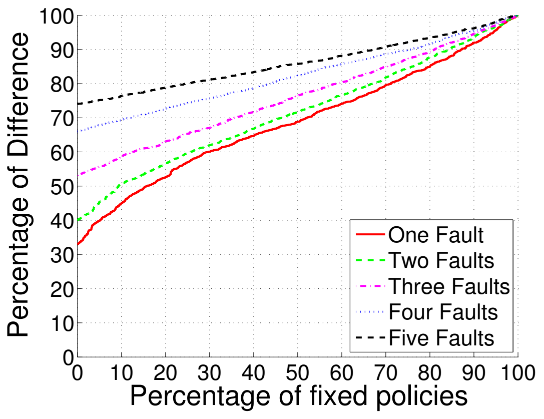
(a) Wrong Order



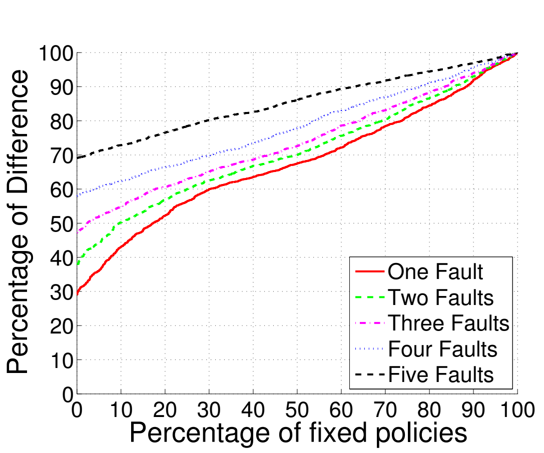
(b) Wrong Decisions



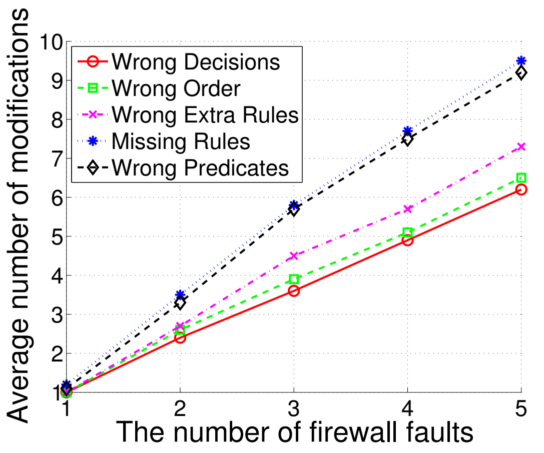
(c) Wrong Extra Rules



(d) Missing Rules



(e) Wrong Predicates



(f) Avg. Number of Modifications

Figure 8: Cumulative distribution of difference ratio and average number of modifications for each type of firewall policy faults

wrong predicate), for each surface of the hyperrectangle, there should be at least one point on it. However, the chance of such a case is very small.

Figure 8(f) shows the average number of modifications for each type of firewall faults. We observe that for faulty firewall policies with k faults, where $k \leq 5$, the ratio between the average number of modifications and the number of faults is less than 2. Note that to correct a faulty firewall policy with k faults, k is the minimum number of modifications. Therefore, the number of modifications of our approach is close to the minimum number.

11.4 Efficiency of Our Approach

We implemented our approach using Java 1.6.0. In our experiments, for a faulty firewall policy, we measure the total processing time of generating test packets, classifying packets into passed and failed tests, and fixing the policy to evaluate the efficiency of our approach. Note that classifying test packets is automatically done in our experiments by comparing two decisions evaluated by the faulty firewall and its corresponding real-life firewall for test packets. In practice, this step should be done by administrators. Our experiments were carried out on a desktop PC running Linux with 2 quad-core Intel Xeon at 2.3GHz and 16GB of memory. Our experimental results show that for the faulty firewall policy with 7652 rules, the total processing time for fixing this faulty policy is less than 10 minutes.

12 Case Study

In this section, we applied our automatic correction tool for firewall policy faults to a real-life faulty firewall policy with 87 rules and demonstrated that our tool can help the administrator to correct the misconfiguration in the firewall policy. The real-life firewall policy is shown in the Appendix B where the policy is anonymized due to the privacy and security concern.

We first employed the automated packet generation techniques [9] to generate test packets for the firewall policy and then asked the administrator to identify passed/failed tests. Among these test packets, we obtained seven failed tests, which are shown in Table 1. Second, we applied our proposed solution to this firewall policy and generated a sequence of modifications to correct the seven failed tests in Table 1. The resulting sequence includes four modifications: swapping rule 6 and rule 38, deleting rules 48, 49, and 50, which suggest that the firewall policy has one wrong-order fault and three wrong-extra-rule faults. We confirmed these faults with the administrator and he admitted that the resulting sequence of modifications generated by our tool can correct these faults automatically.

$p_1 : (157.96.252.36, 157.96.252.66, 13249, 25341, IP) \rightarrow a$
$p_2 : (67.48.121.156, 157.96.139.10, 4537, 109, TCP) \rightarrow a$
$p_3 : (35.121.47.232, 157.96.139.10, 21374, 109, TCP) \rightarrow a$
$p_4 : (25.35.113.153, 157.96.139.10, 7546, 110, TCP) \rightarrow a$
$p_5 : (154.182.56.79, 157.96.139.10, 16734, 110, TCP) \rightarrow a$
$p_6 : (193.21.135.85, 157.96.139.10, 19678, 143, TCP) \rightarrow a$
$p_7 : (213.174.191.25, 157.96.139.10, 24131, 143, TCP) \rightarrow a$

Table 1: Seven failed tests for the real-life firewall policy

13 Conclusions

We make three key contributions in this paper. First, we propose the first comprehensive fault model for firewall policies, including five types of faults. For each type of fault, we present an automatic correction technique. Second, we propose the systematic approach that can automatically correct all or part of the misclassified packets of a faulty firewall policy. To the best of our knowledge, our paper is the first one for automatic correction of firewall policy faults. Last, we implemented our approach and evaluated its effectiveness on real-life firewalls. To measure the effectiveness of our approach, we propose two metrics, which we believe are general metrics for measuring the effectiveness of firewall policy correction tools. The experimental results demonstrated that our approach is effective to correct a faulty firewall policy with three types of faults: wrong order, wrong decisions, and wrong extra rules.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579, NSF grant CNS-0716407, an MSU IRGP Grant, and an NIST grant.

References

[1] Agrawal, H., and Horgan, J. R. Dynamic program slicing. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (1990), pp. 246–256.

[2] Al-Shaer, E., and Hamed, H. Discovery of policy anomalies in distributed firewalls. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)* (2004), pp. 2605–2616.

[3] Baboescu, F., and Varghese, G. Fast and scalable conflict detection for packet classifiers. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)* (2002), pp. 717–735.

[4] CERT. Test the firewall system. http://www.cert.org/security-improvement/practices/p060.html.

[5] Cisco Reflexive ACLs. http://www.cisco.com/.

[6] DEMILLO, R. A., LIPTON, R. J., AND SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer 11*, 4 (1978), pp. 34–41.

[7] HARI, A., SURI, S., AND PARULKAR, G. M. Detecting and resolving packet filter conflicts. In In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)* (2000), pp. 1203–1212.

[8] HWANG, J., XIE, T., CHEN, F., AND LIU, A. X. Systematic structural testing of firewall policies. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems (SRDS)* (2008), pp. 105–114.

[9] HWANG, J., XIE, T., CHEN, F., AND LIU, A. X. Fault localization for firewall policies. In *Proceedings of IEEE International Symposium on Reliable Distributed Systems (SRDS)* (2009), pp. 100–106.

[10] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2005), pp. 273–282.

[11] JÜRJENS, J., AND WIMMEL, G. Specification-based testing of firewalls. In *Proceedings of International Conference Perspectives of System Informatics (PSI)* (2001), pp. 308–316.

[12] LIU, A. X. Change-impact analysis of firewall policies. In *Proceedings of European Symposium Research Computer Security (ESORICS)* (2007), pp. 155–170.

[13] LIU, A. X., AND GOUDA, M. G. Diverse firewall design. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 19*, 8 (2008), pp. 1237–1251.

[14] LIU, A. X., GOUDA, M. G., MA, H. H., AND NGU, A. H. Non-intrusive testing of firewalls. In *Proceedings of International Computer Engineering Conference (ICENCO)* (2004), pp. 196–201.

[15] LIU, A. X., ZHOU, Y., AND MEINERS, C. R. All-match based complete redundancy removal for packet classifiers in TCAMs. In *Proceedings of IEEE Conference on Computer Communications (INFOCOM)* (2008), pp. 574–582.

[16] LYU, M. R., AND LAU, L. K. Y. Firewall security: Policies, testing and performance evaluation. In *Proceedings of International Conference on Computer Systems and Applications (COMPSAC)* (2000), pp. 116–121.

[17] MARMORSTEIN, R., AND KEARNS, P. Assisted firewall policy repair using examples and history. In *Proceedings of USENIX Large Installation System Administration Conference (LISA)* (2007), pp. 1–11.

[18] NESSUS. http://www.nessus.org/.

[19] RENIERIS, M., AND REISS, S. P. Fault localization with nearest neighbor queries. In *Proceedings of IEEE International Conference on Automated Software Engineering (ASE)* (2003), pp. 30–39.

[20] SATAN. http://www.porcupine.org/satan/.

[21] WOOL, A. A quantitative study of firewall configuration errors. *IEEE Computer 37*, 6 (2004), pp. 62–67.

[22] YUAN, L., CHEN, H., MAI, J., CHUAH, C.-N., SU, Z., AND MOHAPATRA, P. FIREMAN: a toolkit for firewall modeling and analysis. In *Proceedings of IEEE Symposium on Security and Privacy (IEEE S&P)* (2006), pp. 199–213.

[23] ZELLER, A. Isolating cause-effect chains from computer programs. In *Proceedings of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)* (2002), pp. 1–10.

Appendix A

Before we prove Theorem 6.1, we first prove the following two lemmas.

Lemma 13.1 *Given a firewall policy* $FW{:}\langle r_1,\cdots,r_n\rangle$ *and its all-match FDD* $\{\mathcal{P}_1,\cdots,\mathcal{P}_h\}$*, for any rule* r_i *in* FW*, if* $\mathcal{P}_{i_1},\cdots,\mathcal{P}_{i_m}$ *are all the decision paths whose terminal node contains* r_i*, the following condition holds:* $C(r_i) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$.

Proof: According to property 5 in the definition of all-match FDDs, we have $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \subseteq C(r_i)$. Consider a packet p in $C(r_i)$. According to the consistency and completeness properties of all-match FDDs, there exists one and only one decision path that p matches. Let $\mathcal{P}$ denote this path. Thus, we have $p \in C(r_i) \cap C(\mathcal{P})$. According to property 5, i is in the label of $\mathcal{P}$'s terminal node. Thus, we have $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$. Therefore, $p \in \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$. Thus, we have $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \supseteq C(r_i)$.

Lemma 13.2 *Given a firewall policy* $FW{:}\langle r_1,\cdots,r_n\rangle$ *and its all-match FDD* $\{\mathcal{P}_1,\cdots,\mathcal{P}_h\}$*, for any rule* r_i *in* FW*, there exists only one set of paths* $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$ *such that* $C(r_i) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t})$.

Proof: Suppose there exists another one set of path $\{\mathcal{P}'_{i_1}, \cdots, \mathcal{P}'_{i_l}\}$, which is different from $\mathcal{P} \in \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$. Thus, there exists at least one $\mathcal{P}'_{i_s} \notin \{\mathcal{P}_{i_1}, \cdots, \mathcal{P}_{i_m}\}$ $(1 \leq s \leq l)$. According to the consistency and completeness properties of all-match FDDs, for any $\mathcal{P}_{i_t}$ $(1 \leq t \leq m)$, $\mathcal{P}_{i_t} \cap \mathcal{P}'_{i_s} = \emptyset$. Thus, $\cup_{t=1}^{m} C(\mathcal{P}_{i_t}) \neq \cup_{s=1}^{l} C(\mathcal{P}'_{i_s})$, which contradicts with our assumption.

Next we can prove Theorem 6.1 based on Lemma 13.1 and Lemma 13.2.

Proof of Theorem 6.1: Based on Lemma 13.2, for each rule $r_i^1 \in \{r_1^1, \cdots, r_n^1\}$ $(1 \leq i \leq n)$, we can find only one set of paths $\mathcal{P} \in \{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\}$ such that $C(r_i^1) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t}^1)$. Because $\{r_1^1, \cdots, r_n^1\} = \{r_1^2, \cdots, r_n^2\}$, there exists r_j^2 $(1 \leq j \leq n)$ such that $r_j^2 = r_i^1$. Thus, for each rule r_i^1 and its corresponding rule r_j^2, we have

$$C(r_i^1) = C(r_j^2) = \cup_{t=1}^{m} C(\mathcal{P}_{i_t}^1)$$

We also know that for the all-match FDD $\{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\}$ generated from FW_1:$\langle r_1^1, \cdots, r_n^1 \rangle$, the following condition holds:

$$\cup_{i=1}^{n} (\cup_{t=1}^{m} \mathcal{P}_{i_t}^1) = \{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\}$$

Similarly, for the all-match FDD $\{\mathcal{P}_1^2, \cdots, \mathcal{P}_{h_2}^2\}$ generated from FW_2:$\langle r_1^2, \cdots, r_n^2 \rangle$, we have

$$\cup_{i=1}^{n} (\cup_{t=1}^{m} \mathcal{P}_{i_t}^1) = \{\mathcal{P}_1^2, \cdots, \mathcal{P}_{h_2}^2\}$$

Thus, $\{\mathcal{P}_1^1, \cdots, \mathcal{P}_{h_1}^1\} = \{\mathcal{P}_1^2, \cdots, \mathcal{P}_{h_2}^2\}$.

Appendix B

The real-life firewall policy with 87 rules is shown as follows.

#	Src IP	Dest IP	Src Port	Dest Port	Protocol	Action
1	67.54.138.163	157.96.119.153	*	9100	TCP	accept
2	67.54.138.163	157.96.119.153	*	161	UDP	accept
3	*	*	*	*	53	deny
4	*	*	*	*	55	deny
5	*	*	*	*	77	deny
6	*	157.96.252.66	*	*	IP	accept
7	32.45.186.83	*	*	*	IP	deny
8	*	157.96.139.14	*	443	TCP	deny
9	231.49.182.251	*	*	*	IP	deny
10	*	*	*	3127	TCP	deny
11	*	*	*	2745	TCP	deny
12	*	*	4000	*	UDP	deny
13	*	*	*	111	UDP	deny
14	*	*	*	111	TCP	deny
15	*	*	*	2049	UDP	deny
16	*	*	*	2049	TCP	deny
17	*	*	*	7	UDP	deny
18	*	*	*	7	TCP	deny
19	*	*	*	6346	TCP	deny
20	*	*	*	7000	TCP	deny
21	*	*	*	161	UDP	deny
22	*	*	*	162	UDP	deny
23	*	*	*	1993	UDP	deny
24	*	*	*	67	UDP	deny
25	*	*	*	68	UDP	deny
26	*	*	*	49	UDP	deny
27	178.95.49.*	*	*	*	IP	deny
28	157.96.119.*	*	*	*	IP	deny
29	157.96.120.*	*	*	*	IP	deny
30	157.96.121.*	*	*	*	IP	deny
31	157.96.122.*	*	*	*	IP	deny
32	157.96.130.*	*	*	*	IP	deny
33	157.96.138.*	*	*	*	IP	deny
34	157.96.139.*	*	*	*	IP	deny
35	157.96.143.*	*	*	*	IP	deny
36	157.96.144.*	*	*	*	IP	deny
37	157.96.158.*	*	*	*	IP	deny
38	157.96.252.*	*	*	*	IP	deny
39	*	157.96.139.9	*	1949	UDP	accept
40	*	157.96.139.10	*	1949	UDP	accept
41	*	157.96.120.2	*	1949	UDP	accept
42	*	157.96.139.9	*	1949	TCP	accept
43	*	157.96.139.10	*	1949	TCP	accept
44	*	157.96.120.2	*	1949	TCP	accept
45	255.255.255.255	*	*	*	IP	deny
46	0.0.0.0	*	*	*	IP	deny
47	*	157.96.119.*	*	*	IP	deny
48	*	157.96.139.10	*	109	TCP	accept
49	*	157.96.139.10	*	110	TCP	accept
50	*	157.96.139.10	*	143	TCP	accept
51	62.78.103.*	*	*	*	IP	deny
52	*	*	*	6667	TCP	deny
53	*	*	*	6112	TCP	deny
54	*	*	*	109	TCP	deny
55	*	*	*	110	TCP	deny
56	*	*	*	1433	UDP	deny
57	*	*	*	1434	UDP	deny
58	*	*	*	135	TCP	deny
59	*	*	*	137	TCP	deny
60	*	*	*	138	TCP	deny
61	*	*	*	139	TCP	deny
62	*	*	*	445	TCP	deny
63	*	*	*	135	UDP	deny
64	*	*	*	137	UDP	deny
65	*	*	*	138	UDP	deny
66	*	*	*	139	UDP	deny
67	*	*	*	445	UDP	deny
68	*	*	*	143	TCP	deny
69	*	*	*	515	TCP	deny
70	*	*	*	512	TCP	deny
71	*	*	*	514	UDP	deny
72	*	*	*	69	UDP	deny
73	*	*	*	514	TCP	deny
74	*	157.96.138.138	*	5900	TCP	accept
75	*	157.96.138.138	*	5166	TCP	accept
76	*	157.96.138.138	*	*	IP	deny
77	*	157.96.138.101	*	5900	TCP	accept
78	*	157.96.138.101	*	5166	TCP	accept
79	*	157.96.138.101	*	*	IP	deny
80	*	157.96.138.80	*	*	IP	deny
81	*	157.96.138.82	*	*	IP	deny
82	*	157.96.138.234	*	*	IP	deny
83	*	157.96.138.235	*	*	IP	deny
84	*	157.96.138.236	*	*	IP	deny
85	*	157.96.128.*	*	*	IP	accept
86	*	157.96.140.*	*	*	IP	deny
87	*	*	*	*	IP	accept

Using TCP/IP traffic shaping to achieve iSCSI service predictability

J. Bjørgeengen
IT operations dept
University of Oslo
0373 Oslo
jarle.bjorgeengen@usit.uio.no

H. Haugerud
Faculty of Engineering
Oslo University College
0130 Oslo
Harek.Haugerud@iu.hio.no

Abstract

This paper addresses the unpredictable service availability of large centralized storage solutions. Fibre Channel is a common connection type for storage area networks (SANs) in enterprise storage and currently there are no standard mechanisms for prioritizing workloads using this technology. However, the increasing use of TCP/IP based network communication in SANs has introduced the possibility of employing well known techniques and tools for prioritizing IP-traffic. A method for throttling traffic to an iSCSI target server is devised: the packet delay throttle, using common TCP/IP traffic shaping techniques. It enables close-to-linear rate reduction for both read and write operations. All throttling is achieved without triggering TCP retransmit timeout and subsequent slow start caused by packet loss. A control mechanism for dynamically adapting throttling values to rapidly changing workloads is implemented using a modified proportional integral derivative (PID) controller. An example prototype of an autonomic resource prioritization framework is designed. The framework identifies and maintains information about resources, their consumers, response time for active consumers and their set of throttleable consumers. The framework is exposed to extreme workload changes and demonstrates high ability to keep read response time below a predefined threshold. It exhibits low overhead and resource consumption, promising suitability for large scale operation in production environments.

1 Introduction

Large scale consolidation of storage has been an increasing trend over the last years. There are two main reasons for this: rapid growth in the need for data-storage and economy of scale savings. Also, centralized storage solutions are essential to realize most of the cluster and server virtualization products existing today. In the last few years the storage market has shifted its focus from expensive fibre channel (FC) technology towards common-off-the shelf TCP/IP based technology. Storage networking is converging into familiar TCP/IP networking as performance of TCP/IP equipment increasingly gets more competitive with respect to performance. The times when dedicated storage administrators took care of storage area networks (SANs) are about to disappear as the underlying technology used to build SANs is shifting towards less specialized technology. iSCSI is an example of a technology enabling TCP/IP networks to connect hosts to their virtual disks in the theirs SANs. The growth in networked storage and the complexity in conjunction with large scale virtualization increase the demand for system administrators to understand and master complex infrastructures of which storage devices are a central part. Understanding the effects of performance and resource utilization in TCP/IP based SANs is vital in order to make keepable promises about storage performance. Predictable storage performance is a vital requirement for promising performance of the applications utilizing it, and it is the system administrator's job to ensure that storage performance meets the requirements of the applications. Figure 1 gives a simple overview of how several hosts share resources in an iSCSI storage appliance. Physical resource pools are colored, and virtual disks from those pools share the available I/O resources in the pool.

The advantages of storage consolidation/centralization are duly recognized. However, there is a major difference between performance attributes of a virtual disk in a centralized pool of storage and a dedicated local storage unit: sharing of the underlying hardware resources. A local disk may exhibit low total performance compared to SAN devices with a pool of many striped disks, but the performance of the local drive is predictable. The virtual disk in the storage pool usually has a much higher performance depending on the available capacity of the underlying hardware resources. The key point is the de-

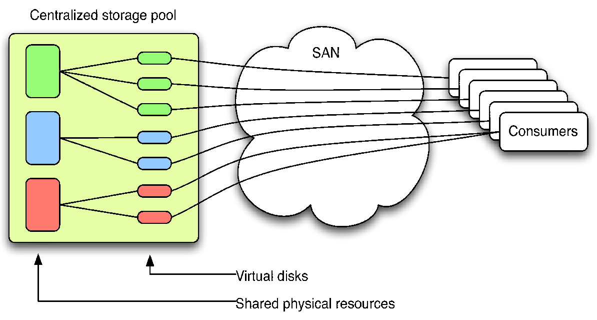


Figure 1: Concept of centralized storage pools

pendence on available capacity, and that available capacity is dependent on the activity towards other virtual disks sharing the same resource pool. A host may saturate the underlying resources of a storage pool causing poor performance of all hosts utilizing virtual disks from that pool. A host utilizing a virtual disk in a shared storage pool has no means to predict the behavior of other hosts utilizing other virtual disks sharing the same pool of resources. Hence, the performance experienced by any host utilizing a virtual disk served by a shared pool is unpredictable by the nature of resource sharing.

Addressing this issue requires a mechanism to prioritize workloads (Quality of Service,QoS) based on some kind of policy defining important and less important workload types. Most storage solutions are able to virtualize the amount of storage presented to the host in a flexible way, but the same storage devices seldom have QoS features. Storage service level agreements (SLAs) presupposes predictability in service delivery, but predictability is not present because of the nature of resource sharing and the absence of prioritization (QoS) mechanisms in storage devices. Application SLAs depend on individual components providing the application with sufficient resources, thus, contributing to the applications' SLA. The disk system is the component saturated first in any computer infrastructure, because it is the slowest one. This condition makes it hard, or impossible, to make keepable promises about performance, and ultimately increases the risk for application SLA violations. Clearly this is sub-optimal situation for system administrators whose mission is to keep applications and their infrastructure running.

The work presented in this paper is motivated by one of the author's experiences with unpredictable service availability of SAN devices at the University of Oslo.

2 System model and design

The goal of this work was to design a working prioritization framework containing throttling, measurements and decision making. The main idea was to utilize common tools in novel ways in order to obtain more predictable service availability of storage devices. The objective was to demonstrate the ability to mend adverse effects of interference between loads using a throttling mechanism for reducing resource contention, thereby improving service availability for important consumers. iSCSI utilizes TCP for transportation and Linux Traffic Control (`tc`) has advanced features for network traffic shaping, hence, the decision to use `tc` for the purpose of throttling was easy.

The amount of consumers that need to be throttled could become large. Also, workloads may rapidly change. Thus, a method to rapidly adapt throttling schemes is a necessary requirement. Traditionally, TCP traffic shaping with Linux Traffic Control is used with static rules targeted only at the network itself, for instance by limiting the network bandwith of traffic to specific IPs. This work utilizes feedback from resources outside of the network layer in order to adapt traffic throttling rules inside the networking layer in a dynamic manner.

In order to have sufficient control of the consumers' resource utilization, both read and write requests must be throttled. It is straightforward to shape outgoing TCP traffic from a server since the rate of transmissions is directly controlled. To the iSCSI server outgoing data translates to delivery of the answer to initiator read requests. Hence, controlling read requests are trivial but controlling write requests is a challenge. iSCSI write requests translates to inbound TCP traffic. Different approaches for dealing with the shaping of inbound traffic are known. The easiest method to achieve this is ingress policing. The concept of ingress policing is to drop packets from the sender when a certain bandwidth threshold is crossed. The congestion control mechanisms of TCP will then adjust the sender rate to a level that can be maintained without packet drops. There are clearly disadvantages to this approach:

- Packet loss which leads to inefficient network link utilization due to packet retransmits.
- The time it takes for the sender to adapt when the receiver decides to change the allowed bandwidth.

Ingress policing might be sufficient for a small number of senders and seldom changes in the receivers' accepted bandwidth. However, the ability to change bandwidth limitations fast is needed for rapid adaption to workload changes. When the number of consumers and bandwidth

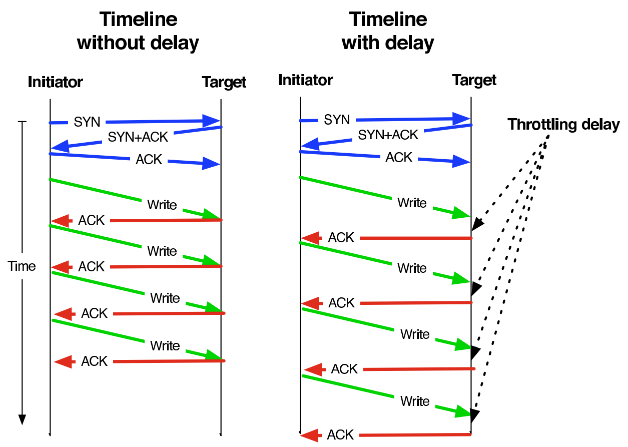


Figure 2: Principle of throttling by delaying packets

limits changes rapidly, this method does not scale, and adapts slowly and inefficiently.

2.1 Dynamic throttling

This paper suggests a novel method of throttling designed to address the limitations just described. The method implies introducing a variable additional delay to packets sent back to initiators, the clients in SCSI terminology. Read requests are simply throttled by delaying all outbound packets containing payload. Outbound ACK packets containing no payload are delayed in order to throttle write request without dropping packets. This method is illustrated in Figure 2. The actual delay is obtained using the `netem` module of Linux Traffic Control, and packets are assigned different delays based on `Iptables` marks.

In section 5 we propose an array agnostic version of this throttle by implementing it in a standalone bridge. The method of delaying packets makes this an attractive idea because of the delay method. Using packet delay rate instant rate reduction is achieved without dropping packets.

As previously argued, the need for a dynamic selection method for throttling packets is needed. `Iptables` provides this dynamic behavior with its many available criteria for matching packets. Combined with the `mark` target, which can be detected by the use of `tc's fw` filters, it is possible to set up a predefined set of delays that covers the needed throttling range with sufficient granularity.

The entities that consume resources in this context are the iSCSI initiators. The entity that provides the resources of interest to the initiators is the iSCSI target. Both initiators and targets have IP addresses. IP addresses can be used for throttling selections. The IP address of the iSCSI initiator will be chosen as the entity to which throttling will apply. Differing priorities for consumers will translate into different throttling schemes of those consumers' IP addresses. The underlying idea is to apply throttling to less important requests in order for important requests to have enough resources available to meet their requirements.

Packet delay throttling makes it possible to influence rates in both directions on a per initiator basis. In production environments the amount of initiators to keep track of quickly becomes overwhelming if throttling is based on individual consumer basis. Moreover, it is likely that the same throttling decisions should be applied to large groups of initiator IP addresses. Applying the same rules, over and over again, on lists of IP addresses is inefficient. To avoid this inefficiency the `Ipset` tool is needed [1]. It is a patch to the Linux kernel that enables creation of sets, and a companion patch to `Iptables` that makes `Iptables` able to match against those sets. This is a fast and efficient method of matching large groups of IP addresses in a single `Iptables` rule: the set of throttleable initiator IP addresses.

2.2 Throttling decision

As pointed out by previous research, remaining capacity is not constant, it is dependent on both rate, direction and pattern of the workloads. Hence, an exact measure of remaining capacity is hard to maintain. However, it is possible to indirectly relate how close the resource is to saturation by measuring individual consumer response times without any knowledge about the cause. In previous research virtual disk response time has successfully been utilized as a saturation level measure [2, 3, 4]. This work uses a similar approach. An Exponentially Weighted Moving Average (EWMA) of the response time is applied before it is used as the input signal to the control mechanism. EWMA is widely adopted as a successful method in the process control field for smoothing sensor input signals. In the process control field, this filter is commonly named a time constant low pass filter. The standard moving average is susceptible for spikes in the data. It is not desirable to trigger large throttling impacts caused by transient spikes in the average wait time, throttling should only occur as a result of persistent problems. The utilization of EWMA enables this behavior.

Interference from less important read or write jobs may lead the measured consumer response time to exceed the desired threshold. The framework should then respond by adding a throttling delay to the packets of the interfering loads, but it is difficult to determine the exact size of this delay. The idea of using a Proportional Integral Derivative (PID) controller as decision engine emerged from the observations of the relation between interfering workloads, the interference by other consumers and the efficient operation of the packet delay throttle. This behavior is similar to the control organs used to control industrial processes operated by PID controllers in the field of control engineering where they are widely used in order to keep process variables close to their set points and ensure stability for complete industry plants [5].

The purpose of our PID controller is to control throttling such that the consumer wait time of important requests stays below or equal to a preset value even when the load interference changes rapidly. The given value of maximum wait time for storage resources is likely to be constant, and close to the saturation point of the underlying resource. However there is nothing that prevents implementation of dynamically adjustable thresholds. The main purpose of the controller in this work is to keep response time of important requests from violating a given threshold in spite of rapidly changing amounts of interference from less important requests. The appendix contains a more detailed description of the PID controller.

2.3 Automated PID control approach

The ultimate goal of this work was the design of a fully automated per-resource read-response-time-controller as an example technique to utilize the throttle and the controller in order to ensure maximum read response times. Other prioritization schemes are equally possible. This section describes experimental results where the automated framework is exposed to the same loads as in the previous section. However, the selection of throttleable consumers are automatically inferred by the framework by the use of simple workload profiling: write activity of a certain amount.

Most I/O schedulers, and those parts of an entity responsible for servicing application I/O requests, generally have a preference for satisfaction of read requests over write requests. This is because waiting for read requests is blocking applications from continuing their work. Thus, read-over-write prioritization demonstrated here comprises a relevant use case for the throttle and the controller.

Usually, write requests are written to cache, at several levels in the I/O path, for later de-staging to permanent storage without blocking the application from further operation. Hence, throttling write requests can be done to a certain limit without affecting application performance. Nevertheless, it has been demonstrated through earlier experimental results that write requests are able to adversely influence the more important read requests. The design goal of the final prototype is the utilization of earlier results to automatically prevent write requests from adversely impacting read requests, thus contributing to improved application service predictability without the need for user input.

In the previous section the saturation level indicator and the set of throttleable consumers where predefined in order to influence the wait time of the important consumers. This section will describe the design of a prototype that completely automates the detection of saturation level and the identification of throttleable consumers, on a per resource basis. Instead of the prototype of the previous section's reliance on user determined list of important consumers, this prototype uses the read-over-write prioritization to automatically find out what to monitor and which consumers are eligible for write throttling.

In most storage devices, the disk group from which virtual disks are allocated, is bound to become the resource first saturated. This is the reason that LVM was chosen to reproduce a similar environment in the lab setup. In the lab setup, volume groups represent the shared resource that logical volumes are striped across. The objective of the prototype is to control the saturation level caused by write activity on a per-resource basis, thereby indirectly controlling the read response time of the resource. This translates to per volume group in the lab setup. In order to achieve this in the lab prototype, the following requirements will be met:

- An entity that maintains sets of IP addresses that are known to be doing write activity at a certain level: eligible throttlers.
  - Each set should have name of the resource of which its members are consumers.
  - Each set should be immediately throttleable by using its name.
- An entity that maintains a value representing the saturation level on a per-resource basis.
- An entity that spawns a PID controller for each resource and:
  - Uses the resource' saturation level as input.
  - Throttles the set of throttleable consumers for that particular resource so that the saturation level is kept below a set threshold.

The requirements are fulfilled by three perl programs working together with `Iptables, Ipset` and `Traffic Control`, utilizing shared memory for information exchange and perl threads for spawning parallel PID controllers. Figure 2.3 illustrates the concept of the framework implemented by the three scripts.

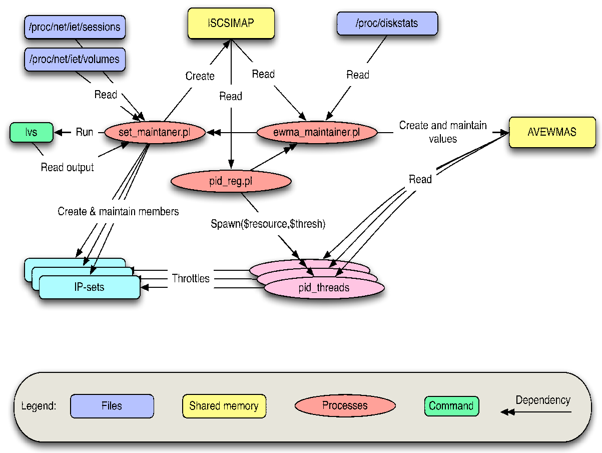


Figure 3: Automated controller framework overview

2.3.1 Automatic population of throttling sets

The Perl program `set_maintainer.pl` reads information about active iSCSI connections from `/proc/net/iet/*`, where information about each iSCSI target id is found: the connected consumer IP and servicing device. For all active iSCSI sessions, the device-mapper (`dm`) name and consumer IP address is recorded. The `lvs` command is used to record the logical volume name and volume group membership of each device-mapper device detected to participate in an active iSCSI session. The information found for each of the device-mapper device is recorded in a data structure and mapped into a shared memory segment with the key *ISCSIMAP*. For each of the volume groups involved in active iSCSI sessions, an empty IP-set is created with the same name as the volume group. When iSCSI device maps are exported to shared memory and the necessary IP-sets are created, the program enters maintenance mode. This is a loop that continuously monitors exponentially weighted averages (EWMAs) of the write sector rates of all `dm` devices involved in active iSCSI sessions. For each of the previously created IP-sets, it then determines the set of consumers that have a write sector rate exceeding a preset configurable threshold. The generated set is compared with the in-kernel IP-set for that resource, and any differences are converged to match the current set of eligible consumes for throttling that were detected. The IP-sets are converged once every second, yielding continuously updated per resource IP-sets known to contain consumers exhibiting write activity at certain level. These sets are immediately throttleable by `Iptables` matching against them.

2.3.2 Automatic determination of saturation monitors

The `ewma_maintainer.pl` program reads the shared memory information exported by the `set_maintainer.pl` program. For each resource, it continuously calculates an exponentially moving average of the read response time using information obtained from `/proc/diskstats`. Only consumers having read activity are included in the calculation. The data structure containing the resources' read response time EWMAs is tied to a shared memory segment with key *AVEWMAS* and updated every $100ms$. The read response time EWMAs serve as per resource saturation indicators which will be used as input values to the subsequently described PID controller threads.

2.3.3 Per resource PID control

The `pid_control.pl` program attaches to the shared memory segment with the key *AVEWMAS*, and reads the saturation indicators maintained by the `ewma_maintainer.pl` program. For each of the resources (volume groups) found in the *AVEWMAS* shared memory segment, a PID controller thread is created with the resource name and its accepted read response time threshold as parameters. Each PID control thread monitors the saturation level of its designated resource and directly controls the delay throttle of the set containing current consumers exhibiting write activity towards that resource. The `pid_control.pl` then detaches from the worker threads and enters an infinite sleep loop, letting the workers control resource saturation levels in parallel until a `SIGINT` signal is received.

3 Results

Experiments are executed using a Linux based iSCSI appliance using striped logical volume manager (LVM) volumes as virtual disks. Each of four striped logical volumes are presented to the blade servers using iSCSI enterprise daemon [6, 7]. The blade servers act as iSCSI initiators and are physically connected to the external iSCSI target server using a gigabit internal blade center switch. Figure 4 shows the architecture of the lab setup.

3.1 Without throttling

When there is no throttling mechanism in place, there is free competition for available resources. Figure 5 shows how four equal read loads, run on each of the equally powerful blade servers, share the total bandwidth of the disk resources, serving each of the logical volumes to

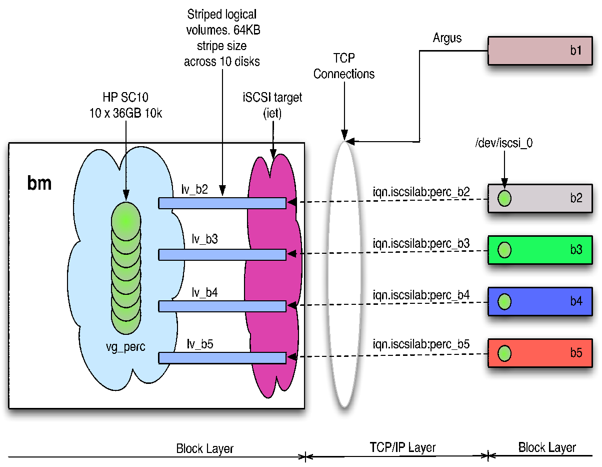


Figure 4: Concept sketch of the lab setup

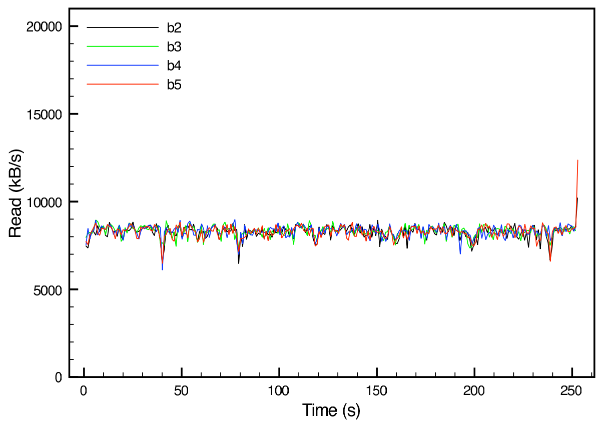


Figure 5: Equal sequential read load from four identically equipped blade servers without throttling

which the blade servers' iSCSI block devices are attached. The plotted read rates show what each of the consuming blade servers achieve individually.

3.2 Throttling by packet delay

Throttling of workloads has been utilized as a means to influence remaining capacity by many previous works, and it is normally carried out by some kind of rate limitation applied to the workloads. Utilization of the iSCSI protocol comes with the additional benefit of utilizing TCP traffic shaping tools to enforce rate limitation. In order to examine the effects on consumers by throttling taking place in the TCP layer, a number of experiments were executed. The first throttling approach involved bandwidth limitations by using hierarchical token bucket filters (HTB). The expected effect of throttling individual consumers was achieved, but the pure bandwidth throttler had a few practical limitations: the need for constantly calculating the bandwidth to be applied and, more important, the inefficient way of controlling write requests. Controlling write rates was not possible without packet loss, resulting in slow and inefficient convergence towards bandwidth target.

The shortcomings of the bandwidth shaping method, especially with respect to writing, inspired the idea of using packet delay for throttling. The `netem` module of Linux Traffic control was used to add delay to packets in a dynamic way in conjunction with `Iptables` packet marks. The concept is to add a small wait time to outgoing ACK packets containing no payload, thus slowing down the packet rate of the sender: the iSCSI writer. The main outcome of the design and subsequent experiments is an efficient way of throttling individual iSCSI consumers' traffic in both directions, with close-to-linear rate reduction and without packet loss. The experiments show that it is possible to throttle write and read activity using the same set of delay queueing disciplines (qdiscs) in Linux Traffic Control (`tc`). For writes, the outgoing ACK packets containing no payload are delayed, and for reads all other packets are delayed.

Figure 6 shows the effect of packet delay based throttling on the same workload as in Figure 5, and Figure 7 shows the effect when writing the same load that was previously read.

The shaping is done in using `Iptables'` packet marking abilities to place packets from individual consumers in different predefined delay qdiscs at different points in time. In this experiment, a shaping script on the target server is throttling down blade servers b2, b3 and b4 at predefined time offsets from the start time of the experiment and releasing them at later points in time. Throttling of blade server b2 frees up resources to the remaining consumers. Next, throttling of b3 and b4 gives increased resources to the remaining consumers. When b2 is freed, b5 is already done with its job, and most resources are available to b2 which increases its throughput dramatically. When b3 is freed, b2 and b3 share the resources again and stabilize at approximately 14 MB/s each. Finally b4 is freed, and b2, b3 and b4 share the resources, each having a throughput of ca. 10 MB/s. When b4 finishes its job, there are two machines left to share the resources, and when b3 finishes, only b2 is left to consume all resources.

Figures 6 and 7 shows a drop in throughput for un-

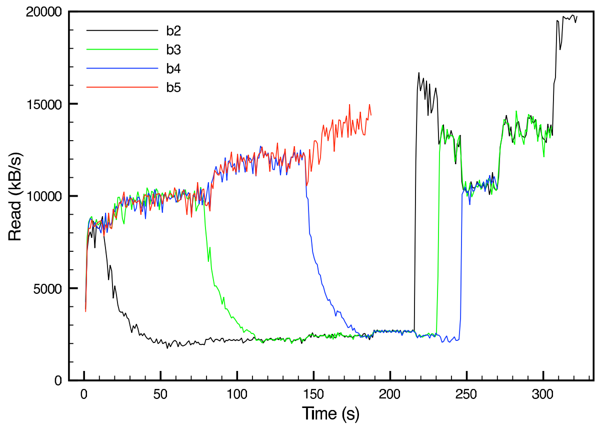


Figure 6: Throttling of initiator's sequential read activity using delayed ACK packets in tc(1).

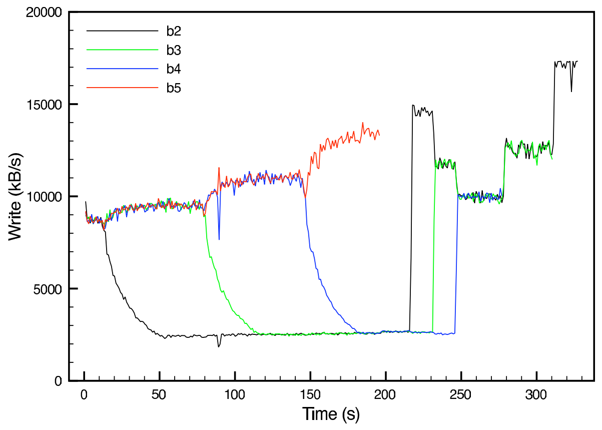


Figure 7: Throttling of initiator's sequential write activity using delayed ACK packets in tc(1).

throttled consumers when throttling starts. No plausible explanation was found for this, and additional research is necessary to identify the cause of this.

3.3 Introduced delay vs throughput

Previous results suggest that the method of introducing artificial delay to outgoing packets could be an efficient way of throttling iSCSI initiators in order to decrease the pressure on shared resources like disk groups. To find out the predictability of throttling as an effect of artificial delay, 200 MB of data was repeatedly read and written from the iSCSI initiator device of one blade server, measuring the time it took to complete each job. Each job were repeated 20 times for each value of artificial delay. Figures 9 and 8 show the results with error indicators, representing the standard error, on top of the bars. The precision of the means is so high that it is hard to see the error indicators at all.

The plots show that variation of artificial delay between 0 and 9.6 ms is consistently able to throttle reads between 22 MB/s and 5 MB/s and writes between 15 MB/s and 2.5 MB/s. There is no absolute relationship between artificial delay and throughput. Rather, the introduced delay has an immediate rate reducing effect regardless of what the throughput was when throttling started. Figures 9 and 8 suggests that there is a close-to-linear functional relationship between introduced delay, the start rate and the resulting rate after throttling.

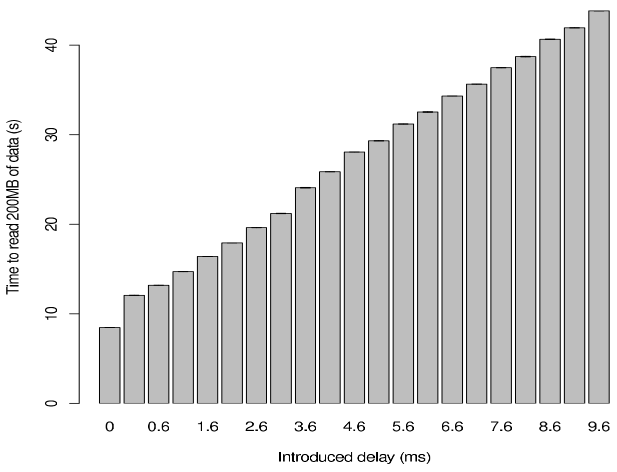


Figure 8: Repeated measurements of the time used to read 200 MB with stepwise increase in artificial delay of outgoing packets from target server.

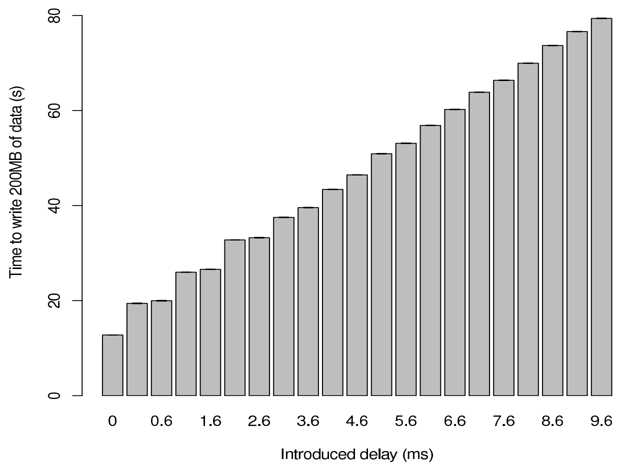


Figure 9: Repeated measurements of the time used to write 200 MB with stepwise increase in artificial delay of outgoing packets (ACK packets) from target server.

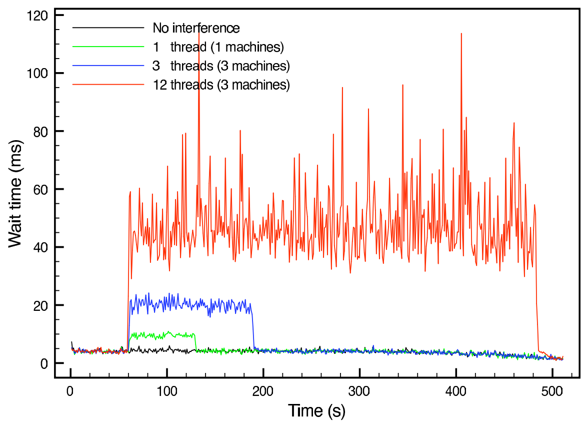


Figure 10: The effect on average wait time for a rate limited (256kB/s) random read job running on one server during interfering write activity from 1 and 3 other machines respectively. The interference is started one minute into the timeline.

3.4 Interference between loads

Figure 10 demonstrates that a small random read job, that causes negligible I/O load by itself, has its response time increased with the amount of load caused by threads running on other hosts. The graphs in the figure is from 4 different runs of the sram random read job, but with different degree of interference in the form of write activity to other logical volumes residing on the same striped volume group. This picture comprises the essence of load interference. The consumer executing the small random read job is unable to get predictable response times from its virtual disk because of activity from other storage consumers.

3.5 Effect of throttling on wait time

Figure 11 shows the effect on a small read job's average wait time when throttling the 12 interfering sequential writers. Packet delay throttling is done in the periods $100s - 190s$ and $280s - 370s$, using $4.6ms$ and $9.6ms$ packet delay respectively. Clearly the throttling of interference contributes to wait time improvement. The magnitude of improvement is higher if the wait time is high before throttling (i.e. level of saturation is high). It means that the throttling cost for improving response time from terrible to acceptable can be very low, but the cost of throttling increases as the response time improves (decreases).

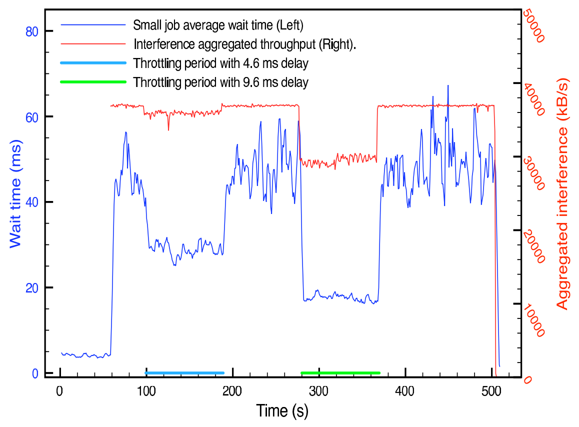


Figure 11: The effect on a small read job's wait time when throttling interfering loads with delays of 4.6 ms and 9.6 ms respectively.

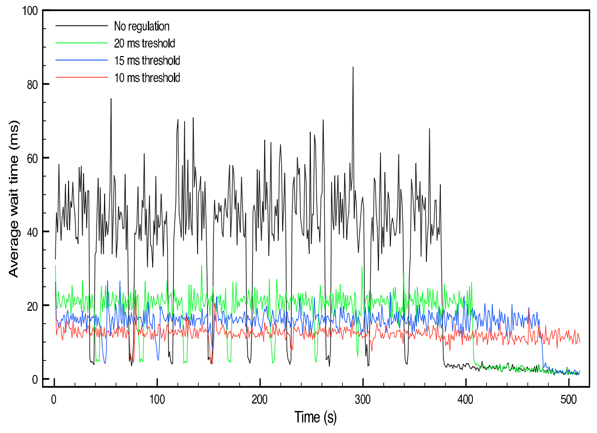


Figure 12: The average wait time of a rate limited (256kB/s) random read job with 12 interfering write threads started simultaneously and repeated with 5 seconds pause in between. The black plot shows the effect with free resource competition. The colored plots show how the PID regulator keeps different latency thresholds by regulating interfering workloads.

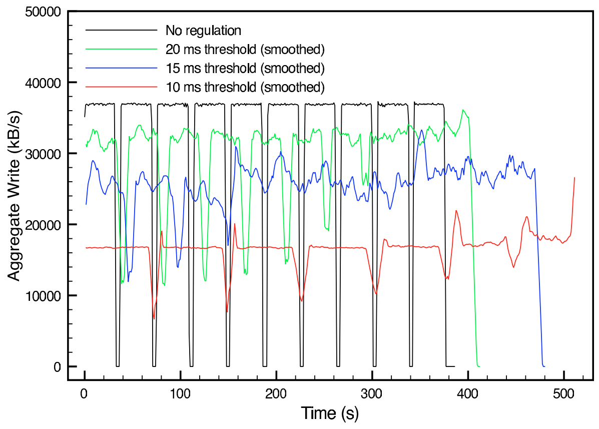


Figure 13: The aggregated throughput caused by throttling to keep latencies at the set thresholds in Figure 12.

3.6 PID control of response time

Figure 12 demonstrates the PID controller's ability to keep the actual wait time below or equal to the desired threshold. The black plot shows how latency is affected by various changing interfering workloads when no throttling is enabled. The colored plots show the effect of the same interfering workloads, but now with the PID regulator enabled having thresholds set to 20,15 and 10 ms respectively. Figure 13 shows the throttling effect on the corresponding interfering workloads (aggregated throughput). Notable is the relatively higher latency improvement for the random read job by throttling aggregate write throughput from its maximum of 39 MB/s down to 33 MB/s, yielding an improvement of 25 ms lower latency. Taking the latency down another five milliseconds costs another seven MB/s of throttling to achieve. Clearly the throttling cost for each step of improved latency increases as latency improves.

3.6.1 Automated PID results

Figure 14 shows that the results with per resource saturation level auto-detection, and dynamically maintained throttleable consumer sets, is close to the results in the previous section where throttleable consumers and response time monitors where defined manually. Figure 15 shows the resulting aggregated write rates as a consequence of the automated throttling carried out to keep read response time below the set thresholds in Figure 14. Again, the black plot depicts response-time/write-rate without regulation, and the colored ones depicts the same but with regulation at different threshold values.

The results shows that the automated per resource PID control framework is able to closely reproduce the previous results where throttleable consumer sets and resource saturation indicators were manually given as parameters to the PID regulators.

There is a slight delay in the throttle response compared to the previous section, giving a slightly larger magnitude and duration of the overshoot created by the simultaneous starting of 12 interfering threads. It is reasonable to speculate that this is caused by the additional time required to populate the sets of throttleable consumers.

During experiment execution, the OUTPUT chain of the Netfilter mangle table was monitored with the command `watch iptables -L OUTPUT -t mangle`. As expected, the rule that marks the outbound ACK packets of all consumers in the set of throttleable consumers appeared as soon as the response time threshold was violated. Further observation revealed rapid increase of the `mark` value as the write interference increased in magnitude, thus directly inhibiting write activity to a level that does not cause write-threshold violation. The command `watch ipset -L` was used to observe that an empty set with the same name as the active resources (the vg_aic volumgroup) were created upon startup of the `set_maintainer.pl` program. Furthermore, the set was populated with the correct IP

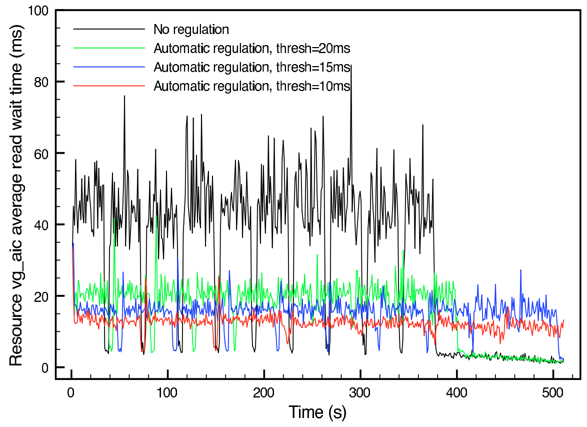


Figure 14: The average wait time of a rate limited (256kB/s) random read job interfered by 12 write threads started simultaneously and repeated with 5 seconds pause in between. The black plot shows the effect with free resource competition. The colored plots show how the PID regulator keeps different response time thresholds by regulating interfering workloads. In this plot, the resource saturation indicator and the set of throttleable host are maintained automatically.

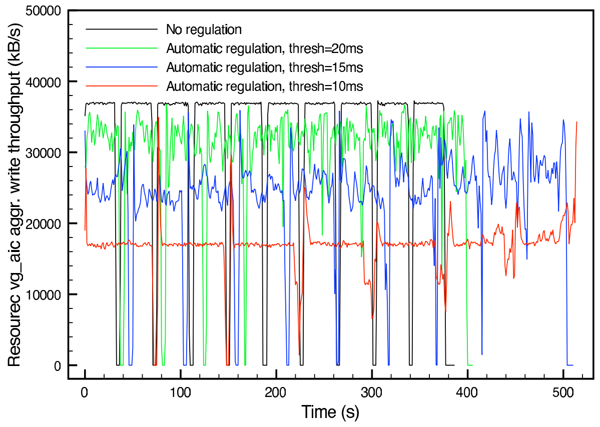


Figure 15: The aggregated throughput caused by throttling to keep latencies at the set thresholds in Figure 14

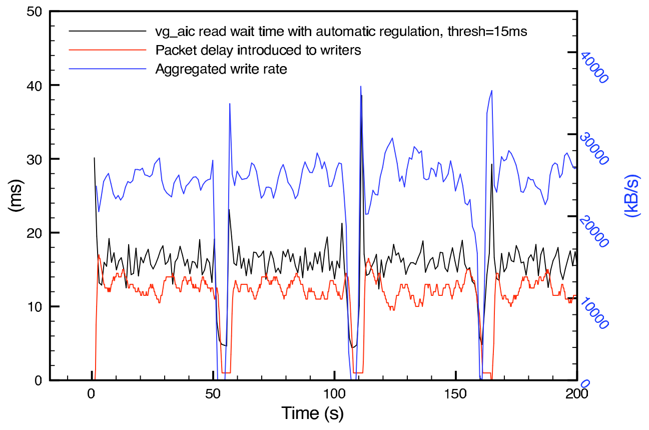


Figure 16: The resource average wait time, the throttling delay and the aggregated write rate with a set resource-wait-time-threshold of 15ms

addresses as the write activity of consumers violated the set threshold, and the IP addresses were removed from the set when consumers ceased/reduced write activity.

Before creating the workload used in this experiment, various smaller workloads were tested while plotting average wait time in realtime during experiments. By applying various increasing and decreasing write interference, the PID controller's behavior was observed in real time. The controller exhibited remarkable stability when gradually increasing interference. Hence, it was decided to produce the most extreme workload variation possible in the lab for the plotted results by turning on and off 12 writer threads (powered by three machines) simultaneously.

It is interesting to examine how the throttle-produced packet delay changes as the the PID controller decides throttle values. Thus, the experiments were run again, capturing the packet delay applied to the set of throttleable hosts along the duration of the experiment. Figure 16 shows the monitored resource's (vg_aic) actual wait time, the throttle value (packet delay) produced by the PID controller and the actual resource's aggregated write rate. The 12 writer threads want as much I/O bandwidth as they can get (37 MB/s without regulation), however, they get throttled by introducing the packet delay seen in the red plot. The decreased write rate caused by packet delay prevents resource saturation, which again prevents read response time of the resource from exceeding the set threshold of 15 ms.

3.7 Measuring overhead

This work introduces new features to prioritize workloads sharing a common resource. It is timely to ask if this new feature comes with an added overhead. When no throttling occurs overhead is unwanted. Since no `Iptables` rules are active when no throttling occurs, there is no overhead introduced by `Iptables`. The only possible source of overhead in this situation is the static `tc` queueing disciplines (qdiscs) and/or the static filters attached to the root qdisc. All outgoing packets are checked for marks by the static filters and there is a risk that this checking introduce overhead. To investigate if the existence of static delay queues and their filters add overhead, the difference in throughput was measured with static qdiscs present and absent.

Throttling only occurs when response time of a resource violates the preset threshold. When no throttling occurs, there is a negligible worst case overhead of 0.4% for reads and 1.7% for writes caused by the static traffic control filters which are always present and ready to detect packet marks. After the experiments where finalized we discovered that `Iptables` is able to classify packets directly to `tc` qdiscs making the check for `Iptables` marks superfluous and there will be no overhead at all when the treshold is not violated. This was confirmed by experiment.

4 Background and previous work

The challenges regarding storage QoS are well recognized and there has been numerous approaches to design of such systems like Stonehenge [8, 9, 10], Cello [11], Façade [12], Triage [13], Argon [14], Chameleon [15] and Aqua [16, 17].

Despite all the research done in the field, specifications regarding QoS functionality are seldom found in the specification sheets of storage devices.

The ability to specify service level objectives (response time and bandwidth), among other data management features, has been the subject of a decade long research at HP Labs Storage Systems department. Looking back in retrospect, Wilkes [18] points out the challenges of incorporating the research results into real production implementations. The challenge is to persuade users to trust the systems to do the right thing. This is a human challenge, one perhaps rooted in general healthy skepticism to new technology and bad experiences from earlier implementations that turned out to not fully take all real life parameters into account. Wilkes points out the need to remember that systems are built to serve people, and the success of technical accomplishments is dictated by how comfortable people ultimately are with them [18].

iSCSI based storage devices are the major competitor to FC based storage devices at the moment. With its lower cost, easier configuration and maintenance and increasingly competitive performance, iSCSI seems to be the enabler of large scale adoption of IP based SAN devices. The introduction of IP as a transportation layer introduces an additional, well known and well trusted toolbox for enforcing policy and fairness amongst storage consumers. Tools for traffic shaping in the TCP/IP layer have been around for many years. The combination of well known and trustworthy throttling mechanisms and an extended knowledge about storage system internals makes an appealing, pragmatic and non-intrusive approach to the problem of QoS in storage systems. Instead of introducing the need to build trust towards new interposed scheduling algorithms, bound to add uncertainty and overhead, this work suggests utilization of previously known and trusted tools to obtain workload prioritization in case of resource saturation. Lumb and coworkers point out the lack of a traffic shaper in storage systems [12] (presumably FC based storage systems). However, when utilizing TCP/IP as transport mechanisms, traffic shapers are available.

The work described in this paper takes a different approach to the problem by utilizing well known tools, with a high level of trust from other fields, and applying them to the storage QoS problem for iSCSI storage devices. The market for iSCSI based storage devices is growing rapidly, making it an interesting target for QoS research. The need for a throttling mechanism, as a means to control storage consumers, has been recognized by previous works [12, 15, 13, 2, 4, 3, 16, 17], and they interpose their own throttlers/schedulers in the critical data path. However, since iSCSI uses TCP for transportation, it is possible to use well known network traffic shaping tools for the purpose of this throttling. With the growing amount of virtual appliances utilizing iSCSI targets as their disk storage, our approach enables global storage QoS directly contributing to application SLAs using well known tools with established trust in the networking field.

5 Future work

This work opens several interesting paths for further research and applications. By using the fundamental ideas explored, it should be possible to create QoS modules to be used as external bridges in front of iSCSI appliances or integrated into Linux based iSCSI appliances similar to the lab prototype. By utilizing the ideas from this work, system administrators and vendors can offer QoS for iSCSI storage. Hence, they can offer differentiated SLAs to storage consumers with a confidence previously very difficult to achieve and contribute their share to overall application SLAs.

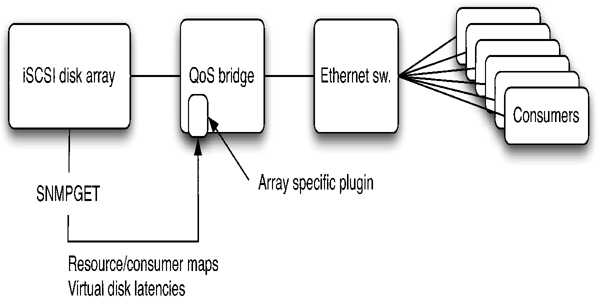


Figure 17: Illustration of how the framework could be utilized as an independent black box with limited array knowledge.

Figure 17 illustrates an approach for moving the controller to an external bridge. Information about consumer/resource mapping and virtual disk read latencies would be necessary in order to directly utilize the techniques demonstrated here. In the figure, usage of SNMP GET requests towards the array is suggested as an easy method for this purpose. However, the ultimate black box approach would be to infer this information from packet inspection. If achievable, this approach could serve as a self contained, non-intrusive, iSCSI QoS machine applicable to all iSCSI solutions regardless of their make and the feedback loop to the storage device would not be necessary. But it is unlikely that the actual consumer/resource mapping can be detected by packet inspection since this is internal storage device knowledge. However, it could be indirectly inferred by using a predefined initiator naming convention that contain resource membership.

Even with high sampling rate, and convergence rate of throttleable consumer sets, the PID controller framework consumes little resources. Small resource consumption and overhead are important attributes to enable high scalability. The small resource consumption and overhead seen in the lab prototype makes it reasonable to project high scalability in a production environment with large amounts of resources and consumers per resource. Combined with the suggested PID controller tuning and rearrangement of `tc` filters an even smaller footprint can be achieved.

The measuring point where virtual disk response time is measured must be moved in order to detect bottlenecks that occur before the local disks of the target server. An approach using agents on iSCSI initiators would be the best way of considering all bottlenecks along the data path by providing the initiator-experienced wait time to the throttling bridge. The advantage of this approach is its simplicity, and how efficiently it will capture all bottlenecks along the iSCSI data path. The disadvantage is its reliance on initiator host modifications. A viable approach could be to use the attribute *has_agent_installed* to infer relative higher importance to the set of initiator that has agents, and automatically use the set of consumers not having agents as a first attempt of throttling before resorting to prioritization between initiators with agents installed. Using this approach, the action of installing an agent serves both the purpose of making performance metrics available to the controller and telling about the membership in the set of hosts with the least importance.

Previously developed algorithms other than the PID algorithm can be combined with the throttling techniques from this work to create even more efficient and/or general purpose QoS mechanisms for iSCSI or even other IP/Ethernet based storage technologies. Furthermore, the PID control algorithm could be evaluated as a means to create stability and predictability in other infrastructure components than just iSCSI devices. It is likely that the problem of controlling iSCSI consumers is not the only one where a PID controller can contribute.

There is always a persistent and large interest in workload classification/modeling techniques in various research areas, not only in the storage field. Together with the ever-evolving efforts to model storage devices, this research can be combined with the ideas and results in this paper in order to add improved and even more generalized frameworks. For example, these techniques could be used to elect candidates for the different sets of throttleable consumers in more sophisticated ways. Also, more advanced algorithms could be combined with response time measurements in order to more accurately detect and/or predict if there is a real problem about to occur.

6 Conclusion

Resource sharing is widely used in storage devices for the purpose of flexibility and maximum utilization of the underlying hardware. Sharing resources like this introduces a considerable risk of violating application service level agreements caused by the unpredictable amount of I/O capacity available to individual storage consumers. The difficulties experienced by system administrators in making keepable promise about storage performance and the amount of previous research in the storage QoS field clearly emphasizes the need for practical and real-world-usable QoS mechanisms for storage systems.

iSCSI based storage solutions are capturing increased market share from FC based storage solutions due to increased performance and low cost. Thus, iSCSI is an interesting target technology for devolpment of QoS mechanisms for wide industry and system administrator adoption. The fact that iSCSI utilizes TCP for transportation makes it possible, and very interesting, to adapt well known network traffic shaping tools for the purpose of QoS in iSCSI environments.

This work reproduces and demonstrates the nature of resource sharing, the effect of resource saturation on throughput and consumer response time, and the resulting interference caused by load interaction. Using a Linux based iSCSI storage appliance, experiments reproduce the varying performance of individual consumers caused by other consumers' activity. The lab environment, verified to exhibit similar properties to problematic real-world storage solutions, is then used to design methods to solve some relevant aspects of load interference. The methods involve using a network packet delay method, available in the `netem` module of Linux Traffic Control, in novel ways and a modified proportional integral derivative (PID) controller. By combining the features of the `netem` module with Iptables' ability to dynamically mark packets, an efficient bidirectional mechanism for throttling individual iSCSI initiators consumers is created. The created packet delay throttle is utilized by a modified PID controller implemented in software. The PID controller utilizes the packet delay throttle as a means to influence its input value: the average wait time of the resource being controlled. The resource being controlled in the lab setup is LVM volume groups, but the methods are generally adaptable to any kind of resource exhibiting similar attributes.

The effect of packet delay throttling and the PID controllers' suitability as decision engine is thoroughly examined through experimental results. Finally, all previously designed and tested elements used in single aspect experiments are tied together in a prototype for a autonomous resource control framework that is able to keep resource read response time below a configurable threshold by throttling write activity to the resource automatically. In spite of rapidly varying write workloads, the framework is able to keep a resource read response time below the set threshold. The set of throttleable write consumers is automatically maintained and ready to be used by the PID controller monitoring read response time. The framework spawns a PID controller per resource, using per resource sets of throttleable consumers and per resource response time measurements. The sets of throttleable consumers are automatically populated using simple workload profiling.

This work opens several interesting paths for further research and applications. By using the fundamental ideas explored, it is possible to create QoS modules to be used as an external bridge in front of iSCSI appliances or integrated into Linux based iSCSI appliances similar to the lab environment. Previously developed algorithms can be combined with the throttling techniques from this paper to create even more efficient and/or general purpose QoS mechanisms for iSCSI or even other IP/Ethernet based storage technologies. Furthermore, the PID control algorithm could be evaluated as a means to create stability and predictability in other infrastructure components than just iSCSI devices.

By using the basic building blocks of this work it is possible to create a vast amount of prioritization schemes. The few examples given serves as a demonstration of the inherent opportunities. With the modular design of the different programs it should be trivial to reimplement the framework in similar setups with minor adjustments only.

With the small resource consumption footprint of the prototype, and room for further improvement of it, this concept should scale to enterprise level production environments with large amounts of resources and storage consumers.

By utilizing the ideas from this work, system administrators and vendors can offer QoS for iSCSI storage, thereby making it possible to offer differentiated SLAs to storage consumers supporting application SLAs with a confidence previously very difficult to achieve.

References

[1] Home page of ipset. URL `http://ipset.netfilter.org/`.

[2] A. Gulati and I. Ahmad. Towards distributed storage resource management using flow control. *ACM SIGOPS Operating Systems Review*, 42(6):10–16, 2008.

[3] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. Parda: proportional allocation of resources for distributed storage access. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[4] Ajay Gulati, Chethan Kumar, and Irfan Ahmad. Modeling workloads and devices for io load balancing in virtualized environments. *SIGMETRICS Perform. Eval. Rev.*, 37(3):61–66, 2009. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1710115.1710127.

[5] F. Haugen. *PID control*. Tapir Academic Press, 2004.

[6] Home page of lvm. URL `http://sourceware.org/lvm2/`.

[7] iscsi enterprise target project homepage. URL `http://iscsitarget.sourceforge.net/`.

[8] L. Huang, G. Peng, and T. Chiueh. Multi-dimensional storage virtualization. *ACM SIGMETRICS Performance Evaluation Review*, 32(1):14–24, 2004.

[9] Lan Huang. Stonehenge: A high performance virtualized storage cluster with qos guarantees. Technical report, 2003.

[10] G. Peng. *Availability, fairness, and performance optimization in storage virtualization systems*. PhD thesis, Stony Brook University, 2006.

[11] P. Shenoy and H.M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems*. *Real-Time Systems*, 22 (1):9–48, 2002.

[12] C.R. Lumb, A. Merchant, and G.A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, page 144. USENIX Association, 2003.

[13] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage (TOS)*, 1(4):480, 2005.

[14] M. Wachs, M. Abd-El-Malek, E. Thereska, and G.R. Ganger. Argon: performance insulation for shared storage servers. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 5–5. USENIX Association, 2007.

[15] S. Uttamchandani, L. Yin, G.A. Alvarez, J. Palmer, and G. Agha. CHAMELEON: a self-evolving, fully-adaptive resource arbitrator for storage systems. URL `https://www.usenix.org/events/usenix05/tech/general/full_papers/uttamchandani/uttamchandani_html/paper.html`.

[16] J.C. Wu and S.A. Brandt. The design and implementation of AQuA: an adaptive quality of service aware object-based storage device. In *Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218. Citeseer.

[17] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[18] W. John. Traveling To Rome: A Retrospective On The Journey. *Operating systems review*, 43(1):10–15, 2009.

[19] F. Haugen. *Anvendt reguleringsteknikk*. Tapir, 1992.

A The PID controller

The problem investigated in this paper is similar to a process control problem solved by PID controllers. Figure 11 demonstrates the instant rate reducing throttling effect freeing capacity, which again influences read response time. Section 3.3 describes a stepwise close-to-linear relationship similar to what a PID controller needs in order to work. Figure 18 shows the concept of a PID controller [1].

PID controllers can be implemented in software using a numerical approximation method. This work uses a numerical implementation of the PID controller with virtual disk wait-time as input signal and packet delay as output signal.

The packet delay throttle is implemented as a range of integers representing a stepwise proportional throttling mechanism. Each integer step represents an increased packet delay, thus, a decreased rate. Figures 8 and 9 suggest that steps of $0.5ms$ is a suitable granularity. At $0.5ms$ granularity, the amount of steps is determined from maximum allowed artificial packet delay: i.e. zero

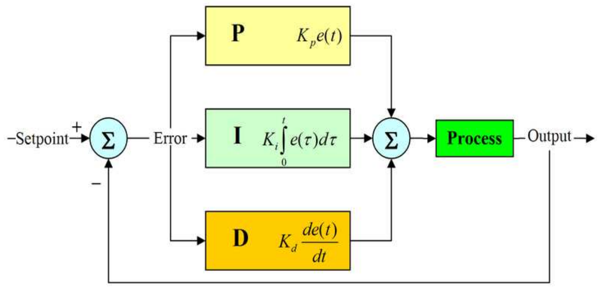


Figure 18: Block diagram of a PID controller. Licensed under the terms of Creative Commons Attribution 2.5 Generic .

rate reduction plus 21 increasing steps of rate reduction with a maximum delay of $20ms$.

$$u(t) = \underbrace{K_p e(t)}_{\text{Proportional}} + \underbrace{\frac{K_p}{T_i}\int_0^t e(\tau)d\tau}_{\text{Integral}} + \underbrace{K_p T_d e'(t)}_{\text{Derivative}} \quad (1)$$

Equation 1 represents the continuous function for outputting throttling amount as a function of the set-point error $e(t)$, the difference between the set value (threshold) and real value. Hence, the PID controller is an error driven controller. The proportional part is the first part of the function and is parameterized by the proportional gain K_p. The second part is the integral part. It is proportional to both the error and the duration of it and is parameterized by the integral time T_i. The purpose of the integrating part is to eliminate the residual steady state error that occurs with a proportional-only controller. The third part of the equation is the differential part. It is parameterized by the derivative gain tuning parameter T_d. The purpose of the derivative part is to slow down the rate of change of the controller output, thereby reducing the magnitude of overshoot created by the integral part.

When computer based controllers replaced older analogue PID controllers, the PID function was discretized using Euler's backward method and became the basic discrete function shown in equation 2 yielding the so-called discrete PID algorithm on incremental form. The function is used as the basis for most discrete PID controllers in the industry [5, 19]. This paper implements a variation of equation 2 that takes the distance above preset response time threshold as input error signal and computes an output throttling value. The modified algorithm is named a single sided PID controller because it only throttles when the error is positive, that is, when the real value is higher than the set threshold.

The PID control algorithm is a direct implementation of equation 2 below with two exceptions: the negative throttling value is capped to the maximum throttling step corresponding to the integer value of the packet delay class with the highest delay, and the positive throttling value capped to zero. This is done to prevent integral windup: the integral part accumulating too high values that takes a long time to wind down again, and to disable throttling completely when the error is negative: real value is below the threshold. The output value of the PID controller is rounded up to the next integer value, and that integer becomes the `Iptables` mark to apply to all outgoing ACK packets matching destination addresses of the iSCSI initiator IP addresses in the set of throttleable consumers.

$$u_k = \underbrace{K_p e_k}_{Proportional} + \underbrace{u_{i_{k-1}} + \frac{K_p T}{Ti} e_k}_{Integral} + \underbrace{\frac{K_p T_d}{T}(e_k - e_{k-1})}_{Derivative} \quad (2)$$

The PID controller must be tuned for optimal control of the process. In control engineering, the best operation of the controller is when the actual value always is stable and equal to the set point no matter how fast the set point changes or environmental forces influence the actual value. This ideal behavior is never achievable in real world applications of the PID controller: there are always physical limitations that makes the ideal case a theoretical utopia. The tuning process' concern is finding the parameters to the controller that makes it behave as close to the theoretical ideal as possible. There are several known methods to tune PID controllers. The Ziegler-Nichols method, the improved Åstrøm-Hägglund method and the Skogestad method are some widely used methods in control engineering [5]. These methods have not been considered during this paper since a few iterative experiments and according parameter adjustments yielded stable and good controller performance in short time. Thus, the process in this paper is easy to tune compared to many industrial processes. However, thorough tuning efforts is likely to produce similar controller efficiency with less less resource usage of the controller loop.

In addition to the PID parameters, the sample interval influences loop stability and tuning. Generally, the discrete PID controller approaches the behavior of a continuous PID controller when the sample interval goes to zero. The reason to keep sample interval short is increased stability and the reason for increasing the sample interval is minimizing resources utilized by the controller. The sample interval used in this paper was found by experimenting with values and observing CPU usage. A sample interval of $100ms$ yielded very stable controller behavior and CPU utilization of approximately 1%.

However, lowering the sample frequency more may be possible without sacrificing stability. Another benefit of lowering the sampling frequency is calmer operation of the throttle. It may not be necessary to move the throttled IP addresses around as agilely as in the experiments, but it must be agile enough to capture rapid workload interference changes. The interval of $100ms$ seems to be a fair balance between controller resource consumption and agility.

Notes

[1]Created by the Silverstar user @ Wikipedia, as required by CC licensing terms.

YAF: Yet Another Flowmeter

Christopher M. Inacio
CERT
Software Engineering Institute
Carnegie Mellon University
inacio@cert.org

Brian Trammell
Communication Systems Group
ETH Zurich
trammell@tik.ee.ethz.ch

Abstract

A flow meter generates flow data - which contains information about each connection observed on a network - from a stream of observed packets. Flow meters can be implemented in standalone measurement devices or inline on packet forwarding devices, such as routers. YAF (Yet Another Flowmeter) was created as a reference implementation of an IPFIX Metering and Exporting Process, and to provide a platform for experimentation and rapid deployment of new flow meter capabilities. Significant engineering effort has also gone into ensuring that YAF is a high performance, flexible, stable, and capable flow collector. This paper describes the some of the issues we encountered in designing and implementing YAF, along with some background on some of the technologies that we chose for implementation. In addition we will describe some of our experiences in deploying and operating YAF in large-scale networks.

1 Introduction

Network traffic continues to grow at an exponential rate, with global internet traffic forecast to increase 34% year-on-year though the first half of this decade [5]. Understanding the uses of the network and the needs of its users is necessary for both operations and planning, for both business and technical reasons. The need for network monitoring has therefore never been greater in today's large-scale networks. While various tools exist to aid in this problem, network flow data represents the most comprehensive way to get an in-depth understanding of network activity while still leveraging a huge amount of data reduction necessary in order to practically analyze large-scale network traffic.

The CERT Network Situational Awareness (NetSA) Group had previously developed the System for Internet Level Knowledge (SiLK) [12] in order to address the analysis issues in this area. The SiLK tools are designed to support the understanding of network flow information for both network traffic and engineering, as well as security. SiLK provides a set of command-line tools modeled after the standard UNIX command-line tools to analyze the collected data. A typical SiLK workflow consists of a query to retrieve information from a SiLK data repository, which is then piped into a set of SiLK tools to further process the results. The data record format for SiLK is proprietary format, but the data fields are fundamentally similar to the NetFlow v5 record, as SiLK was originally designed to process NetFlow v5 data.

However, this approach left us at the mercy of existing flow meters, such as those deployed on forwarding devices, to generate the flow data on which SiLK operates. Existing solutions had various issues. Flow meters on forwarding devices often lose flows, because high-fidelity flow generation is rightly a lower priority for these devices than forwarding packets. Flow meters using unreliable transport for export also suffer from flow loss, especially during times of high traffic load. In addition, at the time no openly available flow meter had support for the then-emerging IPFIX [6] standard.

YAF (Yet Another Flowmeter) was designed to address this situation. We set out to build a standards-conformant, high-performance, bidirectional network flow meter. Standards-conformance was important to ensure a long operational lifecycle and wide interoperability. We selected the IPFIX standard, based on Cisco NetFlow V9, the successor to the successful de facto standard Cisco NetFlow V5 export protocol. The authors actively participated in the standards process within the IETF to feed our experiences in building and deploying YAF into improving the standard itself, and continue to do so.

Performance was of utmost concern given the scale of the networks we needed to monitor, and the ever-increasing link speeds of the Internet backbone and large enterprise borders. Bidirectionality was important to enable analysis on both sides of a communication, as well as to slightly increase export efficiency by eliminating redundant information.

The result of this effort is a software tool, `yaf`, which captures live packets or reads packet trace files, and exports IPFIX flows to a collector or to an IPFIX file [25]. It exports IPFIX bidirectional flows [24], and optionally supports a set of additional Information Elements for additional information derived from packet-level or packet payload information, such as TCP initial sequence numbers or payload Shannon entropy.

YAF, in itself, is not a network analysis application or an intrusion detection system. Instead, it is intended as a stage in a comprehensive flow-based measurement infrastructure, with a focus on security-relevant applications.

The rest of this paper is organized as follows. Section 2 describes network flow data, and the various protocols in use for exporting flows, especially IPFIX, and especially as used by YAF. From there we explore the details of the design of YAF in detail in section 3, focusing on those choices which make YAF unique. Related work is described in section 4. We then describe a few existing applications of YAF in section 5, including its application with SiLK [12] within the NetSA Security Suite and its use in the middle tier of PRISM [11], a multi-stage privacy-preserving network monitoring architecture.

2 Network Flow Data: Properties and Protocols

YAF exports flow data. A flow, simply stated, represents a connection between two sockets. More generally and formally, a flow is "a *set of packets* passing an observation point in the network *during a certain time interval* sharing *a set of common properties*, each of which is the result of applying a function to packet, transport, or application header fields; characteristics of the packet itself; or information about the packets treatment." [6]. Specific flow export methods and protocols may use more restrictive definitions than this, for example, by constraining the set of common properties (the flow key) or the method for selecting time intervals. Flows may be unidirectional, in which case they represent one direction of a socket connection, or bidirectional, in which case they represent both directions, or the entire interaction.

The time interval defining a flow generally spans from the first observed packet of the flow to one of three events: either the natural end of the flow, the idle timeout of the flow, or the active timeout of the flow. The natural end of the flow is determined by observing and maintaining the state of the flow for connection-oriented protocols such as TCP or SCTP. The natural end can be determined exhaustively, completely modeling the state machine for the transport layer protocol, or approximately, e.g. by counting a flow as every packet between the first SYN and the first FIN or RST observed for TCP.

The idle timeout of the flow is the longest period of time between packets after which the flow will be considered idle; this is the natural way to expire flows in non-connection-oriented protocols such as UDP. Idle timeouts are generally configurable, and lead to a measurement tradeoff: a short idle timeout leads to faster reaction and lower state utilization during flow metering at the expense of risking expiring flows prematurely.

The active timeout of the flow is the longest lifetime a flow is allowed to have; any flow longer after the idle timeout will be exported, and subsequent packets accounted to a new flow. This is a final backstop against growth of the flow table.

The exact relationship between idle and active timeout and export time is implementation-specific. For example, active timeout can be implemented as a continuous or periodic process; the latter approach leads to some variation in the actual active timeout in the exported data.

The following few sections describes the origin of the IPFIX flow protocol. The discussion is organized from a historical perspective in chronological order.

2.1 Cisco NetFlow v5

Defined by Cisco, NetFlow v5 [4] is a widely deployed de facto standard protocol and raw storage representation for network flow data. It is based on a fixed-length binary record format, with a fixed set of fields. This implies support only for export of IPv4 flows and 16-bit autonomous system numbers, which has led to its being superceded in recent years by NetFlow v9 (see section 2.2), but existing repositories of flow data as well as long replacement cycles of routers which support NetFlow v5 ensure this protocol and representation will be around for some time.

NetFlow v5 is a unidirectional protocol, with the flow meter sending packets via UDP to the collector. It is a "fire-and-forget" protocol; there is no provision for upstream control messages or error reporting, other than that provided by UDP itself via ICMP. This design choice was made to minimize resource usage and state requirements on the flow meter, which in NetFlow v5 is assumed to be a router.

A NetFlow v5 data stream is made up of packets, each of which has a header followed by a number of records. NetFlow v5 records contain start and end timestamps in terms of the reporting line card's uptime in milliseconds, source and destination IPv4 address, source and destination port, protocol, type-of-service, union of all TCP flags in the flow, input and output interface, source and destination autonomous system number, and source and destination prefix mask length.

The packet header contains the system uptime in milliseconds at export, as well as the system realtime clock at export with nanosecond resolution, which allows flow timestamps to be expressed in millisecond resolution. It also contains a sequence number, which is used to detect dropped NetFlow v5 records.

2.2 Cisco NetFlow v9

Cisco NetFlow v9 [7] is the successor to NetFlow v5, deployed to support IPv6 as well as flexible definition of new record types. It abandons the fixed record format for a template-based system wherein the record format is defined inline. As NetFlow v9 was the base protocol from which IPFIX was developed, the mechanisms it uses are essentially the same as those in IPFIX, though some terminology may be different; therefore, the details of this approach will be elaborated in the following section.

While its flexible data definition makes it nonsensical to speak of a NetFlow v9 record format, and the data exported by Cisco's implementation of NetFlow v9 is administrator-configurable, the information commonly provided in a NetFlow v9 record is more or less equivalent to that available in NetFlow v5.

2.3 IPFIX

IPFIX is a template-based, record-oriented, binary export format. The basic unit of data transfer in IPFIX is the *message*. A message contains a header and one or more *sets*, which contain *records*. A set may be either a template set, containing templates; or a data set, containing data records. A data set references the template describing the data records within that set. This is the mechanism which lends IPFIX its flexibility.

Within the message, each set has a 16-bit ID in its set header. This identifies whether the set contains templates, or data records. In the latter case, the data set ID matches the template ID of the template which describes the records in that data set. A template is then essentially an ordered list of information elements identified by a template ID. An information element (often abbreviated IE) represents a named data field of a specific data type. The data types supported by IPFIX cover the standard primitive types (e.g. unsigned32, boolean) plus additional types for addresses and timestamps; each data type defines an encoding. IEs are then instances of these types, each with its own specific meaning.

IPFIX provides a registry of information elements, administered by IANA [14], that cover most common network measurement applications. This was initially defined in RFC 5102 [18], and is extended both by subsequent IPFIX RFCs as well as by a community process with expert review. Information elements may also be scoped to SMI Private Enterprise Numbers; these can be used to export information (as by YAF) not suitable for standardization through the IANA process.

Because Templates are generally exported once per session, the cost of self-representation is amortized over many records. In this way, IPFIX can support a wide variety of record formats, avoiding tying the implementation of a flow meter to a specific export data structure, without the overhead of other representations with semantic flexibility per record, e.g. XML. This extensibility allows innovation in flow metering and export, and as such was the natural choice for YAF.

2.3.1 As exported by YAF

As shown in 2, YAF can export an extensive set of fields, a superset of those available in earlier NetFlow versions, omitting those specific to packet-forwarding devices. Many of these are IPFIX-standard fields defined in the IANA registry, while others (those with an annotation in the "YAF-specific" column) are enterprise-specific Information Elements defined specifically for YAF.

YAF also takes extensive advantage of IPFIX's template mechanism to enable efficient export, as detailed in section 3.4. As shown in the "Present when" column in table 2, YAF exports IPv4 addresses only when the flow is an IPv4 flow, and IPv6 addresses only when the flow is an IPv6 flow. Reverse information elements are only exported for flows which actually have packets in the reverse direction. In addition, command-line arguments enabling various additional features of YAF at runtime (e.g. DPI, entropy calculation, and others to be described later in this work) cause YAF to capture that data and add information elements to its export templates to represent them. Each exported record contains only the information elements it needs, with YAF selecting the appropriate template at runtime, exporting it if it has not yet been exported, and starting the export of a new Data Set if necessary.

3 Detailed Design of YAF

YAF is designed as a bidirectional network flow meter. At its core, it takes packet data from some source, decodes it, and associates the packet data with a flow. When flows are determined to be complete, it exports them. This is a rather simplified view, to which we will add some more detail in the following subsections.

First we follow a packet through the various stages of the basic YAF workflow shown in 1, from capture through to export. Then we examine other interesting

Name	Present when	YAF-specific
flowStartMilliseconds	always	
flowEndMilliseconds	always	
octetTotalCount	always	*may use reduced-length encoding*
reverseOctetTotalCount	biflow	*may use reduced-length encoding*
packetTotalCount	always	*may use reduced-length encoding*
reversePacketTotalCount	biflow	*may use reduced-length encoding*
sourceIPv6Address	IPv6	
destinationIPv6Address	IPv6	
sourceIPv4Address	IPv4	
destinationIPv4Address	IPv4	
sourceTransportPort	always	*may contain ICMP type/code*
destinationTransportPort	always	
protocolIdentifier	always	
flowEndReason	always	*may contain SiLK-specific flags*
silkAppLabel	–applabel	DPI application label
payloadEntropy	–entropy	Shannon payload entropy
reversePayloadEntropy	biflow –entropy	Shannon reverse payload entropy
mlAppLabel	–mlapplabel	Machine-learning app label
reverseFlowDeltaMilliseconds	biflow	RTT of initial handshake
tcpSequenceNumber	TCP	
reverseTcpSequenceNumber	TCP biflow	
initialTCPFlags	TCP	TCP flags of first packet
unionTCPFlags	TCP	TCP flags of $2..n^{th}$ packet
reverseInitialTCPFlags	TCP biflow	TCP flags of first reverse packet
reverseUnionTCPFlags	TCP biflow	TCP flags of $2..n^{th}$ reverse packet
vlanId	–mac	
reverseVlanId	–mac	
ingressInterface	–live dag multi-IF	
egressInterface	–live dag multi-IF	
osName	–p0fprint	p0f Operating System name
osVersion	–p0fprint	p0f Operating System version
reverseOsName	biflow –p0fprint	p0f reverse Operating System name
reverseOsVersion	biflow –p0fprint	p0f reverse Operating System version
firstPacketBanner	–fpexport	First forward packet IP payload
reverseFirstPacketBanner	biflow –fpexport	First reverse packet IP payload
secondPacketBanner	–fpexport	Second forward packet IP payload
payload	–export-payload	First n bytes of application payload
reversePayload	biflow –export-payload	First n bytes of reverse application payload

Table 2: Information elements in a YAF record

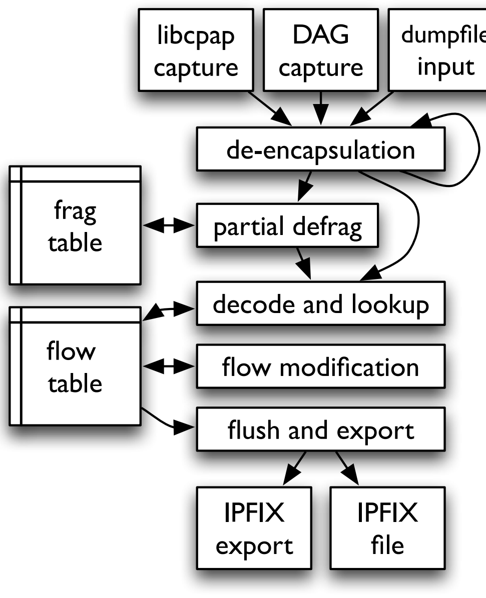


Figure 1: Basic Data Flow in YAF

aspects of YAF's design, and additional optional features it supports compared to other flow meters.

3.1 Recursive De-encapsulation

Packet data input can come from a variety of sources, including `libpcap` dumpfiles, live capture on commodity interfaces via `libpcap` as well as specialized devices with `libpcap`-compatible interfaces such as Bivio and Napatech devices, and Endace DAG cards. Each of these sources generally yields Layer 2 and above information; YAF then recursively unwraps encapsulations to arrive at an IP header, possibly storing certain information (e.g., VLAN tags or MAC addresses) for later export with the flow. In addition to the ubiquitous Ethernet encapsulation, YAF also supports a variety of less common, carrier-use encapsulations. YAF can decode GRE, MPLS, MPLE, PPPoE, cHDLC, Linux SLL, PPP, and PCAP raw. Running on appropriate hardware, this allows YAF to decode network information from Ethernet to DS3 links, to OC-192 connections. Additionally, as depicted in diagram 1, YAF can also decode odd combinations of encapsulation by running through the encapsulation phase multiple times. For example, one site running YAF encapsulates Ethernet over MPLS. Additionally, YAF is constructed to allow new encapsulations to be cleanly added. The decoding system requires only minimal modification to support a new encoding. YAF relies on the capture system to be able to identify the base encapsulation.

3.2 Decoding

De-encapsulated packets are passed to the Layer 3 and 4 decoding layer, which extracts flow keys and counters from the packet data. The flow key determines which flow the packet belongs to, and in YAF consists of the traditional "5-tuple" (source and destination address, source and destination port, protocol) as well as the IP version number (4 or 6) if YAF is compiled for dual-stack support. The flow key may also optionally include the VLAN tag and, in the case of a DAG card as source, the DAG interface number on which the packet was captured. This flow key is used for lookup in the flow table.

3.3 The Flow Table

The YAF flow table is implemented as a hashtable-indexed pickable queue. This data structure is essentially a queue paired with a hashtable. It allows random access to any entry in the flow table via the hashtable, but also constant access to the least-recently-seen entries, which allows efficient timeout of idle flows. This design evolved in part from the bin queue used in NAF [26].

The flow key calculated from the decoding stage is looked up in the flow table's hashtable. If no active flow record corresponding to the flow key is found, a new record is created. Regardless, the flow record is modified with information from the packet (e.g., counters, payload and payload-derived information), and moved to the head of the flow table's pickable queue to implement idle timeout. Active timeout is evaluated when each flow is selected: if a packet belongs to a flow that is older than the active timeout interval, that flow is removed from the flow table and exported, and a new flow record is created for the incoming packet.

From this point on, the YAF data flow operates on flows only.

Since YAF flow records in memory are all equal size, and they have variable lifetimes, they are allocated using a slab allocator [3], which allows fast reuse of expired flow records. This gives YAF additional performance over true dynamic allocation, but still allows the flow table to grow and shrink with variable traffic load unlike with a statically-allocated table. However, since the slab allocator never returns memory to the operating system, its memory footprint will generally not be reduced during low-traffic periods. Growth of the flow table can be controlled by command-line options setting the idle and active timeouts as well as the target maximum table size, which dynamically reduces the timeouts in order to prevent resource exhaustion during traffic bursts or intentional denial-of-service attacks against the flow meter.

During the long transition from IPv4 to IPv6, a sufficiently large and complex organization may use both protocols for some time; therefore, a design goal of YAF is to be able to support measurement of both IPv4 and IPv6 traffic efficiently, with efficient runtime storage and export of both IPv4 and IPv6 flows from the same interface.

If YAF is compiled with IPv6 support, it will dynamically create either an IPv4 or IPv6 flow table entry based upon the protocol of the flow. IPv4 and IPv6 flows are defined as overlaid C structures, so most of the YAF code for handling them that does not handle flow table entry allocation or endpoint addresses treats the two flow types equally. From the slab allocator's point of view, this is like having two separate flow tables, but both IPv4 and IPv6 flows are unified in the same pickable queue. This feature comes at the cost of some additional memory to store the overlaid structure and some delay in selecting flow type at flow creation compared to an IPv4-only YAF, but much less memory than would be required if IPv4 and IPv6 flows were stored in a single `union` data type with enough space for the larger addresses. Efficient template selection as in section 3.4 below minimizes export bandwidth penalty for dual-stack support.

3.4 Efficient Export and Template Selection

When a flow ends, whether through natural completion (presently supported only through the TCP FIN handshake) or idle or active timeout, it is ready for export.

Though YAF exports what is semantically one type of data, it uses multiple IPFIX templates to maximize export efficiency. As mentioned in section 2.3, IPFIX Templates are identified by a 16-bit number. YAF essentially uses some of these bits as flags in order to enable or disable fields in the Template used to export each flow, based on the flow's characteristics. The characteristics used for template selection are:

- whether the flow requires full-length (64-bit) counter export, or can be represented with 32-bit packet and octet counters (reduced-length encoding)
- whether the flow is an IPv4 or IPv6 flow
- whether the flow is a biflow (it has at least one packet in the reverse direction)
- whether the flow is a TCP flow
- whether layer 2 (MAC and VLAN) export is enabled
- whether the flow was captured on a DAG card, and has DAG interface information
- whether the flow has an application label
- whether the flow has entropy information
- whether the flow has a p0f fingerprint
- whether the flow has payload, and payload export is enabled

Compare these characteristics with the record structure in table 2.

When a record is ready to be exported, YAF selects a template by deriving a template ID from the properties of the flow table entry and the configuration of the YAF instance. If this template ID corresponds to a template that has not yet been exported, it exports the template; if it doesn't match the that of the last exported record, it starts exporting a new IPFIX set.

For example, a short IPv4 UDP flow with no reverse direction will be exported using a template containing IPv4 address elements, no reverse elements, no TCP elements, and reduced length counters.

When a flow is exported, YAF forgets about it, and its entry is recycled by the slab allocator.

This concludes our trip through the "normal" YAF workflow; subsequent subsections handle YAF's design approach to particular caveats of flow metering, or optional features supported by YAF.

3.5 Just Enough Defragmentation

IP packet fragmentation causes a problem for flow metering. Some implementations of flow assembly, especially those on resource-constrained devices or on devices where flow metering is a secondary, lower-priority function (e.g., routers), simply ignore fragmentation. For TCP or UDP, these would treat the first four bytes below the IP layer as the source and destination port of the flow regardless of whether the packet contained the first fragment; all fragments other than the first per packet are accounted to the wrong flows. While this may be acceptable for some applications, given the relatively low prevalence of fragmented traffic on the Internet [21], it presents a simple attack against any flow meter ignoring fragmentation: most packets can be accounted to the wrong flows simply by aggressive fragmentation.

At the same time, full fragment reassembly is resource-intensive, especially when most of the information stored and reassembled will then be discarded, as is the case with flow key extraction from fragments.

For this reason, YAF supports just enough defragmentation: a fragment table designed very much like the flow table (i.e., using pickable queues, slab allocation, and dynamic timeouts for defense against resource overuse) which keeps track of the flow a fragment belongs to, and defragments only enough payload per flow to support the other features selected at runtime. Defragmentation occurs between de-encapsulation and decoding.

Defragmentation is enabled by default, but can be disabled at runtime to save resources (e.g., on networks where fragmented packets are known not to be present) or for compatibility with flow meters not supporting defragmentation.

3.6 Packet Clock

YAF is designed to accept data both from live capture as well as from files containing ordered packet traces. For that reason, designed into the core of YAF is the concept of the packet clock: YAF in effect "pretends" that the current time is the timestamp of the packet it is presently processing. This implies that timeouts are evaluated against the data, and not against the system realtime clock of the machine running YAF. This is important to ensure repeatability: that the same packet trace processed multiple times will lead to identical output data, as well as that YAF will produce identical data whether from live capture or from playback.

3.7 Per-flow Payload Capture

If so configured at compile-time, YAF supports per-flow payload capture. Payload capture is limited to the first n bytes of each flow, configured on the command-line, in order to provide the administrator control over YAF's resource consumption; payload capture can significantly increase YAF's requirements. Payload capture for TCP flows provides full TCP reassembly.

Since each YAF record must fit within an IPFIX record, and IPFIX imposes as 65515-byte content limit on records, this maximum exportable payload is somewhat under 64kB for uniflows and somewhat under 32kB for biflows. As well as supporting direct export of flow payload, YAF payload capture can be used to support entropy calculation and application identification, as described in the following subsections.

3.8 Flow Payload Entropy

YAF can calculate the Shannon [22] entropy of the captured payload for use in understanding the nature of the traffic within a flow. The Shannon entropy is calculated by scanning through the first n bytes of captured payload (the "banner") byte-by-byte and creating a histogram distribution within a 256-entry array. The partial entropy of each histogram value x is then summed to compute the total entropy H as follows:

$$H = \sum_{i=0}^{255} \frac{x_i}{n} * \frac{\log x_i}{\log 2.0}$$

The entropy is scaled to the range 0-255, for single byte export, as follows:

$$H_{export} = -1 * \frac{H}{8.0} * 256$$

The same operation is done for the reverse payload to generate the reverse entropy.

This method of entropy calculation results in a measure of entropy in bits per byte (log base 2); i.e., a number between 0 and 8 in 8-bit fixed-point representation. In pure terms, a value of 255 would indicate a *perfectly* random set of data, while a value close to 0 would indicate an extremely redundant set of data with almost no information content. High entropy values indicate data that is either compressed or encrypted. Lower values likely to indicate something such as an ASCII-based protocol, or English text.

As a guide to the actual usage of the numbers from YAF, values above approximately 230 indicate compressed or encrypted traffic. Values centered around 140 are likely to be English text. A quirk in SSL/TLS is that it commonly zeros its packets out before sending them. This leads to extremely low numbers often indicating an SSL/TLS encrypted flow.

3.9 Application Labeling

YAF can analyze the banner on each flow that it captures in order to recognize the protocols above layer 4 in captured flows, and to label each flow with the application it is running. Labeling runs at flush-and-export time (i.e., once the flow payload is known to be complete). YAF independently evaluates each direction of a biflow during labeling. Application labeling is designed to be transport port neutral in recognizing the protocol being used; however, port information is used as a hint to which protocol match to attempt first, for efficiency purposes. For example, if YAF has captured a biflow from two hosts with ports 5238 and 80, YAF will attempt to match HTTP first, due to the presence of port 80 in the flow. Application labeling is first match wins; in the example above, the flow would be labeled HTTP and labeling would stop.

YAF application labeling currently recognizes the following protocols: HTTP, SSH, SMTP, Gnutella, Yahoo Messenger, DNS, FTP, SSL/TLS, SLP, IMAP, IRC, RTSP, SIP, RSync, PPTP, NNTP, TFTP, MySQL, and POP3. Additional protocols are actively being added. Areas of future work include an experimental integration of the OpenDPI project [16], including the capabilities of that engine as a plug-in. Additionally, it is possible for users to add some recognizers for some simple protocols simple by text editing a configuration file, which is described below.

3.9.1 Implementation Details

YAF uses multiple mechanisms for protocol recognition. For some network protocols, e.g. DNS, YAF includes a shared library written in C which can decode the structure of the DNS packets. The DNS protocol uses a well-formatted binary structure making the tests for determining a valid DNS packet relatively easy in C. Additionally, when exporting the extended packet details as in 3.9.2, the C library has the added advantages of handling the binary fields easily. In addition for cases like DNS implementing name "decompression" is also relatively easy.

Conversely, other protocols, such as SMTP, IMAP, and SIP, are ASCII or UTF-8 based protocols; these lend themselves instead to textual analysis based on regular expressions. YAF provides a method for defining labeling of these protocols based on the widely used PCRE engine [13]. This provides two distinct mechanisms for protocol recognition within YAF. This creates an advantage of allowing application labeling to be applied using two different mechanisms for the two different types of protocols. Another advantage of using a regular expression system is that it allows easy in-the-field experimentation, without recompiling anything, to find new protocols.

In order to illustrate how the recognition system is configured, a small sample of the application labeler configuration file is shown:

```
# HTTP
label 80 regex HTTP/\d\.\d\b
# SSH
label 22 regex ^SSH-\d\.\d
# SMTP
label 25 regex (?i)^(HE|EH)LO\b
# DNS
label 53 plugin dnsplugin \
         dnsplugin_LTX_ycDnsScanScan
```

The structure of the entries is keyword `label` followed by the port number. This port number is used as both the label that YAF will put into the record it produces as the output record as well as the port number for hinting based on the source and destination ports. The next keyword is either `regex` for a regular expression based rule or `plugin` for a C-callable plugin. In the case of the `regex` expression, everything beyond the `regex` keyword will be interpreted as the regular expression.

The C plugin requires a set of functions to be defined and a standard naming convention to be used. There are 11 functions to be defined required of every plugin, and optionally, a twelfth function used for the deep packet inspection. These functions are defined in the source file `yafhooks.c`. Advanced users of YAF may also be capable of implementing a YAF extension in this way.

As previously mentioned the label which YAF applies is defined as the primary port number on which the application is expected to be seen. This is also used at runtime to determine the most likely matching application for a given flow, based on the source and destination ports of the flow. If there is no match, the rules are evaluated in order, which allows performance tuning by ordering the rules in the order of their expected precedence on the network. However, regular expression-based flow labeling still presents a performance risk, and users should take care that regular expressions are kept simple in order to minimize negative impact on performance.

3.9.2 Extended Application Information Export

YAF can export extended information about a large number of protocols in its application labeling capability. Many of these fields are relatively innocuous and detail the general workings of the network and its protocols. However, some fields may contain sensitive information. Depending on jurisdiction, captured payload data or data derived from payload capture may be considered Personally Identifiable Information (PII), such that turning on these features may require special handling of the flow record output.

As previously mentioned, the recently released YAF 1.2 can identify via deep packet inspection (DPI) the following protocols: HTTP, SSH, SMTP, Gnutella, Yahoo Messenger, DNS, FTP, SSL/TLS, SLP, IMAP, IRC, RTSP, SIP, RSYNC, PPTP, NNTP, TFTP, Teredo, MySQL, and POP3. In addition to identifying those protocols, YAF may (or may not) export extended information depending on the various protocols. YAF will currently export extended information about the following protocols: HTTP, SSH, SMTP, FTP, IMAP, RTSP, SIP, and DNS. Futhermore, we collect user name type information in SMTP, FTP, IMAP, and SIP.

As an example of the extended information collected, we will consider a single protocol, SIP [19]. For SIP messages, YAF will optionally capture, identify, and export the following fields: the `Via`, `Max-Forwards`, `To`, `From`, `Contact`, and `Content-Length` headers, as well as the SIP method.

3.10 Performance

YAF was designed and developed with the goal of building a high-performance flow meter, while still maintaining a cleanliness of design allowing future maintainability and extensibility. Performance is measured throughout development and maintenance using profile-based measurement tools such as Shark and Instruments on the Mac OS X development platform.

YAF's performance depends as well upon the performance of the underlying fixbuf IPFIX library, since IPFIX transcoding on export is a significant portion of the work YAF does. fixbuf is also designed to be a high-performance IPFIX implementation, operating on in-memory buffers containing messages and records, and exploiting natural alignment of data structures in order to speed the rearranging and copying from application internal data structures to IPFIX records on the wire.

Subject to the capabilities of the underlying capture device, YAF is tested for performance using various carrier line speeds and encapsulations during live capture. It is tested on Ethernet systems from 100Mbit to 10Gbit, and on optical carrier lines from OC-3 to OC-192, using generated traffic from a dedicated load generator.

YAF is designed to perform well on generic PC hardware running most UNIX variants as well as Apple OS X. For many types of links, a capable PC will be sufficient. YAF is also tested and tuned to run on various custom capture systems including Endace DAG capture cards and Bivio appliances. A future release of YAF will also be developed with specific enhancements for Napatech capture cards.

4 Comparing YAF to Other Flowmeters

The name Yet Another Flowmeter plays on the old UNIX-community joke of prefixing the nth instance of a particular type of tool "Yet Another x"; however, we designed YAF to meet a combination of requirements that were at the time not generally available: we needed high performance, easy extensibility of both the output record format and the flow-level measurement capabilities, and compliance to an emerging standard to ensure a long operational lifetime. Here we compare YAF to existing flow meters, or flow-meter-like systems.

4.1 Software NetFlow Meters/Exporters

While there is a wide variety of both free and commercial software designed to operate as NetFlow collectors and analyzers, there is a smaller number of available NetFlow meter/exporters, which generate flow data from packet data and export via NetFlow or IPFIX. Two popular examples are softflowd [15] and nProbe [8].

`softflowd` does semi-stateful assembly of flows and export via NetFlow v5 and v9; a related tool `pfflowd` uses the OpenBSD packetfilter flow table instead. It supports raw capture from `libpcap` only. It was designed to be fast and simple, and as such supports none of YAF's flexibility or advanced features. Development appears to have been inactive since 2006.

nProbe is an "all-in-one" tool for handling flow data as part of the nTop [9] network measurement suite. One of the features it provides is flow generation and export from packet data. Its feature set is much more comparable to YAF's: it supports IPFIX export, high-speed collection from dedicated capture cards such as Endace DAG and Napatech devices, and protocol inspection . In addition, it does a few things YAF doesn't: operating as an IPFIX Mediator to translate older NetFlow versions to IPFIX, and storing flow data directly into MySQL or sqlite databases, for example. Development is active as of summer 2010.

YAF and nProbe have to some extent been developed in parallel; features showed up in one or another at roughly the same time, and the authors indeed tested the interoperability the underlying IPFIX export implementation as early as 2006. However, in contrast to YAF, while nProbe is published under the GNU GPL, it is not generally freely available, with source download behind a donation paywall and limited mirroring of older versions.

4.2 Argus

Argus [17] is a flow meter and analysis toolkit in a single set of tools. While Argus does contain a set of powerful analysis tools, similar in some ways to SiLK, that is beyond the scope of this paper. Instead, here we focus on its flow measurement and export protocol type.

Argus is designed to measure bidirectional flows in the network control plane. In order to complete that task, Argus will attempt to merge relevant control plane information into a control flow via its monitoring points independent of the link types monitored. For example, if the goal is to monitor a high-performance cluster system using Infiniband, Argus can monitor the control plane on the cluster as well as the external link running an Ethernet or optical carrier connection. In addition to monitoring both of those links, Argus will attempt to match a DNS query on the external link with activity on the Infiniband connection.

Argus is unique in its fundamentally "philosophically" different approach to flow metering. While Argus attempts to merge and relate flow information at the sensor from related flows, YAF and most other flow sensors attempt to capture the flow information in an IPFIX standard way and allow the back end analysis tools, such as SiLK or others, find the relationships among the various flows.

Argus also has a proprietary "sensor-centric" export protocol in contrast to IPFIX and its predecessors, and provides flexibility at the record level as opposed to the informaiton element level. Argus "clients" (collectors) initiate connections to the sensor and pull flow data off the sensor.

Each approach has its pros and cons. In our applications, flow meters are generally deployed in environments where inbound connections are forbidden, and where software maintenance is often difficult; meters must be stable and not change very often; therefore, we prefer to centralize the harder correlation work, and any analysis which may see further innovation, while keeping the edge fast and stable, and the bandwidth from the edge to the data center as small as practical.

4.3 sflow

sflow [23] is a protocol which takes a different approach to the same problems solved by flow collection and analysis. The key design decision here is that attempting full flow assembly on high rate packet data requires too many resources, and for many applications (e.g., traffic matrix generation) sampled packet data is sufficient. Despite the "flow" in the name, sFlow is not a flow metering or export technology: it simulates flows with packet sampling. There is a free reference implementation, as well as support in nTop in addition to switches and routers from a variety of manufacturers [20].

4.4 Netflow, Flexible NetFlow, and other router-based approaches

The key difference between YAF and flow metering processes running on routers is one of application: NetFlow and related technologies run as secondary processes on routers, and as such an important design consideration is that packet forwarding performance must not be impeded by flow metering. This leads to reduced data fidelity during peak traffic times, as the router allocates its limited resources to forwarding instead of monitoring. When high data fidelity in flow metering is a primary requirement, such as for security, flow meters such as YAF can be used on a switch span port or optical tap to offload the task from the router.

A key difference between YAF and Flexible NetFlow is that, although both utilize the template functionality in IPFIX or NetFlow V9, YAF uses it only for export efficiency while Flexible NetFlow uses it for flow key flexibility: it exports different record types, aggregating packets into flow records on other than the traditional flow key. In this way it is more akin to YAF followed by aggregation operations in SiLK, or NAF [26].

5 Applying YAF

YAF was initially released in 2006, although it was marked as an alpha-quality release for quite some time. The YAF 0.7 release of August 2007 market the first release ready for operational deployment. Since then, YAF has been adopted by several organizations as their main software flow meter platform, and has been in production use for quite some time. YAF is still used in the experimental side of network flow analysis as well. In the following two sections we will describe our experiences using YAF for those various purposes.

5.1 YAF and the Security Suite

SiLK [12] is designed to allow very large scale collection and analysis of network flow data. It provides a set of command-line tools modeled after the standard UNIX command-line tools (e.g. sort(1), uniq(1), cut(1)) to analyze the collected data. A typical SiLK command-line to query a SiLK data repository is then piped into another set of SiLK tools to further process the results. The data record format for SiLK is a proprietary format, but the data fields are fundamentally similar to the NetFlow v5 record, as SiLK was originally designed to process NetFlow v5 data.

Large SiLK deployments include one with more than 50 geographically distributed sensors monitoring everything from DS3 to 10 gigabit Ethernet links. In this case, YAF is run on one of two types of sensors: one a Linux-based PC server with an Endace DAG card installed, for monitoring a DS3 link; the other a Bivio 7500 series appliance using multiple blades to listen to a 10 Gigabit Ethernet link.

Each deployed sensor sends data back to a centralized data center via secure, encrypted connection. At that data center, each record from the sensors is collected, tagged, and stored onto a large SAN system for analysis by various analysis groups. This system collects many tens of gigabytes of flow data per day, allowing analysts to see the large-scale picture of network activity occurring across the largest of enterprise networks.

A typical configuration for a PC with an Endace card installed is to have YAF start at system boot. For this purpose, YAF includes a set of startup scripts that can be installed on a typical Linux system to manage YAF. These scripts start YAF via the included `airdaemon` utility, which ensures that YAF will restart on abnormal shutdown due to hardware issues. YAF then typically exports via IPFIX to a local instance of the `rwflowpack` process, part of the SiLK packing system. `rwflowpack` then packs the received IPFIX record into a SiLK format, and then compresses the records to make them smaller still. After the records are fully compressed to an average of about 15 bytes per flow, `rwflowpack` passes the records to `rwsender` for transmission back to the data center.

YAF's enhanced flow metering has been applied in production in combination with SiLK in order to solve operational problem. As a simple example, flows containing traffic running on nonstandard ports can be detected simply by enabling application labeling, then running an `rwfilter` query to compare the application label (exported in the `silkAppLabel` information element) with the ports in the flow. Enhanced information can also be applied to inventory problems: DNS support has been used to identify and patch DNS servers vulnerable to the Kaminsky [10] exploit.

5.2 PRISM: Data Reduction in a Multi-Stage Monitoring Architecture

YAF was also deployed in 2010 as part of the integrated trial of the European Union Seventh Framework PRISM [11, 2] project at a regional network service provider in Italy. The aim of the PRISM project is to apply semantically-aware access control, a multi-stage monitoring architecure, aggressive data reduction, and data protection and anonymisation techniques to enable privacy-aware and privacy-preserving network monitoring. The PRISM architecture is split into front-end (FE) components which observe a packet stream and reject packets unlikely to be of interest, back-end (BE) components which further reduce, analyze, and store data received from the FE, and beyond-the-back-end (BBE) components.

A key insight of the PRISM project is that data reduction, such as reducing a packet stream down to a flow stream early in the monitoring pipeline, removes potentially privacy-relevant information; in the case of flow data generation, the elimination of payload data significantly reduces the potential privacy impact of the content in the observed data stream.

YAF was used as one of the data reduction components in the back-end of the integrated trial for a Skype-detection application. In this scenario, higher levels of privilege (e.g., a network administrator debugging a specific connection issue for a customer) would allow full dissection of the Skype traffic from one host to another, using a packet-based Skype analyzer [1] beyond-the-back-end, while a lower level of privilege (e.g., a junior administrator preparing a report on the volume of Skype traffic on the network) would use a flow-based method, reducing the fidelity of data seen beyond the back-end. The PRISM access control system would automatically select the correct reduction component based upon the privilege and purpose of the request.

In this deployment, YAF was used as a straight flow meter - payload inspection and capture were specifically disabled. YAF was invoked on `libpcap` dumpfiles generated by the `capfix` utility developed as part of the PRISM project, which provides for the framing of packet traces in IPFIX/PSAMP, and configured to export to a `snack` instance beyond the back-end, which in turn generated a list of Skype connection events used to count distinct hosts running Skype.

YAF was selected for this application due in large part to the effortlessness of integration. PRISM had selected a data plane based entirely on IPFIX early in development, in order to maximize the potential to leverage off-the-shelf standards-compliant components within the PRISM architecture. When the project decided on a split packet/flow based approach to Skype traffic analysis, YAF was an obvious choice for the flow meter, and its selection reduced the implementation of this stage of the data plane to simply writing a little glue.

6 Availability

YAF is distributed by the Software Engineering Institute as free software under a dual-license system. The general public may download YAF (and all of the Network Situational Awareness team's software, including SiLK) from

`http://tools.netsa.cert.org`

under the terms of the GNU General Public License version 2. The US government maintains separate rights in the software, and may use the software under the terms of the Government Purpose License Rights of DFARS. Support for building and using YAF and all the Network Situational Awareness tools is available by sending email to

`netsa-help@cert.org`

YAF should work on most flavors of Unix, and is developed and tested on Mac OS X, Linux, FreeBSD, OpenBSD, and Solaris.

7 Acknowledgments

The authors would like to express their deep gratitude to the members of the Network Situational Awareness team at CERT for helping build and support YAF, especially Emily Ecoff, Michael Duggan, and Tony Cebzanov. Funding and support for the development of YAF has come from the United States Department of Defense, which supports the Software Engineering Institute as a federally funded research and development center.

References

[1] Adami, D., Callegari, C., Giordano, S., Pagano, M., and Pepe, T. A real-time algorithm for Skype traffic detection and classification. In *9th International Conference on Wired/Wireless Networking* (Sept. 2009).

[2] Bianchi, G., Teofili, S., and Pomposini, M. New directions in privacy-preserving anomaly detection for network traffic. In *NDA '08: Proceedings of the 1st ACM workshop on Network data anonymization* (New York, NY, USA, 2008), ACM, pp. 11–18.

[3] BONWICK, J. The slab allocator: an object-caching kernel memory allocator. In *USTC'94: Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference* (Berkeley, CA, USA, 1994), USENIX Association, pp. 6–6.

[4] CISCO SYSTEMS, INC. Cisco IOS Netflow Introduction. http://www.cisco.com/go/netflow. [Accessed 19 August 2010].

[5] CISCO SYSTEMS, INC. Hyperconnectivity and the Approaching Zettabyte Era. Cisco VNI White Paper, June 2010.

[6] CLAISE, B., BRYANT, S., LEINEN, S., DIETZ, T., AND TRAMMELL, B. Specification of the IP Flow Information Export Protocol. RFC 5101 (Proposed Standard), Jan. 2008.

[7] CLAISE, B., SADASIVAN, G., VALLURI, V., AND DJERNAES, M. Cisco Systems NetFlow Services Export Version 9. RFC 3954 (Informational), Oct. 2004.

[8] DERI, L. nprobe – netflow/ipfix network probe. http://www.ntop.org/nProbe.html, Oct 2006. [Accessed 9 August 2010].

[9] DERI, L., AND SUIN, S. Effective traffic measurement using ntop. *IEEE Communications Magazine* (May 2000), 138–143.

[10] DOUGHERTY, C. R. "Vulnerability Note VU#800113 multiple DNS implementations vulnerable to cache poisoning". http://www.kb.cert.org/vuls/id/800113, 2009. [Accessed 27 July 2010].

[11] FP7 PRISM PROJECT. PRIvacy-aware Secure Monitoring. http://www.fp7-prism.eu. [Accessed 6 August 2010].

[12] GATES, C., COLLINS, M., DUGGAN, M., KOMPANEK, A., AND THOMAS, M. More netflow tools for performance and security. In *LISA '04: Proceedings of the 18th USENIX conference on System administration* (Berkeley, CA, USA, 2004), USENIX Association, pp. 121–132.

[13] HAZEL, P. PCRE - Perl Compatible Regular Expressions. http://www.pcre.org.

[14] INTERNET ASSIGNED NUMBERS AUTHORITY. IP Flow Information Export (IPFIX) Information Elements. http://www.iana.org/assignments/ipfix/.

[15] MILLER, D. softflowd - fast software netflow probe. http://www.mindrot.org/projects/softflowd, Oct 2006. [Accessed 9 August 2010].

[16] OPENDPI. http://www.opendpi.org. [Accessed 6 August 2010].

[17] QOSIENT, LLC. Argus: Auditing Network Activity. http://www.qosient.com/argus/. [Accessed 19 August 2010].

[18] QUITTEK, J., BRYANT, S., CLAISE, B., AITKEN, P., AND MEYER, J. Information Model for IP Flow Information Export. RFC 5102 (Proposed Standard), Jan. 2008.

[19] ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, E. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630.

[20] SFLOW.ORG. sflow products - network equipment. http://www.sflow.org/products/network.php, 2010. [Accessed 9 August 2010].

[21] SHANNON, C., MOORE, D., AND CLAFFY, K. C. Beyond folklore: observations on fragmented traffic. *IEEE/ACM Trans. Netw. 10*, 6 (2002), 709–720.

[22] SHANNON, C. E. A mathematical theory of communication. *Bell System Technical Journal 27* (Jul and Oct 1948), 379–423,623–656.

[23] STEENBERGEN, R. A. sflow – why you should use it and like it. In *39th Meeting of the North American Network Operator's Group (NANOG 39)* (Feb 2007).

[24] TRAMMELL, B., AND BOSCHI, E. Bidirectional Flow Export using IP Flow Information Export. RFC 5103 (Proposed Standard), Jan. 2008.

[25] TRAMMELL, B., BOSCHI, E., MARK, L., ZSEBY, T., AND WAGNER, A. Specification of the IP Flow Information Export File Format. RFC 5655 (Proposed Standard), Oct. 2009.

[26] TRAMMELL, B., AND GATES, C. NAF: the NetSA Aggregate Flow Tool Suite. In *20th USENIX Large Installation System Administration Conference (LISA '06)* (Dec 2006), pp. 221–231.

Nfsight: NetFlow-based Network Awareness Tool

Robin Berthier
Coordinated Science Laboratory
Information Trust Institute
University of Illinois
Urbana-Champaign, IL, USA
rgb@illinois.edu

Michel Cukier
The Institute for Systems Research
Clark School of Engineering
University of Maryland
College Park, MD, USA
mcukier@umd.edu

Matti Hiltunen, Dave Kormann, Gregg Vesonder, Dan Sheleheda
AT&T Labs Research
180 Park Ave.,
Florham Park, NJ, USA
{*hiltunen,davek,gtv*}*@research.att.com, dsheleheda@att.com*

Abstract

Network awareness is highly critical for network and security administrators. It enables informed planning and management of network resources, as well as detection and a comprehensive understanding of malicious activity. It requires a set of tools to efficiently collect, process, and represent network data. While many such tools already exist, there is no flexible and practical solution for visualizing network activity at various granularities, and quickly gaining insights about the status of network assets. To address this issue, we developed Nfsight, a NetFlow processing and visualization application designed to offer a comprehensive network awareness solution. Nfsight constructs bidirectional flows out of the unidirectional NetFlow flows and leverages these bidirectional flows to provide client/server identification and intrusion detection capabilities. We present in this paper the internal architecture of Nfsight, the evaluation of the service, and intrusion detection algorithms. We illustrate the contributions of Nfsight through several case studies conducted by security administrators on a large university network.

1 Introduction

Network awareness, i.e., knowledge about how hosts use the network and how network events are related to each other, is of critical importance for anyone in charge of administering a network and keeping it secure [11]. The goal of network awareness is to provide relevant information for decision-making regarding network planning, maintenance, and security. NetFlow is among the most-used information sources for gaining awareness in large networks because it offers a good trade-off between the level of detail provided and scalability. As a result, a majority of networks are already instrumented through their routers to collect and export NetFlow, and a variety of tools are available to process such data [18, 36, 27]. However, there is still no practical solution to visualizing network activity at various granularities and quickly gaining insight about the status of network assets. Numerous attempts have been made [37, 31, 5] and are detailed in Section 4, but none has gained a broad audience.

We developed a tool called *Nfsight* to address these challenges. The objective of Nfsight is to offer a comprehensive network awareness solution through three core functions: 1) passive identification of client and server assets, 2) a web interface to query and visualize network activity, and 3) a heuristic-based intrusion detection and alerting system. Nfsight is designed to be simple, efficient, and highly practical. It consists of three major components: a Service Detector, an Intrusion Detector, and a front-end Visualizer. The Service Detector component analyzes unidirectional NetFlow flows to identify client and server end points using a set of heuristics and a Bayesian inference algorithm. The Intrusion Detector component detects suspicious activity through a set of graphlet-based signatures [13], and the front-end Visualizer allows administrators to query, filter, and visualize network activity. We trained and evaluated the Service Detector using two different datasets of 30 minutes of packet dumps collected at the border of a large university network. The Intrusion Detector was evaluated by security experts over a period of four months. Based on several months of testing in a production environment of 40,000 computers, we believe Nfsight can greatly assist administrators in learning about network activity and managing their assets.

The rest of the paper is organized as follows. In Section 2, we provide an overview of Nfsight and we present the implementation and evaluation of the different components: the Service Detector (Section 2.2), the Intrusion Detector (Section 2.3), and the front-end Visualizer (Section 2.4). We discuss a number of use cases in Section 3 and we compare our approach to related work in Section 4. Finally, Section 5 offers some concluding remarks.

2 Architecture and Implementation

This section provides an overview of the architecture of Nfsight and describes in detail the implementations of the Service Detector, the Intrusion Detector and the front-end Visualizer.

2.1 Nfsight Architecture Overview

The architecture of Nfsight is presented in Figure 1. Nfsight uses non-sampled unidirectional NetFlow provided by a collector such as Nfdump/Nfsen [19]. A network flow is defined as a unidirectional sequence of packets that share source and destination IP addresses, source and destination port numbers, and protocol (e.g., TCP or UDP). A NetFlow flow carries a wide variety of network-related information about a network flow including the timestamp of the first packet received, duration, total number of packets and bytes, input and output interfaces, IP address of the next hop, source and destination IP masks, and cumulative TCP flags in the case of TCP flows.

The Service Detector component takes NetFlow flows and converts them into bidirectional flows in the IPFIX format (bidirectional flow format specified by the IPFIX working group [4]). During this process, it identifies client and server end points using a set of heuristics and a Bayesian inference algorithm. The bidirectional flows, denoted by *IPFIX* in Figure 1, are stored in flat files, while the server end points, denoted by *Assets* in Figure 1, are stored in a MySQL database. The Intrusion Detector component detects suspicious activity through a set of graphlet-based signatures [13] applied on the bidirectional flows. The high-level network activity and event alerts generated by the Intrusion Detector are stored in a MySQL database. An aggregation script runs periodically to maintain a round-robin structure in the database and to provide three aggregation levels: every five minutes, hourly, and daily. We detail the data storage and representation solution of Nfsight in Section 2.4. The front-end Visualizer allows administrators to query, filter, and visualize network activity. They can access the application simply by using a web browser and they can collaborate through a shared knowledge base of events reported either automatically by the Service Detector and Intrusion Detector or manually by operators.

2.2 Passive Service Detection

2.2.1 Definitions

In the rest of the paper we use the following definitions. A *server* is a network application that provides a service by receiving request messages from clients and generating response messages. A server is hosted on a computer identified by its IP address and accepts requests sent to a specific port. In this paper, we focus on servers using the UDP and TCP protocols. We are interested in both transient and permanent servers. Specifically, we consider P2P transactions a part of the client/server model, even if the server in this case may be handling client requests for only a few minutes and for only specific clients. We define an *end point* as a tuple $\{$IP address, IP protocol, Port number$\}$. An end point may represent either a client or a server.

We define a *network session* as a *valid communication* between one client end point and one server end point. All UDP flows are considered to be valid, but TCP flows are valid only if both the request and the reply flows carry at least two packets and the TCP acknowledgement flag. For example, if a server refuses a TCP connection handshake by sending a reset flag to the source end point, the communication is not considered valid. Finally, we use the term *network transaction* to describe any set of flows between two end points during a time window smaller than the maximum age limit of a flow (usually 15 minutes). There are two types of network transactions: unidirectional and bidirectional. We assume that bidirectional transactions are always between a client and a server and that bidirectional transactions are always initiated by a client.

2.2.2 Approach

The task of accurately detecting servers based solely on NetFlow flows is challenging because NetFlow does not keep track of the logic of network sessions between clients and servers. Specifically, we have to address the following challenges:

1. NetFlow may break up a logical flow into multiple separate flows;

2. NetFlow is made of unidirectional flows and therefore we need to identify the matching unidirectional flows to make up bidirectional flows and identify valid network sessions;

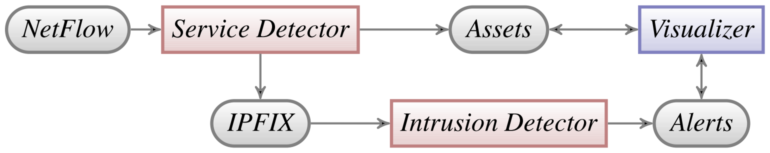


Figure 1: Nfsight architecture

3. Identifying the server end point in a network session is difficult because the TCP flags in the request and reply flows are typically identical for valid bidirectional flows. Furthermore, the flow timestamps have proven to be sometimes unreliable and more often, the request and reply flows have identical timestamps due to the granularity of the timestamps.

We solve the first and second challenges by matching and merging the NetFlow flows as follows. First, for each collection period (usually 5 minutes), we merge all network flows that have the same source and destination end points to eliminate any artificial breaking of unidirectional flows. Then, to address the issue of combining unidirectional flows into network sessions, we generate bidirectional flows by merging all flows collected during a given time window that have opposite source and destination end points. The network sessions are then selected based on the number of packets and flags and according to the definition of valid communication above. The last step is to address the third challenge, i.e., to identify client and server end points for every network session. We describe below the approach we developed to perform this task.

2.2.3 Server Identification Heuristics

To correctly identify client and server end points for every valid bidirectional flow, we developed a set of heuristics that determine if an end point is a server (or not). These heuristics were developed to cover a variety of intuitions gathered from network experts. A heuristic may be based on the attributes of an individual (bidirectional) flow or it may consider a set of flows.

The heuristics implemented are:

H.0 Flow timing. Let t_1 and t_2 be the timestamps of the unidirectional flows constituting a bidirectional flow. The source of the flow with the larger (more recent) timestamp is likely the server. The difference between t_1 and t_2 provides an indication on the probability that this heuristic will identify the correct end point as a server. If the timestamps are identical, they cannot be used to decide which end point is the server.

H.1 Port number. Let p_1 and p_2 be the port numbers associated with a bidirectional flow. The end point with the smaller port number is likely the server. If the port numbers are identical, they cannot be used to decide which end point is the server.

H.2 Port number with threshold at 1024. If an end point has a port number lower than 1024, then it is likely a server. The value of 1024 corresponds to the limit under which ports are considered privileged and designated for well-known services. If both ports are above or below 1024, this heuristic cannot be used to decide which end point is the server.

H.3 Port number advertised in /etc/services. If the port number of an end point is listed in the standard UNIX file /etc/services that compiles assigned port numbers and registered port numbers [12], then it is likely a server. If both or neither port numbers are in /etc/services, this heuristic cannot be used to decide which end point is the server.

H.4 Number of distinct ports related to a given end point. If two or more different port numbers (in different flows) are associated with an end point, the end point is likely a server. The number of different port numbers related to an end point provides an indication on the probability that this heuristic will correctly identify the server. This heuristic comes from the fact that ports on the client-side are often randomly selected. Therefore, ports on the client-side of a connection are less likely to be used in other connections compared to ports on the server-side. If both end points are related to the same number of ports, then this heuristic cannot be used to decide which end point is the server.

H.5 Number of distinct IP addresses related to a given end point. This heuristic is identical to the previous one but counts IP addresses instead of ports.

H.6 Number of distinct tuples related to a given end point. This heuristic is identical to the previous one but counts end points instead of single IP addresses. This heuristic is based on the observation that each server typically has two or more clients that use the service. Furthermore, even if only one real user accesses the service (e.g., identified by the IP address of the user's machine), the communication will likely require multiple connections and the client side of the access often uses different port numbers. Thus, multiple end points will be detected.

2.2.4 Evaluation of Individual Heuristics

We evaluated the accuracy of each heuristic by using bidirectional flows generated by Argus [26] as the ground truth. Argus is a flow processing application that generates bidirectional flows from packet data. We considered Argus to be more accurate than Nfsight, and able to produce a baseline dataset for our evaluation, since it uses detailed packet data as input instead of the high level flow data used by Nfsight. We collected a dataset of 30 minutes of network traffic from the border of a large university network and analyzed the data using Argus to identify bidirectional flows and their server end points. We then processed the data using Nfsight to generate bidirectional flows (6.2 million records) and applied the heuristics to determine the server end points. We define the accuracy of a heuristic as the probability that it correctly identifies the server end point of a bidirectional flow. The accuracy is estimated by dividing the number of bidirectional flows correctly oriented based on ground truth from Argus by the total number of bidirectional flows correctly and incorrectly oriented.

For heuristics H.1, H.2 and H.3 the accuracy probability is a single value. Specifically, based on our input data, we calculated the accuracies of these heuristics to be 0.78, 0.75, and 0.74, respectively. Heuristics H.0, H.4, H.5, and H.6 depend on parameter values, either on time difference or number of distinct ports, IP addresses, or tuples. Therefore, we can evaluate their accuracy with regard to the parameter value as demonstrated in Figures 2 to 5 (up to 10 seconds for H.0, and up to 100 ports, IPs, and tuples for H.4, H.5 and H.6). These plots show that the accuracy increases with the time difference between requests and replies (Figure 2), the number of related ports (Figure 3), IP addresses (Figure 4) and {IP, protocol, port} tuples (Figure 5) between source and destination end points. Note that the similarities between Figures 3 and 5 can be explained by the fact that the client ports are randomly selected among 64,511 values. Therefore, the number of client ports and the number of clients are different only in the case where two clients communicating with the same server select the same source port randomly.

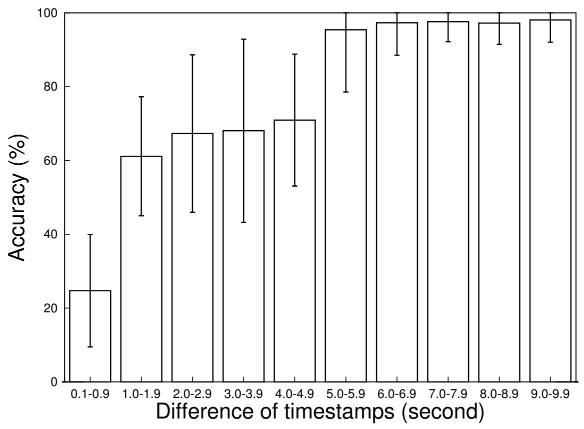


Figure 2: Bidirectional flow orientation accuracy increases with the timestamp difference between request and reply flows (H.0)

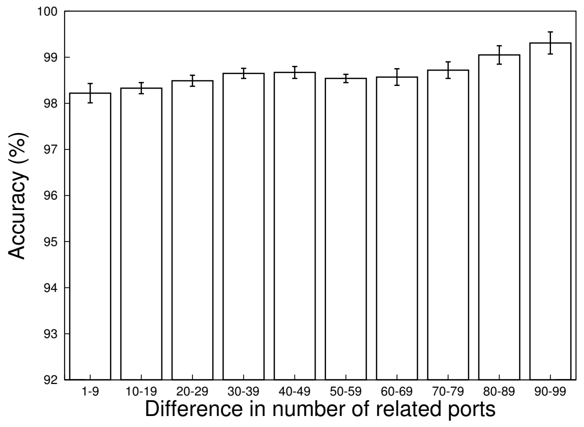


Figure 3: Bidirectional flow orientation accuracy increases with the difference between the number of source and destination related ports (H.4)

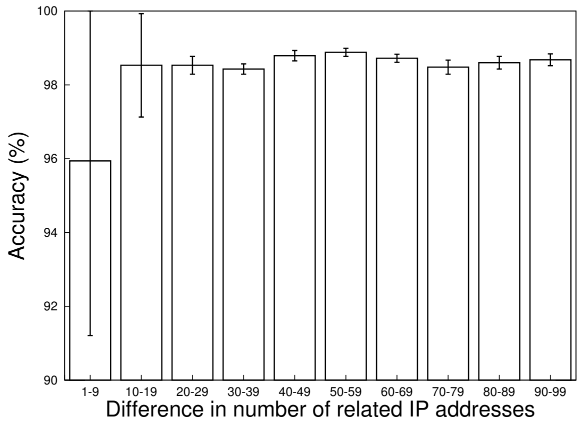


Figure 4: Bidirectional flow orientation accuracy increases with the difference between the number of source and destination related IP addresses (H.5)

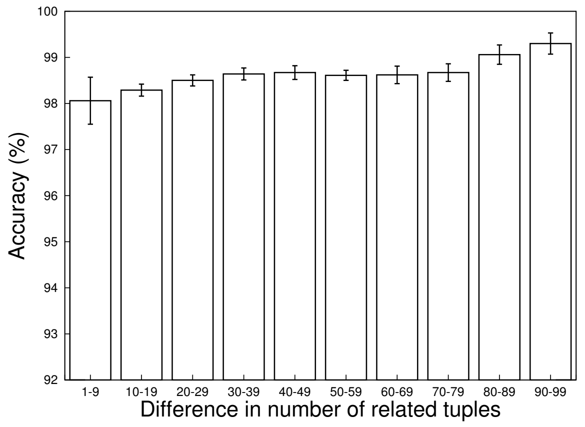


Figure 5: Bidirectional flow orientation accuracy increases with the difference between the number of source and destination related tuples (H.6)

2.2.5 Combining heuristics

While individual heuristics can be used to identify server end point, they cannot make a decision for all the flow processed. For example, some flows have similar request and reply timestamps, or similar source and destination port numbers. To address this issue and to get a better estimate, we combine the evidence provided by the different heuristics using basic Bayesian inference. We consider each end point that is present in at least one bidirectional flow. For each end point X, we have two possible hypotheses:

- H_s: end point X is a server.
- H_c: end point X is a client.

The different heuristics are used to identify evidence E in the bidirectional flows. For example, the fact that there is a difference in unidirectional flow timestamps provides evidence based on heuristic H.0. Bayesian inference combines any prior knowledge (the prior probability of hypothesis H_i being true denoted by $P(H_i)$) with information gained from new evidence E to produce a new estimate of the probability that the hypothesis is true using the formula:

$$P(H_i|E) = \frac{P(E|H_i) * P(H_i)}{\sum P(E|H_j) * P(H_j)}$$

where $P(E|H_i)$ denotes the probability that evidence E is present in a flow or set of flows given that hypothesis H_i is true, that is, that a heuristic we use to generate the evidence is accurate. While these conditional probabilities could be assigned using expert knowledge, we use the heuristic accuracies measured previously. We summarize these empirical results in Table 1.

Table 1: Individual heuristic accuracies used as conditional probabilities for Bayesian inference

Heuristic	*Output*	*Accuracy*
H.0	]0; 1.0[	0.25
	[1.0; 5.0[	0.7
	[5.0; ∞[	0.99
H.1	True	0.78
H.2	True	0.75
H.3	True	0.74
H.4	1	0.97
	[2; 29]	0.9825
	[30; 74]	0.9875
	[75; ∞[	0.99
H.5	1	0.95
	[2; ∞[	0.98
H.6	1	0.97
	[2; 29]	0.9825
	[30; 74]	0.9875
	[75; ∞[	0.99

Note that while the naive Bayesian formulation used assumes independence of evidence, and some of the heuristics are obviously correlated, we find the approach still useful for combining the heuristics. We are evaluating other combining techniques, such as Bayesian networks, that allow explicit representation of dependencies between heuristics.

2.2.6 Evaluation of Bayesian Inference

We evaluated the accuracy of Nfsight to address two related issues: 1) generating correctly oriented bidirectional flows, and 2) accurately identifying server end points. For the first issue, we applied the approach previously described to individually evaluate heuristics by using Argus to provide ground truth. For the second issue, we compared server end points discovered by Nfsight against Pads [23]. Pads is a packet-based passive service discovery tool. Similarly to Argus, we considered Pads to be more accurate than Nfsight and able to produce a baseline dataset for our evaluation, since it works from detailed packet data instead of high level flow data. In our evaluation, we are interested in measuring how much accuracy we lose by working only with flows.

We collected a second dataset of 30 min of network traffic from the border of the same large university network. Note that the dataset used for determining the accuracy of individual heuristics (summarized in Table 1) and the dataset used for this evaluation were collected five months apart.

Concerning the issue of generating correctly oriented bidirectional flow, we analyzed 3,617,077 bidirec-

Table 2: Bidirectional flow orientation accuracy grouped by confidence level from Bayesian inference

Heuristic	*Able to decide*	*Accuracy*
H.0	11.49%	94.54%
H.1	63,98%	85.54%
H.2	48.14%	98.15%
H.3	47.73%	98.17%
H.4	63.28%	93.72%
H.5	55.51%	88.76%
H.6	63.38%	92.58%

tional flows generated by both Nfsight and Argus. On this dataset, Argus could decide on the orientation for 2,356,616 flows (65.15%) while Nfsight could make a decision for 3,616,942 flows (99.996%). When Argus could decide, we evaluated that Nfsight agreed on the orientation for 2,183,440 flows. This represents an accuracy of 92.65%.

To understand further the contribution of the Bayesian inference to combine heuristics, we expand the comparison against Argus for each individual heuristic in Table 2. These results reveal that individual heuristics provide high accuracies but they are able to decide for only a fraction of the flows. For instance, H.0 agrees with Argus for 94.54% of the flows, but could decide for only 11.49% of the flows. The accuracies of H.1 to H.6 range from 85.54% to 98.17%, while the decision capabilities of H.1 to H.6 lie between 47.73% and 64.98%. These results show the importance of the Bayesian inference to combine heuristics, because it allows the overall decision capability to reach almost 100% while keeping the overall accuracy above 92%.

The final step of the evaluation was to address the second issue of accurately identifying server end points. We compared server end points identified by Nfsight and Pads. Out of 57,985 TCP servers detected by Pads from the packet data, Nfsight was able to identify 45,932, which represents an accuracy of 79.21%. We investigated the services detected by Pads and not by Nfsight, and we found that the majority of them were source end points of unidirectional flows. This pattern indicates that our evaluation dataset did not contain both directions of network sessions for some flows. The lack of request or reply flows can come from asymmetric routing or sampling. We discuss in Section 3.4 the need to develop additional heuristics that would allow Nfsight to handle such cases.

2.3 Intrusion Detection

Once bidirectional flows have been generated by the Service Detector, the Intrusion Detector identifies malicious activity using a set of detection rules based on the graphlet detection approach [13]. In this approach, the patterns of host behavior are captured based on the flows, and then these patterns are compared with intrusion detection signatures. Patterns are generated for each host and contain statistical information such as host popularity, number of ports used, number of failed connections, and total number of packets and bytes exchanged. Note that working with bidirectional flows simplifies the definition of the detection rules and the pattern lookup since the source and destination end points of each network transaction are already known. We describe in detail the data structure and the different detection rules we evaluated in the remainder of this section.

2.3.1 Data Structures

The intrusion detection algorithm processes each bidirectional flow generated over the last batch of NetFlow flows collected (5 minutes in our setup) and creates or updates two dictionary structures: one for the source and the other for the destination IP addresses of the flow under review. The structure for source IP addresses captures the fan-out relationships, while the other captures the fan-in relationships. These dictionaries are organized in a three-level hierarchy, where the IP address and the protocol are used as keys for the first and second levels, respectively. The different fields at the third level are therefore all related to a specific {IP, protocol}. These fields are:

- *Peer*: the set of distinct related IP addresses;
- *Port*: the set of distinct related destination or source ports;
- *TCP flag*: the set of distinct flag combinations used;
- *Packet*: the total number of packets sent or received;
- *Byte*: the total number of bytes sent or received;
- *Flows*: the total number of bidirectional flows sent or received;
- *Failed connections*: the total number of unidirectional flows sent or received;
- *Last source end point*: the source port, IP address and TCP flag of the last flow captured;
- *Last destination end point*: the destination port, IP address and TCP flag of the last flow captured.

The last two fields are not used by the detection rules but were requested by our team of administrators as an additional time-saving information when classifying alerts sent by email. For example, consider a case where a host is detected as initiating a large number of failed connections over the last 5 minutes. If the last source port appears to be random and the last destination port is TCP/445, then the host will be immediately classified as compromised by a malware that spreads over the Netbios service. On the other hand, if the last source port is TCP/80 and the last destination port appears to be random, then the host will likely be classified as a victim of a denial-of-service attack.

2.3.2 Detection Rules

The next step performed by the intrusion detection algorithm is to process each bidirectional flow again and to try to match flow information and source and destination host patterns against a set of signatures. We created a set of 13 rules organized in 3 categories: malformed flows, one-to-many, and many-to-one relationships. These rules and categories are described in Table 3. They were based on expert knowledge and on a study of attack traces to cover noisy malicious activity such as scanning and denial-of-service activities generated by compromised hosts. We note that Nfsight provides the data structures and rule matching algorithm to enable administrators to create and evaluate more fine-grained rules.

As shown in Table 3, rules in the one-to-many and many-to-one categories use thresholds. We defined these thresholds empirically from a study of attack traces and the feedback we received during the testing of the different detection rules. These thresholds are likely specific to a given network and a given time window of analysis. Thus, they are subject for future tuning. The threshold values used in our experiments were *max_dst_ip* = 200, *max_dst_port* = 250, *max_src_ip* = 500, and *max_src_port* = 500. Rules in the malformed flow category use three data structures to catch incorrectly formed packets. These are: *invalid_code* to detect incorrect ICMP type and code combinations; *invalid_ip* to detect forged or misconfigured IP addresses sent to private or unallocated subnets; and *invalid_flag* to detect incorrect TCP flag combinations.

2.3.3 Evaluation

Flow-based intrusion detection implementations often suffer from two problems: 1) the difficulty to validate and tune anomaly detection rules and 2) the difficulty to access and understand the root cause of the malicious activity detected. The first problem is illustrated in the context of application detection in [14], where the authors observe that the tuning of the 28 configurable threshold parameters of the original graphlet approach [13] is too cumbersome. To simplify rule tuning and validation,

```
192.168.1.2 [One-to-many IP] IP contacting more than 200 distinct
targets in less than 5min

 * Heuristic: 201

 * First detected on: 2010-08-10 14:05:00
 * Last detected on: 2010-08-10 16:55:00
 * Number of occurrences: 52,908
 * Total flows: 52,908
 * Unanswered flow requests: 52,908 (100\%)
 * Packets: 89,918
 * Bytes: 4,316,160

 * Average number of related host every 5min: 4,580
 * Average number of related port every 5min: 2

 * Last source port: 3317 (2,339 distinct port(s) used every 5min)
 * Last related tuple: 192.168.26.198 TCP/445
 * Last flag value (if TCP): 2

To visualize related Nfsight data:
https://nfsight/index.php?net=192.168.1.2&time=201008101655

---------------------------------
Please rate this alert by clicking on one of the following links:

 [+] True Positive:
https://nfsight/email_validation.php?q=156505&r=1&auth=r25kfGVk

 [-] False Positive:
https://nfsight/email_validation.php?q=156505&r=-1&auth=r25kfGVk

 [?] Inconclusive:
https://nfsight/email_validation.php?q=156505&r=0&auth=r25kfGVk
---------------------------------
```

Figure 6: Example of an alert email with validation links

we developed an evaluation process using email alerts. The objective is to leverage administrator expertise while minimizing the time and effort required to validate detection rules. Specifically, each alert emailed to security administrators contains three embedded links that allow the alert receiver to rate the alert as true positive, false positive, or unknown. A fourth link allows administrators to open the front-end Visualizer and display the network activity related to the alert under review. An example of an alert email with validation links is given in Figure 6.

The second problem is due to the fact that flows are based on aggregated header information and lack details on the payloads required to precisely identify attack exploits. It is not possible to fully address this problem if we restrict ourselves to Netflow, but we note that the different visualization solutions offered by Nfsight and described in Section 2.4 help to understand and assess the illegitimate nature of suspicious network activity.

We configured the email validation script to send no more than five alert emails in two batches per day to four experts: two security administrators and two graduate students working in network security. Alerts were ranked according to the number of flows and the number of detection occurrences. Then the top five internal IP addresses for which no alerts email had been previously sent were selected. Table 4 presents the validation results collected over a period of four months for the five detection rules that triggered alerts. In this table, *TP* denotes the number of alerts labeled as “true positives”, *FP* denotes the number of alerts labeled as “false positives”, and *Unknown* represents alerts for which experts could not decide if the activity was malicious. The results in-

Table 3: Intrusion detection rules

Id	*Name*	*Category*	*Filter*
101	Identical source and destination	Malformed flow	src_ip = dst_ip
102	Invalid ICMP flow size	Malformed flow	proto = ICMP and total_byte $\leq$ 64000
104	Invalid ICMP code	Malformed flow	proto = ICMP and icmp_code $\in$ invalid_code
105	Invalid IP address	Malformed flow	(src_ip or dst_ip) $\in$ invalid_ip
106	Invalid TCP flag	Malformed flow	proto = TCP and flag $\in$ invalid_flag
201	One-to-many IP	One-to-many	failed_connection $\geq$ 1 and unique_dst_ip $\geq$ max_dst_ip and unique_flag $\leq$ 1
301	One-to-many Port	One-to-many	failed_connection $\geq$ 1 and unique_dst_port $\geq$ max_dst_port and unique_flag $\leq$ 1
401	Many-to-one IP on TCP flows	Many-to-one	proto = TCP and flag $\notin$ {19, 27, 30, 31} and unique_src_ip $\geq$ max_src_ip and unique_flag $\leq$ 1
402	Many-to-one IP on ICMP flows	Many-to-one	proto = ICMP and unique_src_ip $\geq$ max_src_ip
403	Many-to-one IP on UDP flows	Many-to-one	proto = UDP and unique_src_ip $\geq$ max_src_ip
501	Many-to-one Port on TCP flows	Many-to-one	proto = TCP and flag $\notin$ {19, 27, 30, 31} and unique_src_port $\geq$ max_src_port and unique_dst_port = 1 and unique_flag = 1
502	Many-to-one Port on ICMP flows	Many-to-one	proto = ICMP and unique_src_port $\geq$ max_src_port and unique_dst_port = 1
503	Many-to-one Port on UDP flows	Many-to-one	proto = UDP and unique_src_port $\geq$ max_src_port and unique_dst_port = 1

dicate that rules *105* and *201* are relatively accurate. We note that these two rules allowed our team of administrators to detect 18 internal compromised hosts. However, rules *106*, *301*, and *501* have a high rate of false positives. The poor performance of rule *106* can be explained by the facts that invalid TCP flag combinations may be due to misconfigured hosts or legitimate TCP connections may be broken over different flows. The false positives for rules *301* and *501* are mainly due to heavily used servers for which the thresholds *max_src_ip* and *max_src_port* were too low. The feedback offered by this validation process and the labeled alerts help adjusting the parameters and thresholds of the detection rules. We are working towards implementing an automated process to adjust these values and revise the detection rules.

2.4 Data Visualization

The front-end Visualizer allows administrators to query, filter, and visualize network activity. This section presents the web interface of Nfsight and the underlying data storage solution.

2.4.1 Hybrid Data Storage

Alerts and client/server end points identified by the Service Detector and Intrusion Detector modules are stored in a MySQL database at three aggregation levels: five minutes, hourly, and daily. An aggregation script that expires data at different granularities runs periodically to maintain a round-robin structure in the database. This structure allows the storage of a large volume of data (88 million records organized in 107 tables in our implementation) while offering a fixed database size (11GB in our implementation) and a fast access to network end points at different time granularity levels. We configured the 5-minute granularity data to expire after two weeks.

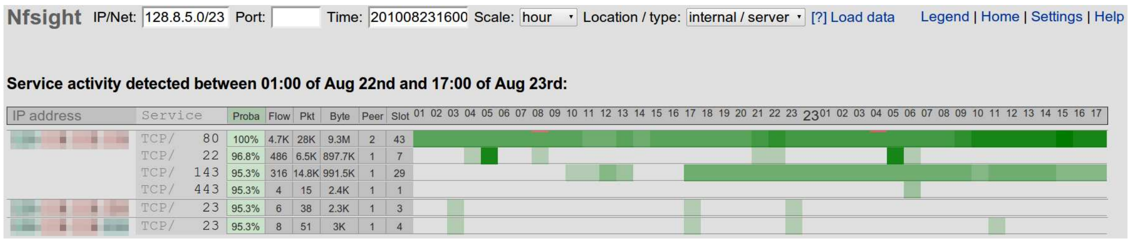
2.4.2 Web Front-end

The front-end is developed in PHP and consists of a search engine, a dashboard, and a network activity visualization table. The dashboard presents the latest generated alerts and the top 20 servers, services, scanned services, and internal scanners. The search form and the network activity visualization table are represented in Figure 7. We note that IP addresses in Figure 7 and in Section 3 have been pixelated on purpose. The search form enables administrators to filter activity per subnet, IP, time period, and type of activity (i.e., internal or external client and/or server).

The visualization table is organized by host IP, port number, and type of activity (either client for source port or server for destination port). For each end point, the tool provides both statistical information and a visualization of the activity over the given time period. The statistical information includes the confidence value given

Table 4: Validation results for each detection rule triggered

Id	*Total Validated*	*TP*	*FP*	*Unknown*	*Accuracy: TP/(FP+TP)*
105	23	11	4	8	73.3%
106	27	3	19	5	13.6%
201	68	40	21	7	65.6%
301	94	30	41	23	42.3%
501	78	21	38	19	35.6%


Nfsight IP/Net: 128.8.5.0/23 Port: Time: 201008231600 Scale: hour Location / type: internal / server [?] Load data Legend | Home | Settings | Help

Service activity detected between 01:00 of Aug 22nd and 17:00 of Aug 23rd:

IP address	Service	Proba	Flow	Pkt	Byte	Peer	Slot
[redacted]	TCP/ 80	100%	4.7K	28K	9.3M	2	43
	TCP/ 22	96.8%	486	6.5K	897.7K	1	7
	TCP/ 143	95.3%	316	14.8K	991.5K	1	29
	TCP/ 443	95.3%	4	15	2.4K	1	1
[redacted]	TCP/ 23	95.3%	6	38	2.3K	1	3
[redacted]	TCP/ 23	95.3%	8	51	3K	1	4

Figure 7: Nfsight front-end Visualizer

by the Bayesian inference algorithm and the number of flows, packets, and bytes. The network activity is represented as a time series using a heat map that visually reveals the number and type of flows detected over the time period. A color code enables network operators to separate client activity (blue) from server activity (green), and also to identify the fraction of invalid, i.e., non-answered (red), flows sent/received by an end point. The intensity of the color is used to represent the number of flows. Some servers may receive both unidirectional and bidirectional flows, represented by a block divided into green and red parts that represent the proportion of unidirectional and bidirectional flows received by the server. These unidirectional flows may be due to invalid packets that the server rejected, an overwhelming number of requests, or unidirectional flows that the Service Detector component failed to pair correctly. Additional examples of the visualization capabilities of Nfsight are provided in Section 3.

3 Use Cases

We present in this section different use cases to demonstrate how Nfsight can help security administrators and network operators in their daily tasks.

3.1 Network Awareness

3.1.1 Server Identification

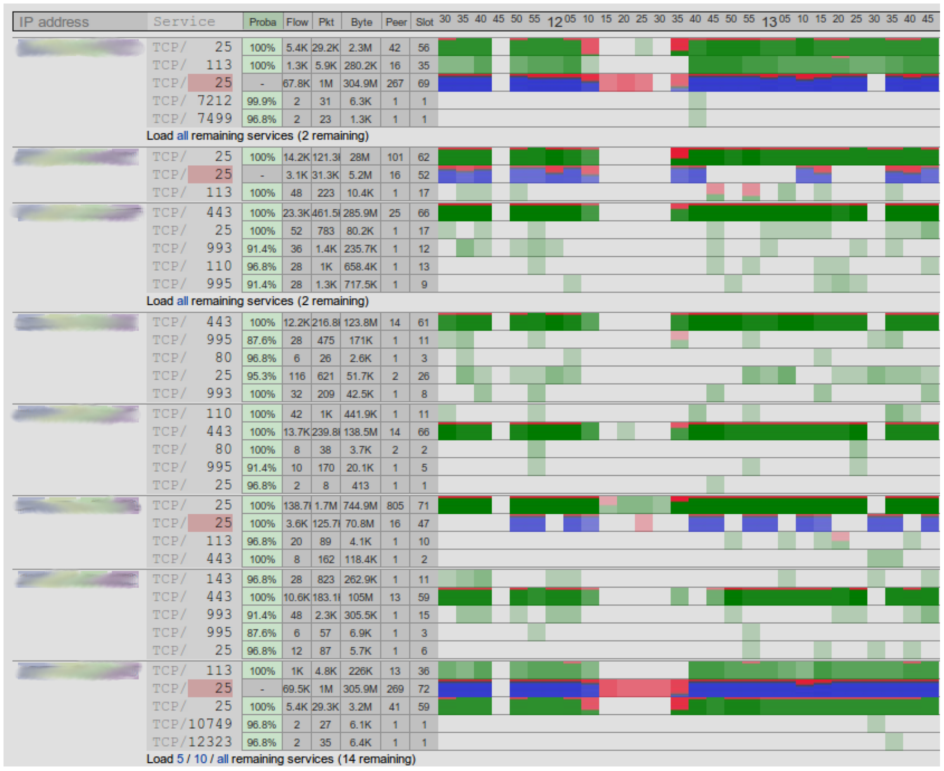
Nfsight can be used to rapidly identify the population of internal servers. The passive service detection algorithm identifies servers actively used in the organization network. Through the front-end, operators can query monthly, weekly, or daily network activity by port number. For example, one can query all internal IP addresses hosting a VNC server (port TCP/5900), and display the daily average number of peers each of the IP addresses has been connected to over the past few weeks. The dashboard also provides the top 20 hosted services ranked by the number of internal servers.

3.1.2 Network Monitoring

In addition to filtering activity by port, one can query activity by subnet to check for anomalies in a specific part of the network. An example of anomaly is the loss of network connectivity for a set of hosts. We illustrate this case in Figure 8, which represents the effect of a power outage from the perspective of both the servers which lost power (activity in green) and the clients which could no longer reach the servers (activity in red). The visualization provided by Nfsight makes it easy to determine the duration of the event (it started at 12:10 PM and activity was fully restored at 12:40 PM) and the list of internal hosts affected.

3.1.3 Policy Checking

In most organizations, critical subnets are subject to a tight security policy to prevent exposure of sensitive hosts. Nfsight can be used to check that these policies are properly implemented and are not compromised. The front-end Visualizer organizes assets per IP address and service, providing the operators an instant view to detect rogue hosts or rogue services. A watchlist allows one to register hosts with a service profile and be alerted when an unknown service is detected. For example, the pro-



IP address	Service	Proba	Flow	Pkt	Byte	Peer	Slot
[redacted]	TCP/ 25	100%	5.4K	29.2K	2.3M	42	56
	TCP/ 113	100%	1.3K	5.9K	280.2K	16	35
	TCP/ 25	-	67.8K	1M	304.9M	267	69
	TCP/ 7212	99.9%	2	31	6.3K	1	1
	TCP/ 7499	96.8%	2	23	1.3K	1	1
	Load all remaining services (2 remaining)						
[redacted]	TCP/ 25	100%	14.2K	121.3	28M	101	62
	TCP/ 25	-	3.1K	31.3K	5.2M	16	52
	TCP/ 113	100%	48	223	10.4K	1	17
[redacted]	TCP/ 443	100%	23.3K	461.5	285.9M	25	66
	TCP/ 25	100%	52	783	80.2K	1	17
	TCP/ 993	91.4%	36	1.4K	235.7K	1	12
	TCP/ 110	96.8%	28	1K	658.4K	1	13
	TCP/ 995	91.4%	28	1.3K	717.5K	1	9
	Load all remaining services (2 remaining)						
[redacted]	TCP/ 443	100%	12.2K	216.8	123.8M	14	61
	TCP/ 995	87.6%	28	475	171K	1	11
	TCP/ 80	96.8%	6	26	2.6K	1	3
	TCP/ 25	95.3%	116	621	51.7K	2	26
	TCP/ 993	100%	32	209	42.5K	1	8
[redacted]	TCP/ 110	100%	42	1K	441.9K	1	11
	TCP/ 443	100%	13.7K	239.8	138.5M	14	66
	TCP/ 80	100%	8	38	3.7K	2	2
	TCP/ 995	91.4%	10	170	20.1K	1	5
	TCP/ 25	96.8%	2	8	413	1	1
[redacted]	TCP/ 25	100%	138.7	1.7M	744.9M	805	71
	TCP/ 25	100%	3.6K	125.7	70.8M	16	47
	TCP/ 113	96.8%	20	89	4.1K	1	10
	TCP/ 443	100%	8	162	118.4K	1	2
[redacted]	TCP/ 143	96.8%	28	823	262.9K	1	11
	TCP/ 443	100%	10.6K	183.1	105M	13	59
	TCP/ 993	91.4%	48	2.3K	305.5K	1	15
	TCP/ 995	87.6%	6	57	6.9K	1	3
	TCP/ 25	96.8%	12	87	5.7K	1	6
[redacted]	TCP/ 113	100%	1K	4.8K	226K	13	36
	TCP/ 25	-	69.5K	1M	305.9M	269	72
	TCP/ 25	100%	5.4K	29.3K	3.2M	41	59
	TCP/10749	96.8%	2	27	6.1K	1	1
	TCP/12323	96.8%	2	35	6.4K	1	1
	Load 5 / 10 / all remaining services (14 remaining)						

Figure 8: Effect of a power outage on connectivity

file for an email server could consist of three services: TCP/25 (mail), TCP/143 (IMAP), and TCP/993 (IMAP over SSL). Any additional open port detected on this host would raise an alert automatically. This functionality can also be achieved by active scanning tools such as PBNJ [24], but the passive approach provided by Nfsight is less intrusive and offers a continous view of the service activity.

3.2 Malicious Activity

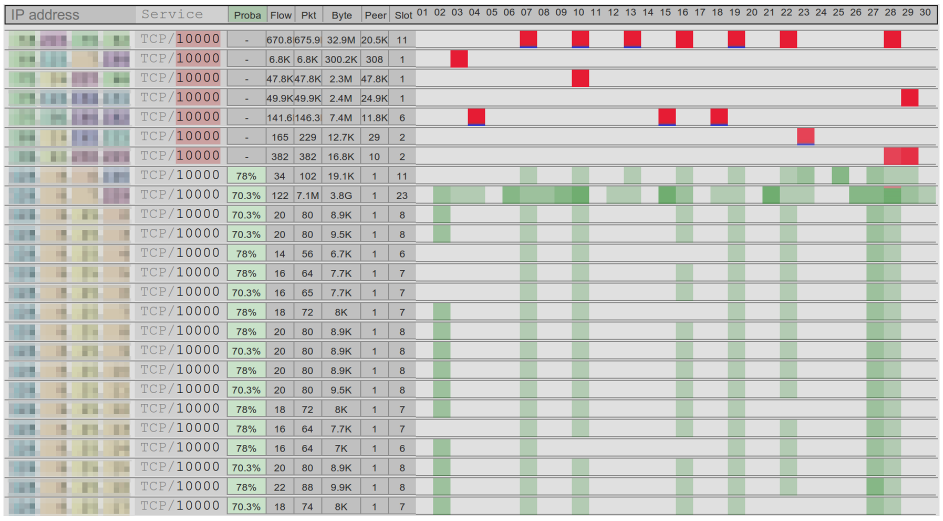
3.2.1 Scanning Activity and Vulnerable Servers

The filtering features of the front-end Visualizer allows one to query external clients generating unidirectional flows. These clients are often scanners targeting the organization IP addresses randomly or sequentially, and trying to find open services to compromise. As shown in Figure 9, the dashboard of Nfsight also provides the top 20 probed services ranked by number of scanners. Operators can click on a service to display the details of the scanning activity and more importantly, the list of internal hosts that scanners were able to find. This information is critical when a new vulnerability linked to a specific service is discovered, because security administrators can use Nfsight to learn, first, if attackers are actively trying to exploit it, and, second, what are the internal hosts that potentially need to be patched or closed.

Figure 10 illustrates this feature by showing the activity for port TCP/10000 over a period of 19 days. This port is known to host the Webmin application, which has been vulnerable to remote exploits [34]. We can see two parts in Figure 10: the top part in red shows external hosts scanning the organization network to find vulnerable applications on port TCP/10000. The bottom part in green represents internal hosts listening on port TCP/10000. The coloring is automatic based on the number of unanswered unidirectional flows (red) versus valid bidirectional flows (green). Moreover, the average number of peers displayed for each end point in the metric section clearly discriminates scanning activity (between 16 and 27,200 peers scanned per day) and server activity (1 client on average per day).

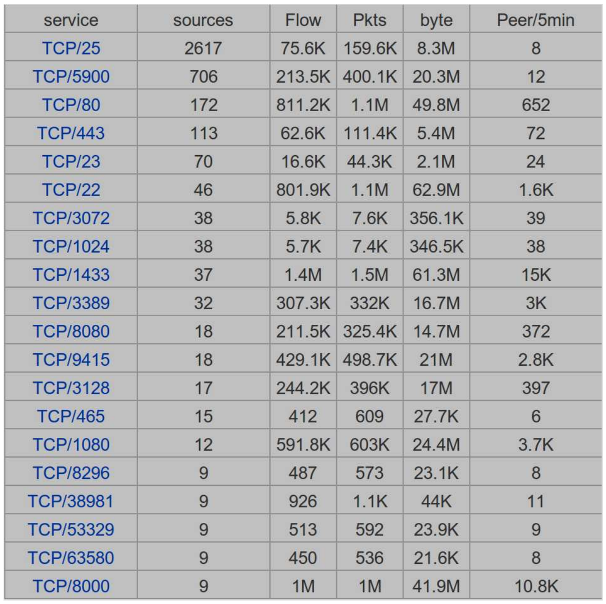
3.2.2 Compromised Hosts

In addition to external scanners, Nfsight can detect and display internal hosts generating an abnormal volume of unidirectional flows. These hosts are often compromised by a malware that tries to spread. The Intrusion Detector notifies the operators by means of automatically generated alarms when such a host is observed in the network. As described in Section 2.3.3, each alert contains a link that shows the service activity detected by Nfsight and



IP address	Service	Proba	Flow	Pkt	Byte	Peer	Slot
[redacted]	TCP/10000	-	670.8	675.9	32.9M	20.5K	11
[redacted]	TCP/10000	-	6.8K	6.8K	300.2K	308	1
[redacted]	TCP/10000	-	47.8K	47.8K	2.3M	47.8K	1
[redacted]	TCP/10000	-	49.9K	49.9K	2.4M	24.9K	1
[redacted]	TCP/10000	-	141.6	146.3	7.4M	11.8K	6
[redacted]	TCP/10000	-	165	229	12.7K	29	2
[redacted]	TCP/10000	-	382	382	16.8K	10	2
[redacted]	TCP/10000	78%	34	102	19.1K	1	11
[redacted]	TCP/10000	70.3%	122	7.1M	3.8G	1	23
[redacted]	TCP/10000	70.3%	20	80	8.9K	1	8
[redacted]	TCP/10000	70.3%	20	80	9.5K	1	8
[redacted]	TCP/10000	78%	14	56	6.7K	1	6
[redacted]	TCP/10000	78%	16	64	7.7K	1	7
[redacted]	TCP/10000	70.3%	16	65	7.7K	1	7
[redacted]	TCP/10000	78%	18	72	8K	1	7
[redacted]	TCP/10000	78%	20	80	8.9K	1	8
[redacted]	TCP/10000	70.3%	20	80	8.9K	1	8
[redacted]	TCP/10000	78%	20	80	8.9K	1	8
[redacted]	TCP/10000	70.3%	20	80	9.5K	1	8
[redacted]	TCP/10000	78%	18	72	8K	1	7
[redacted]	TCP/10000	78%	16	64	7.7K	1	7
[redacted]	TCP/10000	78%	16	64	7K	1	6
[redacted]	TCP/10000	70.3%	20	80	8.9K	1	8
[redacted]	TCP/10000	78%	22	88	9.9K	1	8
[redacted]	TCP/10000	70.3%	18	74	8K	1	7

Figure 10: Scanners targeting port 10000 and internal servers hosting a service on this port



service	sources	Flow	Pkts	byte	Peer/5min
TCP/25	2617	75.6K	159.6K	8.3M	8
TCP/5900	706	213.5K	400.1K	20.3M	12
TCP/80	172	811.2K	1.1M	49.8M	652
TCP/443	113	62.6K	111.4K	5.4M	72
TCP/23	70	16.6K	44.3K	2.1M	24
TCP/22	46	801.9K	1.1M	62.9M	1.6K
TCP/3072	38	5.8K	7.6K	356.1K	39
TCP/1024	38	5.7K	7.4K	346.5K	38
TCP/1433	37	1.4M	1.5M	61.3M	15K
TCP/3389	32	307.3K	332K	16.7M	3K
TCP/8080	18	211.5K	325.4K	14.7M	372
TCP/9415	18	429.1K	498.7K	21M	2.8K
TCP/3128	17	244.2K	396K	17M	397
TCP/465	15	412	609	27.7K	6
TCP/1080	12	591.8K	603K	24.4M	3.7K
TCP/8296	9	487	573	23.1K	8
TCP/38981	9	926	1.1K	44K	11
TCP/53329	9	513	592	23.9K	9
TCP/63580	9	450	536	21.6K	8
TCP/8000	9	1M	1M	41.9M	10.8K

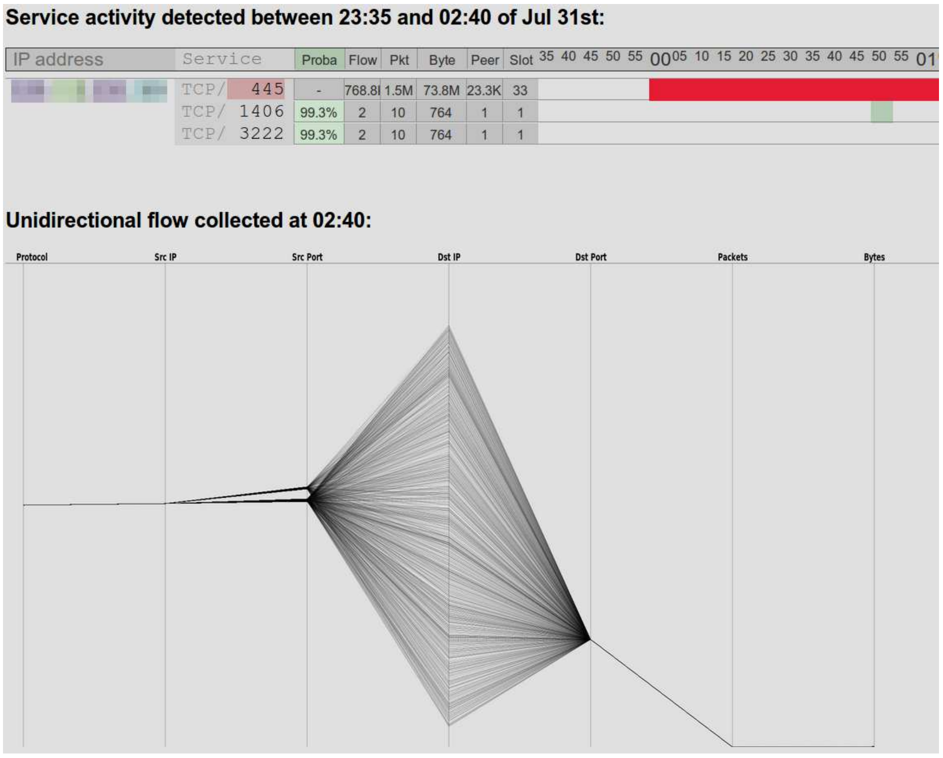
Figure 9: Top 20 scanned services

the details of flows related to the event. Consequently, operators can check if these alerts are due to malicious behavior or normal server behavior.

Figure 11 illustrates the activity of an internal host which was compromised and started at midnight to send a massive number of probes to random destination IP addresses on port TCP/445. Nfsight provides information about the scanning rate, on average 23,300 IP every 5 minutes, and the uniform distribution of targets from the parallel plot provided by Picviz [33]. Security administrators who tested Nfsight indicated that they cannot configure their IPS devices to detect and block this type of massive scanning activity, because the IPS devices would be at risk of becoming overloaded. Therefore, Nfsight complements other security solutions by leveraging NetFlow for scalable security monitoring.

3.2.3 Distributed Attacks

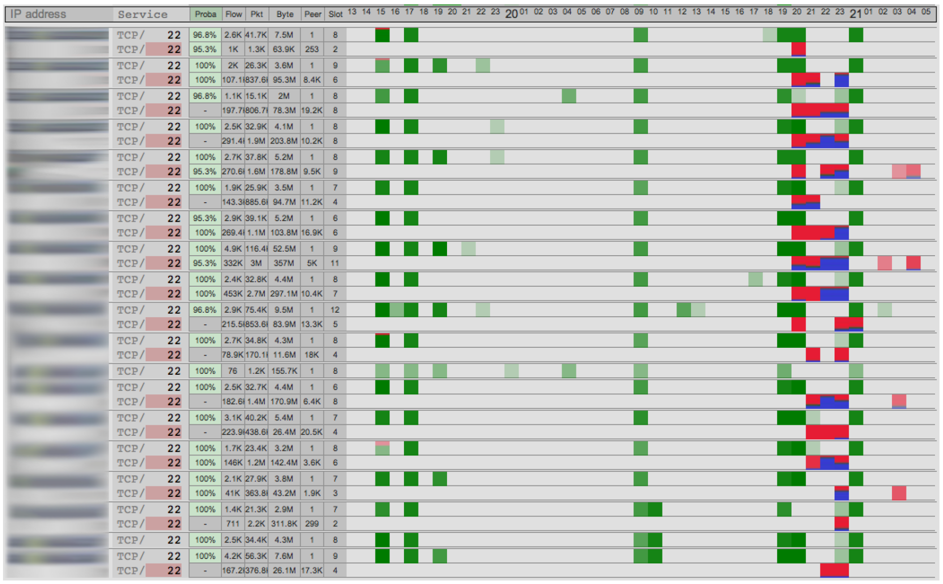
The visualization feature of Nfsight enables security administrators to identify coordinated attacks and to understand their scope. An example of a distributed scan originating from a set of internal SSH servers is provided in Figure 12. A total of 19 servers were compromised because the password for one shared account was determined through brute-force attack. Attackers installed a remote control software on each host and then launched a distributed scan at 8 PM to find additional SSH servers to compromise. The timeseries representation and the distinction between client/server activity allows administrators to immediately see the coordinated nature of the attack.


Service activity detected between 23:35 and 02:40 of Jul 31st:

IP address	Service	Proba	Flow	Pkt	Byte	Peer	Slot
[redacted]	TCP/ 445	-	768.8	1.5M	73.8M	23.3K	33
	TCP/ 1406	99.3%	2	10	764	1	1
	TCP/ 3222	99.3%	2	10	764	1	1

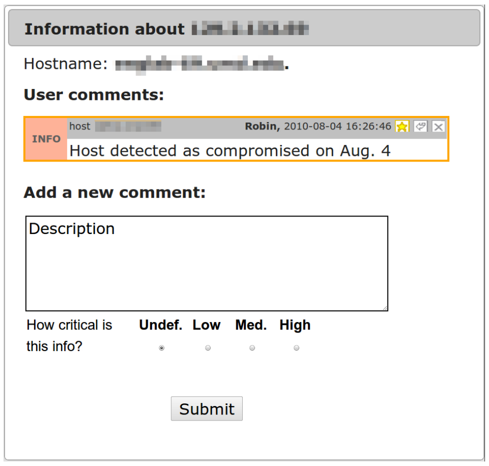
Unidirectional flow collected at 02:40:

Figure 11: Compromised internal host scanning a large range of destination IP on port TCP/445 (Netbios service)



IP address	Service	Proba	Flow	Pkt	Byte	Peer	Slot
[redacted]	TCP/ 22	96.8%	2.6K	41.7K	7.5M	1	8
[redacted]	TCP/ 22	95.3%	1K	1.3K	63.9K	253	2
[redacted]	TCP/ 22	100%	2K	26.3K	3.6M	1	9
[redacted]	TCP/ 22	100%	107.1	837.6	95.3M	8.4K	6
[redacted]	TCP/ 22	96.8%	1.1K	15.1K	2M	1	8
[redacted]	TCP/ 22	-	197.7	806.7	78.3M	19.2K	8
[redacted]	TCP/ 22	100%	2.5K	32.9K	4.1M	1	8
[redacted]	TCP/ 22	-	291.4	1.9M	203.8M	10.2K	8
[redacted]	TCP/ 22	100%	2.7K	37.8K	5.2M	1	8
[redacted]	TCP/ 22	95.3%	270.6	1.6M	178.8M	9.5K	9
[redacted]	TCP/ 22	100%	1.9K	25.9K	3.5M	1	7
[redacted]	TCP/ 22	-	143.3	885.6	94.7M	11.2K	4
[redacted]	TCP/ 22	95.3%	2.9K	39.1K	5.2M	1	6
[redacted]	TCP/ 22	100%	269.4	1.1M	103.8M	16.9K	6
[redacted]	TCP/ 22	100%	4.9K	116.4	52.5M	1	9
[redacted]	TCP/ 22	95.3%	332K	3M	357M	5K	11
[redacted]	TCP/ 22	100%	2.4K	32.8K	4.4M	1	8
[redacted]	TCP/ 22	100%	453K	2.7M	297.1M	10.4K	7
[redacted]	TCP/ 22	96.8%	2.9K	75.4K	9.5M	1	12
[redacted]	TCP/ 22	-	215.5	853.6	83.9M	13.3K	5
[redacted]	TCP/ 22	100%	2.7K	34.8K	4.3M	1	8
[redacted]	TCP/ 22	-	78.9K	170.1	11.6M	18K	4
[redacted]	TCP/ 22	100%	76	1.2K	155.7K	1	8
[redacted]	TCP/ 22	100%	2.5K	32.7K	4.4M	1	6
[redacted]	TCP/ 22	-	182.6	1.4M	170.9M	6.4K	8
[redacted]	TCP/ 22	100%	3.1K	40.2K	5.4M	1	7
[redacted]	TCP/ 22	-	223.9	438.6	26.4M	20.5K	4
[redacted]	TCP/ 22	100%	1.7K	23.4K	3.2M	1	8
[redacted]	TCP/ 22	100%	146K	1.2M	142.4M	3.6K	6
[redacted]	TCP/ 22	100%	2.1K	27.9K	3.8M	1	7
[redacted]	TCP/ 22	100%	41K	363.8	43.2M	1.9K	3
[redacted]	TCP/ 22	100%	1.4K	21.3K	2.9M	1	7
[redacted]	TCP/ 22	-	711	2.2K	311.8K	299	2
[redacted]	TCP/ 22	100%	2.5K	34.4K	4.3M	1	8
[redacted]	TCP/ 22	100%	4.2K	56.3K	7.6M	1	9
[redacted]	TCP/ 22	-	167.2	376.8	26.1M	17.3K	4

Figure 12: Set of 19 compromised SSH servers remotely controlled (server activity in green) and launching a synchronized distributed scan towards port TCP/22 (client activity in blue and red)


Information about [redacted]

Hostname: [redacted].

User comments:

INFO host [redacted] **Robin,** 2010-08-04 16:26:46

Host detected as compromised on Aug. 4

Add a new comment:

Description

How critical is this info? **Undef.** **Low** **Med.** **High**

Figure 13: User comment window for information sharing about a specific host

3.3 Forensic and Collaboration

The different case studies described previously show that Nfsight can be efficiently used to perform forensic tasks. The overview representation and detail-on-demand capability offer a fast and easy solution to understand what happened in the network. This functionality is augmented by several collaboration features. First, operators can click on any IP address or service to leave a comment and rate its criticality (low, medium or high). The comment window is illustrated in Figure 13. Second, email alerts contain links that the operators can use to rate the alert as true positive, false positive, or unknown. The web page displayed after clicking on these links allows operators to write a comment and rate the criticality of the alert. These comments are displayed on the dashboard of Nfsight and colored by criticality. Operators can reply to comments left by others and share their finding or expertise.

3.4 Limitations and Future Work

Nfsight provides a practical network situational awareness solution based on NetFlow flows. The main contributions are 1) passive service discovery, 2) intrusion detection and 3) automated alert and visualization. We showed with different use cases how Nfsight can help network administrators and security operators in their monitoring tasks. However, Nfsight has still important limitations that we plan to address in our future work.

First, Nfsight works with non-sampled flows. We note that results from other evaluations of passive detection techniques indicate that sampling has a limited impact on the overall accuracy. For example, [1] reports that capturing only 16% of the data results only in an 11% drop in discovered servers. However, we believe that random flow sampling will likely break our algorithm for identifying bidirectional flows. We plan on assessing the effect of sampling on the detection accuracy of the different heuristics. Furthermore, asymmetric routing can challenge our approach. Specifically, we assumed in this study that NetFlow collectors covered the pathways for both requests and replies. In some organization networks, replies and requests can sometimes take different routes for which there is no NetFlow collector deployed and therefore, we would not be able to pair the unidirectional flows into bidirectional flows.

We also note that Nfsight works at the network layer and therefore heavily relies on port numbers. As a consequence, it can be difficult or impossible for a network operator to identify the application behind a service detected by Nfsight. This issue arises from the fact that some applications use random ports or hide behind well-known ports. For example Skype is famous for using port 80 or port 443, normally reserved to web traffic, in order to evade firewall protection. Related work [6] on flow-based traffic classification proved that it is possible to accurately identify applications using only NetFlow. We plan on developing additional heuristics for Nfsight to be able to classify traffic regardless of the port numbers used. These heuristics can work on 1) relationships between flow characteristics, such as the ratio between number of packets and number of bytes or the time distribution of flows, and 2) relationships between hosts. We believe that discovering communication patterns between hosts would be critical to identify not only applications but also large communication structures such as those used by P2P networks or botnets.

Finally, the current intrusion detection rules are rudimentary and the fact that most of them are threshold-based means that they are prone to generate a significant volume of false positives. We implemented a feedback mechanism to leverage human expertise and facilitate the task of tuning the detection rules, but this process still involves important manual development. We plan to automate this task and integrate a machine-learning approach to create and tune rules based on samples of true and false positives.

4 Related Work

NetFlow is highly popular among network operators and researchers because it offers a comprehensive view of network activity while being scalable and easy to deploy in large networks. As a result, an important number of tools and publications have been produced over the past decade, as shown by [28] and [17]. We present in this section an overview of these resources organized according to our areas of interests: Netflow processing and visualization, and service detection.

4.1 NetFlow Processing and Visualization Applications

Working with NetFlow is a multi-step process. First, flow records are generated by a compatible network device, typically a router, or by a software probe such as [29, 21, 36]. These flows are then sent over the network in UDP packets to collectors according to the NetFlow protocol. The role of a collector is to store flow records in flat files or in a database. The collector is often linked to a set of processing tools to allow a network operator to read and filter flow records. Processing tools include CAIDA Cflowd [2], OSU flow-tools [27], SiLK [8] and more recently Nfdump [18].

In addition to command line tools, several graphical user interfaces exist to visualize and query network activity. NTOP [22] and Nfsen [10] are two popular solutions that provide a web interface to network operators. We note that we developed Nfsight as a plugin of Nfsen because of its simplicity, extensibility and processing capability.

An important body of research has been conducted on the topic of NetFlow visualization. The NCSA research center at the University of Illinois produced NvisionIP [16] and VisFlowConnect [38]. NvisionIP provides a two-dimensional map to visualize the network characteristics of up to 65,536 hosts in a single view. It has been extended to include a graphical filtering rule system [15] to allow operators to easily spot abnormal activity. VisFlowConnect offers a parallel-plot view with drill-down features. Compared to Nfsight, the main limitation of these two tools is that they work offline, while our solution processes NetFlow flows in near real time.

Researchers at the University of Wisconsin developed FlowScan [25] and NetPY [3]. NetPY is an interactive visualization application written in Python on top of flow-tools. It provides an automated sampling algorithm and enables operators to understand how network traffic is used through heatmaps, timeseries and hierachical heavy hitters plots. FlowScan works at a higher level by providing traffic volume graphs of network applications. The architecture of FlowScan, which consists of Perl scripts and uses RRDTool, is very similar to the architecture of Nfsen. Also, Nfsight shares with FlowScan the idea of using heuristics to classify flow records. However, FlowScan lacks alerting capabilities and does not determine client/server relationships.

Other research projects on the topic of flow visualization include FloVis [31], VIAssist [5] and NFlowVis [7]. FloVis offers a set of modules such as Overflow [9] and NetByte Viewer [30] to display the same network activity through different perspectives in order to gain a better understanding of host behavior. VIAssist and NFlowVis adopt the same objective with drill-down features and multiple visualization techniques. NFlowVis integrates state-of-the art plots by making use of treemap and a hierachical edge bundle view. Similarly to Nfsight, VIAssist offers collaboration features to allow operators to share items of interest and to communicate findings. We note that none of these three visualization frameworks are publicly available.

4.2 Service Detection and Bidirectional Flows

Solutions for service discovery can be divided into active and passive techniques. Active techniques send network probes to a set of targets to check the presence of any listening service, while passive techniques extract information about services from network sniffing devices. A well-known open source active scanner is Nmap [20]. The drawbacks of active techniques are: 1) they provide only a snapshot in time of the network, 2) they cannot detect services protected by firewalls, 3) they are intrusive and not scalable, and 4) aggressive scanning may also cause system and network disruptions or outages [35, 1]. Passive solutions offer a continuous view of the network, their results are not impacted by firewalls, and they are highly scalable. The main limitation of the passive approaches is that they detect only active services, i.e., any unused services with no incoming traffic cannot be discovered. However, by providing a low overhead continuous passive discovery approach, services that do communicate will be detected. A well-known open source passive service detector working on packet data is Pads [23].

A passive and accurate detection of network services working on network flows would be trivial with bidirectional flows where request flows initiated by clients and reply flows initiated by servers can be easily identified. However, most organization networks are currently instrumented with traditional unidirectional flow solutions such as NetFlow, and they lack the capability to generate and collect bidirectional flows. This motivated us to design a solution based only on unidirectional flow. We note that the IPFIX IETF working group has recently introduced a new standard format to export network flows based on NetFlow version 9 [4], which includes the capability to export bidirectional flows generated directly at the measurement interface [32]. We see our approach as a robust intermediate solution between the current large scale deployment of NetFlow, which is unidirectional, and the future implementation by router vendors and deployment by organization networks of IPFIX, which can be bidirectional.

Rwmatch from SiLK [8] shares the same motivation of generating correctly oriented bidirectional network flows from unidirectional flows. Rwmatch uses two heuristics to decide on the orientation of bidirectional flows: timestamp of request and reply flows, and server port number being below 1024. However, we have observed that both of these heuristics can be fallible by themselves. Therefore, we use five additional heuristics and combine heuristic outputs through Bayesian inference in order to improve the accuracy of server detection over time. We note that another tool similar to rwmatch called flow-connect, developed as part of the OSU Flow-tools framework, has been suggested in [27] but has actually never been implemented.

Finally, two alternative approaches YAF from CERT [36] and Argus [26] generate bidirectional flows not from unidirectional flows but from packet data. Both tools work by processing packet data from PCAP dump files or directly from a network interface, and then export bidirectional flows following the IPFIX format.

5 Conclusion

Timely information on what is occurring in their networks is crucial for network and security administrators. Nfsight provides an easy to use graphical tool for administrators to gain knowledge on the set of services running in their networks, as well as on any anomalous activities. Nfsight is non-intrusive since it relies on passively collected NetFlow data, provides a near real-time report on network activities, allows data to be viewed at different time granularities, and supports collaboration between system administrators. Nfsight uses a combination of heuristics and Bayesian inference to identify services and graphlet-based technique to detect intrusions. In this paper, we described the architecture and heuristics used by Nfsight, evaluated its accuracy in service discovery, and presented a number of real use-cases. Our future work includes development and evaluation of additional server discovery heuristics. We also plan to revise the intrusion detection rules and to complete the implementation of the feedback mechanism to adjust detection thesholds automatically.

6 Acknowledgments

We would like to thank Gerry Sneeringer, Kevin Shivers and Bertrand Sobesto for their ideas and their help labeling and investigating malicious activity. We are also grateful to Virginie Klein for her contribution on the Intrusion Detector. Finally we would like to thank William Sanders, Jenny Applequist, Danielle Chrun, Eser Kandogan and the anonymous reviewers for their guidance and comments on the paper.

7 Availability

The documentation and the source code of Nfsight are freely available at:

`http://nfsight.research.att.com`

References

[1] BARTLETT, G., HEIDEMANN, J., AND PAPADOPOULOS, C. Understanding passive and active service discovery. In *Proc. 7th ACM Internet Measurement Conference* (2007), pp. 55–60.

[2] Caida cflowd. http://www.caida.org/tools/measurement/cflowd/, 2010.

[3] CIRNECI, A., BOBOC, S., LEORDEANU, C., CRISTEA, V., AND ESTAN, C. Netpy: Advanced Network Traffic Monitoring. In *Proc. Int Conf. on Intelligent Networking and Collaborative Systems (INCOS'09)* (2009), pp. 253–254.

[4] CLAISE, B., QUITTEK, J., BRYANT, S., AITKEN, P., MEYER, J., TRAMMELL, B., BOSCHI, E., WENGER, S., CHANDRA, U., WESTERLUND, M., ET AL. RFC 5101 Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information, 2008.

[5] D'AMICO, A., GOODALL, J., TESONE, D., AND KOPYLEC, J. Visual discovery in computer network defense. *IEEE Computer Graphics and Applications 27*, 5 (2007), 20–27.

[6] ERMAN, J., MAHANTI, A., ARLITT, M., AND WILLIAMSON, C. Identifying and discriminating between web and peer-to-peer traffic in the network core. In *Proceedings of the 16th international conference on World Wide Web* (2007), ACM, p. 892.

[7] FISCHER, F., MANSMANN, F., KEIM, D., PIETZKO, S., AND WALDVOGEL, M. Large-scale network monitoring for visual analysis of attacks. In *Proc. Workshop on Visualization for Computer Security (VizSEC)* (2008), Springer, p. 111.

[8] GATES, C., COLLINS, M., DUGGAN, M., KOMPANEK, A., AND THOMAS, M. More NetFlow tools: For performance and security. In *Proc. 18th USENIX Large Installation System Administration Conf. (LISA)* (2004), pp. 121–132.

[9] GLANFIELD, J., BROOKS, S., TAYLOR, T., PATERSON, D., SMITH, C., GATES, C., AND MCHUGH, J. OverFlow: An Overview Visualization for Network Analysis. In *Proc. 6th Int. Workshop on Visualization for Cyber Security (VizSec)* (2009), pp. 11–19.

[10] HAAG, P. Watch your Flows with NfSen and NFDUMP. In *50th RIPE Meeting* (2005).

[11] HUGHES, E., AND SOMAYAJI, A. Towards network awareness. In *Proc. 19th USENIX Large Installation System Administration Conf. (LISA)* (2005), pp. 113–124.

[12] Iana assigned port numbers. http://www.iana.org/assignments/port-numbers, 2010.

[13] KARAGIANNIS, T., PAPAGIANNAKI, K., AND FALOUTSOS, M. BLINC: multilevel traffic classification in the dark. In *Proc. ACM SIGCOMM Conference* (2005), pp. 229–240.

[14] KIM, H., CLAFFY, K., FOMENKOV, M., BARMAN, D., FALOUTSOS, M., AND LEE, K. Internet traffic classification demystified: myths, caveats, and the best practices. In *Proceedings of the 2008 ACM CoNEXT conference* (2008), ACM, pp. 1–12.

[15] LAKKARAJU, K., BEARAVOLU, R., SLAGELL, A., YURCIK, W., AND NORTH, S. Closing-the-loop in NVisionIP: Integrating discovery and search in security visualizations. In *Proc. IEEE Workshop on Visualization for Computer Security (VizSEC)* (2005).

[16] LAKKARAJU, K., YURCIK, W., AND LEE, A. NVisionIP: netflow visualizations of system state for security situational awareness. In *Proc. ACM Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC)* (2004), pp. 65–72.

[17] LEINEN, S. FloMA: Pointers and Software, NetFlow. Tech. rep., SWITCH, 2010.

[18] Nfdump. http://nfdump.sourceforge.net, 2010.

[19] Nfsen. http://nfsen.sourceforge.net, 2010.

[20] Nmap. http://www.nmap.org, 2010.

[21] Nprobe: Netflow/ipfix network probe. http://www.ntop.org/nProbe.html, 2010.

[22] Ntop: Network traffic probe. http://www.ntop.org, 2010.

[23] Pads. http://passive.sourceforge.net, 2010.

[24] Pbnj. http://pbnj.sourceforge.net, 2010.

[25] PLONKA, D. Flowscan: A Network Traffic Flow Reporting and Visualization Tool. In *Proc. 14th USENIX Large Installation System Administration Conf. (LISA)* (2000), pp. 305–318.

[26] QOSIENT, L. Argus: Network Audit Record Generation and Utilization System.

[27] ROMIG, S., FULLMER, M., AND LUMAN, R. The OSU flow-tools package and CISCO NetFlow logs. In *Proc. 14th USENIX Large Installation System Administration Conf, (LISA)* (2000), pp. 291–304.

[28] SO-IN, C. A Survey of Network Traffic Monitoring and Analysis Tools. Cse 576m computer system analysis project, Washington University in St. Louis, 2009.

[29] Softflowd: fast software netflow probe. http://www.mindrot.org/projects/softflowd/, 2010.

[30] TAYLOR, T., BROOKS, S., AND MCHUGH, J. NetBytes viewer: An entity-based netflow visualization utility for identifying intrusive behavior. pp. 101–114.

[31] TAYLOR, T., PATERSON, D., GLANFIELD, J., GATES, C., BROOKS, S., AND MCHUGH, J. FloVis: Flow Visualization System. In *Proc. Cybersecurity Applications and Technologies Conference for Homeland Security (CATCH)* (2009), pp. 186–198.

[32] TRAMMELL, B., AND BOSCHI, E. RFC 5103: Bidirectional Flow Export Using IP Flow Information Export (IPFIX), 2008.

[33] TRICAUD, S., AND SAADÉ, P. Applied parallel coordinates for logs and network traffic attack analysis. *Journal in computer virology 6*, 1 (2010), 1–29.

[34] Webmin vulnerability, cve-2006-3392, 2006.

[35] WEBSTER, S., LIPPMANN, R., AND ZISSMAN, M. Experience using active and passive mapping for network situational awareness. In *Proc. 5th IEEE Int. Symp. on Network Computing and Applications (NCA)* (2006), pp. 19–26.

[36] Yaf. http://tools.netsa.cert.org/yaf/, 2010.

[37] YURCIK, W. Visualizing NetFlows for security at line speed: the SIFT tool suite. In *Proc. 19th Large Installation System Administration Conf. (LISA)* (2005), USENIX, pp. 169–176.

[38] YURCIK, W. VisFlowConnect-IP: a link-based visualization of Netflows for security monitoring. In *18th Annual FIRST Conf. on Computer Security Incident Handling* (2006).

Using Syslog Message Sequences for Predicting Disk Failures

R. Wesley Featherstun and Errin W. Fulp
Department of Computer Science
Wake Forest University

August 24, 2010

Abstract

Mitigating the impact of computer failure is possible if accurate failure predictions are provided. Resources, and services can be scheduled around predicted failure and limit the impact. Such strategies are especially important for multi-computer systems, such as compute clusters, that experience a higher rate of failure due to the large number of components. However providing accurate predictions with sufficient lead time remains a challenging problem.

This research uses a new spectrum-kernel Support Vector Machine (SVM) approach to predict failure events based on system log files. These files contain messages that represent a change of system state. While a single message in the file may not be sufficient for predicting failure, a sequence or pattern of messages may be. This approach uses a sliding window (sub-sequence) of messages to predict the likelihood of failure. Then, a frequency representation of the message sub-sequences observed are used as input to the SVM. The SVM associates the messages to a class of failed or non-failed system. Experimental results using actual system log files from a Linux-based compute cluster indicate the proposed spectrum-kernel SVM approach can predict hard disk failure with an accuracy of 80% about one day in advance.

1 Introduction

Clusters are quickly growing in size in terms of both computing power and storage space. It is predicted that by 2018, large systems could have over 800,000 disks. Out of these 800,000 disks, it is possible that 300 of them may be in a failure state at any given time [13]. Since multicore processors are becoming more prevalent, even one disk being unavailable means that multiple processors may be unable to perform their work. Assuming one could predict these failure events, the distribution of work on the cluster could be altered to avoid effected disks before they failed or jobs could be paused while the necessary data is backed up.

Much work has been done in the field of hardware failure predictions. Hammerly *et al.* [5] used a naive Bayesian classifier on SMART data and managed to predict disk failures that would occur in the next 48 hours with 52% accuracy. A team from IBM [8] used data from a specialized logging system on its BlueGene cluster. While the team achieved high accuracy, its data set may be too specialized to be of use by the general public. Peter Broadwell [2] used a supervised Bayesian approach to predict SCSI cable failures. While he was able to create an effective prediction method, the approach presented in the paper is not scalable. Murray *et al* [11] compared the effectiveness of data mining techniques such as SVMs, clustering, and a rank-sum test for failure predictions using SMART data. Finally, Turnbull *et al* [16] proposed an approach similar to the one presented in this paper. However, Turnbull focused on predicting system board failures instead of disk failures.

The prediction process in this paper uses the `syslog` event logging service as data to predict failure events. The `syslog` facility is common to all Linux distributions as well as Unix variants. Using blocks of `syslog` entry tag and message strings, a Support Vector Machine [3] creates a model of the data which isolates patterns of log information that indicate future disk failures. The main focus of this approach is to predict disk failures at least one day before the failure. One day's notice gives administrators enough time to make any necessary changes to the scheduling process or ensure that they can obtain another hard drive of the correct model [4].

The remainder of this paper is organized as follows. Section 2 provides a description of the Unix `syslog` facility and SMART messages. Section 3 describes the approach to failure predictions proposed in this paper. Section 4 describes the Support Vector Machine data mining technique while section 5 discusses experimental results. Finally, section 6 summarizes this paper and discusses some areas for future work.

2 System Log Facilities and Messages

`Syslog` is a standard Unix logging facility, which means that every computer running Linux is able to use `syslog` [9]. The ubiquity of `syslog` means that performing an analysis on `syslog` data allows for the creation of a failure prediction approach which can be used by anyone using a Linux or Unix system. `Syslog` records any change of system state, such as a login or a program failure.

As seen in Table 1, the standard `syslog` message contains six fields [9]. However, the approach in this paper uses only the timestamp, tag number, and message fields. The tag is a numerical representation of the importance of the message. The tag number is an integer, where a lower number indicates a higher importance. For example, a message with a tag number of 1 is more urgent than a tag number of 20. The tag number field corresponds to the priority field in the actual `syslog` packet, whose value is determined by multiplying the facility by 8 and then adding the level. Therefore, it provides a numerical representation of both the facility which posted the message and how important the message is. The time field records the time at which the message was posted, commonly in Linux epoch time. The final field is the message field, which consists of a plain text string of varying length. The message is an explicit description of the associated event. While the other fields only indicate how important the event was and when it took place, the message field tells an observer that the event was, for example, a login attempt or a disk failure [9].

SMART messages record and report information that relates solely to hard disks, such as their current health and performance and is deployed with most modern ATA and SCSI drives [1]. Since SMART disks monitor health and performance information, they are able to report and possibly predict hard drive problems. Some of the attributes monitored by SMART are the current temperature, the number of scan errors, and the number of hours for which a disk has been online. SMART checks the disk's status every 30 minutes and passes along any information regarding the possibility of an upcoming failure to `syslog`. Pinheiro *et al.* have shown that using individual SMART messages to build a prediction model is ineffective [12]. Therefore, the approach in this paper uses all `syslog` data, including, but not limited to, SMART data.

3 Approach

3.1 Sequential Data

As described by Pinheiro *et al.*, single messages are not sufficient for predicting failure [12]. However, examining sequences of messages may be a more effective means of failure predictions. Instead of considering messages in isolation, the approach in this paper analyzes sequences of messages, which provides context for individual messages.

A sliding window approach is used to isolate sequential data. In this method, a window of fixed length, n, is placed at the beginning of the list of data. All of the data that fall in that window is considered to be one sequence. Then, the sliding window is moved forward one item and the next n items are made into a sequence.

The type of information being examined alters how

Host	Facility	Level	Tag	Time	Message
`node226`	`daemon`	`info`	30	1205054912	`ntpd 2555 synchronized to 198.129.149.215, stratum 3`
`node226`	`local4`	`info`	166	1205124722	`xinetd 2221 START: auth pid=23899 from=130.20.248.51`
`node165`	`local3`	`notice`	157	1205308925	`OSLevel Linux m165 2.6.9-42.3sp.JD4smp`
`node165`	`syslog`	`info`	46	1205308925	`syslogd restart.`

Table 1: Example entries from a `syslog` file

the window moves across message boundaries. When classifying based on tag numbers, each tag number represents one message. Therefore, the sliding window indicates the criticality of the last n messages. However, this paper also examines the use of keystrings to predict disk failures. In the case of keystrings, there may be zero, one, or more keystrings in a given message. Since the keystrings are arranged by order of appearance in the logs, a given window can provide context either within a single message or two or more messages.

The spectrum kernel technique was devised by Leslie *et al.* [7] to leverage sequences of data for use with a classifier. For any $k \geq 1$, the k-spectrum of a given input sequence is defined as all of the subsequences of length k that the sequence contains. Given a sequence length k, an alphabet size b and a single member of the alphabet, e, the spectrum kernel representation of a given sequence can be obtained using Equation 1 [15]. The equation must be applied for each letter in the input.

$$f(t) = mod(b * f(t-1), b^k) + e \qquad (1)$$

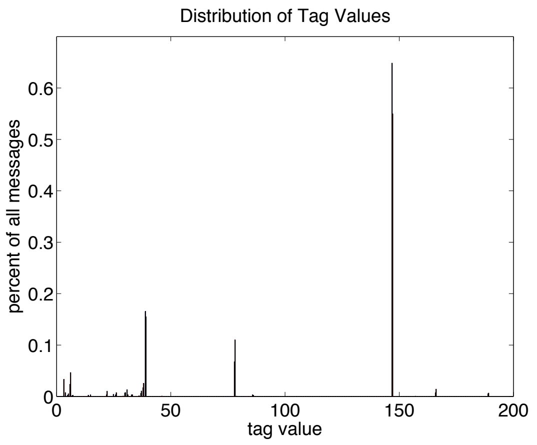
3.2 Tag-Based Features

Consider the tag numbers that occur within a message window. The order in which these messages appear forms a list of tag numbers. From this list of tag numbers, one can create a feature vector which combines two types of features: a count of the number of times each tag number and sequence number appears in a window. For example, the sequence of tag numbers shown in Table 2 correspond to a tag count vector of {40:1, 88:1, 148:2, 158:3, 188:3}.

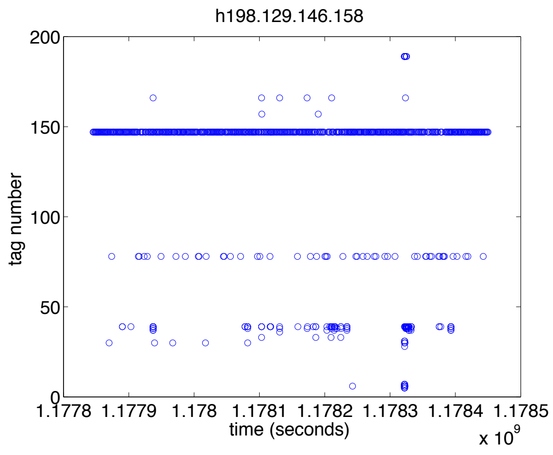
At first, the size of the alphabet is the number of unique tag numbers in `syslog`, which is 191. However, as the maximum tag number seen in the experimental data set is 189, the alphabet size is considered to be 189. Using sequences of length 5, the list of possible features is 189^5, which equates to over 241 billion unique combinations. Computing sequence numbers for all of these combinations will take an excessively long time. Therefore, the alphabet is reduced by assigning multiple tag numbers to a number of 0, 1, or 2 based on the tag's criticality [4].

Tag numbers which are less than or equal to a 10 are considered to be high priority. Tag numbers between 11 and 140 are considered medium priority and tag numbers above 140 are considered low priority. The size of the reduced alphabet and cutoff values are determined by examining the distribution of tag numbers as seen in Figure 2(a). Using an alphabet of size 3 reduces the possible number of features to 243.

Table 2 illustrates the process of assigning criticality scores to each tag number and then determining sequence numbers where $k = 5$. The left hand column contains a list of tag numbers. The middle column shows the sequence of the most recent 5 criticality scores, which are obtained by using the sliding window method and criticality cutoff values described earlier. The criticality score of the current tag number is placed on the righthand side of the criticality sequence. Finally, the righthand column shows the sequence number for the current sequence of criticality scores. The sequence number is determined using Equation 1. While intermediate sequence numbers are calculated for the first $k-1$ sequences, sequence numbers are not recorded until a full k-length sequence has passed. In this example, sequence numbers are only recorded starting at the fifth tag number.



(a) A histogram which indicates the percentages of messages in the data set which contained a given tag number. For example, about 60% of all messages in the data set had a tag number of 149.



(b) An example of the tag number distribution on a given host across time. Each circle represents the tag number for a single `syslog` message.

Figure 1: Illustrations of tag number distribution in the experimental data set

Tag	Translated	Sequence Number
148	2	
148	22	
158	222	
40	2221	
158	22212	239
188	22122	233
188	21222	215
88	12221	160
158	22212	239
188	22122	233

Table 2: An example of a tag list being translated into sequences of criticality scores and then assigned sequence numbers using these criticality scores. For this example, $k = 5$.

3.3 Tags With Timing Information

Timing information is another feature that may help improve failure predictions. The purpose of examining timing information is to discern whether or not a change in message rate can be used to predict failures. During the creation of the sequence numbers, the difference between time of the first message in the sequence and the time of the last message in the sequence is recorded. Doing so provides an indication of how quickly or how slowly those messages were posted. The differences in time are recorded in the same format as the tag and sequence numbers. This information has not been considered in previous work using this method [4].

3.4 Keystring-based Features

The myriad possible `syslog` configurations allow for a set up in which tag numbers are not present [9]. One thing an administrator is unlikely to remove is the message field itself. A string is defined as any space-delineated collection of characters in the message field. For example, a string can be an English word, an IP address, or a number. Tag numbers are not factored into this approach. The goal of this method is to discover some pattern of actual strings which will allow for failure prediction.

Unfortunately, the list of possible strings can be quite large. For example, the `syslog` data set used in this paper contains over 2 billion unique strings, despite the fact that the English language only consists of about 1 million words [14]. Consider a message which posts the temperature of a disk drive. Even if the temperature only fluctuated by ten degrees across all log files, those ten values (assuming the message only posts integer values) are assigned unique identifiers in the alphabet. This alphabet results in a feature space of over 8×10^{27} when $k = 3$.

In an effort to reduce the number of strings, it is possible to isolate only the strings that the SVM finds useful in creating a model. To do this, the SVM is trained on the entire data set, using only the number of times each string appears. Once the SVM builds a model of the data, the feature space is examined to determine the most important strings. Any string which the classifier finds useful will henceforth be referred to as a keystring.

Now that there is a list of the most important strings, these strings are used to create the message list. A count of each string is used as well as a count of each sequence of strings. When building a sequence number, message boundaries are ignored. For example, if keystrings 0 and 29 are in one message, keystring 10 in the next message, and keystring 1 in the third message, the keystring sequence when $k = 4$ is $\{0, 29, 10, 1\}$.

Many keystrings may represent similar items. For example, each computer may have a unique ID number. While each of these keystrings is unique, they all fall under the general label of an ID number. In the interest of reducing the alphabet further, keystrings are grouped into general types, such as computer ID number, and a number is assigned to each type.

Timing information can also be included when using the keystring approach. As with the tag approach, a count is taken of the differences between the time at which the first message in a sequence is posted and that of the final message in a sequence.

4 Support Vector Machines

Each disk can be separated into one of two classes: a disk which failed or a disk which did not fail. The research in this paper uses an SVM to build a model of the two classes based on past `syslog` events.

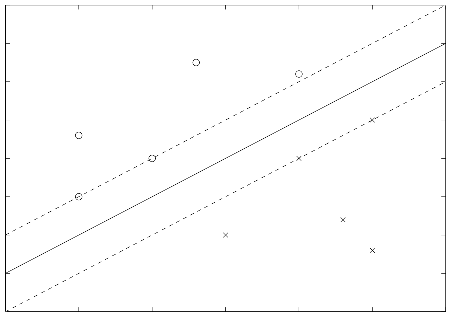


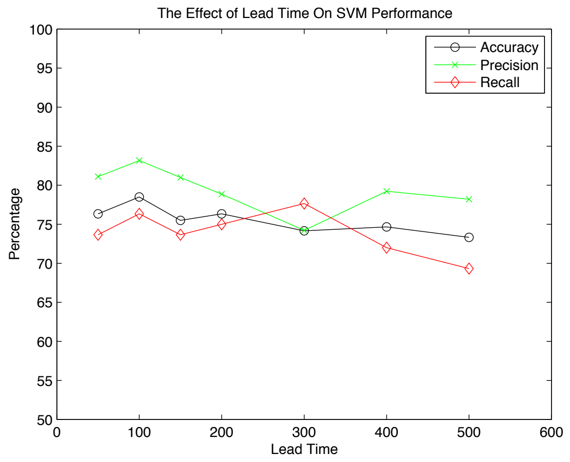
Figure 2: An illustration of the optimal 2-D hyperplane

An SVM is a classification method that takes a set of labeled training examples. Each training example is labeled to indicate which class the example belongs to. Since an SVM is a binary classifier [6], each training example must be in one of two, and only two, classes. The binary nature of the SVM makes it an ideal choice for predicting disk failures, as each example must either fail within the given window or not fail within the given window.

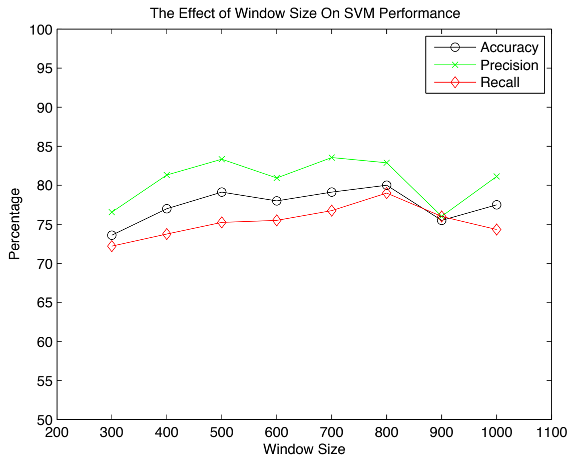
4.1 Optimal Hyperplane

Using an input set of labeled data points, the SVM attempts to find an optimal hyperplane to separate the data. Consider Figure 1, which provides an illustration of the optimal hyperplane between the class of circles and the class of crosses. The optimal hyperplane is the plane which maximizes the distance between the two classes. In the case of the figure, the optimal hyperplane is represented by the solid line.

To calculate the optimal hyperplane, one finds the planes that separate the data which are located closest to each class. In Figure 1, these hyperplanes are represented by dashed lines. Since these hyperplanes are defined by the points closest to the optimal hyperplane, only these few examples are needed to calculate the optimal hyperplane. These data points are called *support vectors*.



(a) The accuracy, precision and recall of tag-based methods as lead time before an event changes while window size remains constant



(b) The accuracy, precision, and recall of tag-based methods as the window size changes while lead time remains constant

Figure 3: The change in performance metrics as window size and lead time vary

5 Experimental Results

The `syslog` data used in these experiments is from a 1024 node Linux-based cluster managed by the Pacific Northwest National Laboratory. Each system contained multiple processors and disks. There were an average of 3.24 messages per machine per hour, which results in about 78 messages per system per day. There were 61 unique tag values, the distribution of which is shown in Figure 2(a). There were over 120 disk failures during the 24 months over which the data was collected.

To train the SVM, blocks of `syslog` messages from each system must first be isolated. The size of the message window specifies the number of messages to isolate prior to a failure. For example, if the window size is 500, then the 500 messages immediately preceding the failure message are isolated. However, if there are not enough messages before the failure, then that window of messages is not used. If a given failure comes within twenty four hours of a previous failure, then the failure is removed from the data set to keep any patterns or events which lead to the first failure from affecting predictions for subsequent failures. The removal of these failures result in 100 useable disk failures. If there are no failures on a given system, then a random window of 500 sequential messages is chosen.

Once all of the message windows are created, they have to be trimmed. To simulate lead time before a failure, a specified number of messages at the end of the window is removed. By deleting messages at the end of the window, there is a gap between the end of the message list and the event to be predicted. As an example, consider a window size of 1,200 messages. If the window is then trimmed by 200 messages, there are 1,000 messages left to classify on. By eliminating the last 200 messages, there is a gap of a little over two days between the final message in the block and the failure or non-failure.

All experiments are performed using hold out and 10-fold cross validation. Hold out means that for both the training and testing stages, an equal number of failure and non-failure examples are in the data set [17]. When using 10-fold cross validation, the data is broken up into ten sets of equal size. The classifier is then trained on nine of these sets and tested on the final set. A different test set is then chosen from the ten groups, while the previous test set is added to the training set, so each of the ten slices eventually is used for testing [17].

The experiments performed in this paper use three metrics to determine the effectiveness of a model: accuracy, precision and recall [10]. Accuracy is the total number of predictions which the model made correctly. Precision is the true positive rate, which indicates the number of disks which were predicted to fail within the given window that actually did fail within that window. Finally, recall is the percentage of actual disk failures that the model successfully predicted.

5.1 Optimal Lead Time and Window Size

The following experiment uses tag sequences of length 5 and a window size of 1,200 messages [4]. The amount of lead time is varied to examine whether or not attempting to predict a failure closer to the failure event improves classification performance.

Figure 3(a) shows the accuracy, precision, and recall of varying the lead time for predictions. The x-axis indicates the lead time in number of messages before a failure event. A failure event occurs where $x = 0$. Each experiment uses a fixed window size of 1,200 messages; therefore, a smaller lead time means that the number of messages used to classify increases, while the time between the final message in the block and the failure event decreases. All three metrics peak with a lead time of 100 messages, which translates to a little over one day.

The recall dips as lead time increases beyond 300 messages due to the widening gap between the end of the window and the failure event. By adding more lead time before a failure, fewer of the messages and patterns which lead up to a disk failure may be present. While some disks might operate in a reduced state for a few days before failure, some start showing signs only a few hours or a day before the failure. By increasing lead time, the model is unable to predict the failure of disks which only provide warning signs closer to the failure event, as those events are no longer in the training or test set.

5.1.1 The Effect of Window Size Using Tag-Based Features

Figure 3(b) illustrates the effect of increasing the window size. Since the results of the previous experiment suggest that a lead time of 100 messages is the most effective, this experiment also uses a lead time of 100 messages. All three metrics increase until the window size hits 800 messages. After a window size of 800 messages, just like the previous experiment, recall begins declining. Recall declines because, as the window size increases, the SVM must classify using more and more information. The increased information can make the two classes begin to look similar. In the case of disk failures, the disks only produce warning signs for a certain period of time. Before these warning signs appear, they operate as normal disks. By adding information from before the disks start to fail, that disk acts more like a working disk than a failing disk.

5.2 Tag-Based Features Without Timing Information

The previous experiments all use sequences of length 5. This experiment varies the sequence length between sequences of length 3 and sequences of length 8. While increasing the sequence length may increase the effectiveness of the model, the increase will also exponentially increase the feature space. As such, the time required to train and classify the data will increase. Therefore, a balance must be struck between the effectiveness of the model and the time required to train. Table 3 compares the accuracy, precision, and recall of training using each sequence length. All experiments used a window size of 800 messages and a lead time of 100 messages. The recall is maximized using sequences of length 6. On the other hand, the precision jumps to 85% at a sequence length of 7. The recall plummets using sequences of length 8. Henceforth, sequences of length 5 are used because they provide fewer false positives compared to a sequence of length 6 while achieving similar recall while using a smaller feature space than either sequences of length 6 or of length 7.

Sequence Length	Accuracy	Precision	Recall
3	73.166	74.9003	75.0011
4	75.6666	80.8341	72.6681
5	79.9993	82.8838	79.0012
6	79.4994	80.5503	80.6674
7	80.999	85.4837	78.668
8	78.4992	85.7335	73.3339

Table 3: A comparison of sequence lengths when using tag-based features

5.3 Tag-Based Features With Timing Information

The performance of classification using timing information is compared to the performance without timing information in Tables 4 and 5. In neither case did the addition of time differentials significantly increase any of the three metrics. In the case of length 5 sequences, the recall actually gets substantially worse. When using sequences of length 7, the results with and without time are almost identical, as seen in Table 5. In both cases, the recall may dip because the message logging rate of nodes on a which a failure is going to occur within the next 100 messages is similar to the message logging rate of nodes which are not predicted to fail. Since the two rates are similar, the inclusion of timing information makes the two classes look more similar than when no timing information is included. Therefore, the addition of timing information not only does not provide improvement over tag sequences without timing information, but it also increases the feature space. As a result, tag based features are best when used without timing information.

Feature Space	Accuracy	Precision	Recall
Sequences Using Tags	79.9993	82.8838	79.0012
Sequences Using Tags and Time	77.8329	82.2338	71.667

Table 4: Comparing performance between features using only tags and features including time information using sequences of length 5

Feature Space	Accuracy	Precision	Recall
Sequences Using Tags	80.999	85.4837	78.668
Sequences Using Tags and Time	81.1661	86.9337	76.005

Table 5: Comparing performance between features using only tags and features including time information using sequences of length 7

5.4 Keystring Based Features Without Timing Information

The initial dictionary contains 54-keystrings. However, 25 of the strings in this dictionary are the names of nodes on the cluster. To see whether or not the SVM is learning what nodes tend to fail instead of actual patterns which lead to failures, another dictionary is tested. The second dictionary, made up of 24-keystrings, assigns all keystrings of a given type to a single number. For example, all node names are assigned a 0 and all number strings are assigned a 23. In the 24-keystring dictionary, all node names, even those not in the original 54-keystring dictionary, are included. The results of these experiments using a window size of 800 messages, lead time of 100 messages and sequences of length 3 are recorded in Table 6. The fact that the 54-keystring and 24-keystring dictionaries perform similarly well suggests that the SVM is not training on specific node names. Instead, the SVM is learning that the appearance of any node name is useful for predicting failures. The 24-keystring dictionary has the benefit of reducing the alphabet size dramatically when compared to the 54-keystring dictionary. As a result, all keystring experiments henceforth use the 24-keystring dictionary.

Dictionary	Accuracy	Precision	Recall
54	77.6661	81.8171	76.0008
24	77.6659	79.1004	78.6676

Table 6: A comparison of keystring dictionaries

Table 7 shows the change in performance as the sequence length increases when using the 24-keystring dictionary. Sequence of length 4 perform significantly better across the board than those of length 3. While length 5 sequences perform slightly better than those of length 4, the improvement is not significant with respect to recall, although the false positive rate dips slightly. Since length 5 sequences significantly increase the feature space with marginal benefit, the 24-keystring dictionary performs best when using sequences of length 4.

Sequence Length	Accuracy	Precision	Recall
3	77.6659	79.1004	78.6676
4	79.4996	82.9838	80.6676
5	82.1428	85.0008	80.9543

Table 7: Performance of the 24-keystring dictionary as sequence length increases

5.5 Keystring Based Features With Timing Information

Despite the ineffectiveness of combining timing information with tag sequences, the usefulness of timing with regards to the keystring based approach is tested. Table 8 compares the performance of the 24-keystring approach both with and without timing information. The sequence length used for both experiments is 4. The accuracy and recall values of both approaches are essentially the same. However, when using time information, there is a slightly lower false positive rate. Since a lower false positive rate means fewer instances when a node goes into a preemptive maintenance stage, minimizing the false positives is a worthy goal. While the feature space increases, a system administrator may be willing to endure the longer training and classification time if it results in fewer false positives. Thus, time information keystring sequences are added to the final experiment.

Experiment	Accuracy	Precision	Recall
Without Time Info	79.4996	82.9838	80.6676
With Time Info	80.1657	85.567	78.6679

Table 8: A comparison of the 24-keystring dictionary with and without the addition of time information

5.6 Combination Results

Classifying works almost identically well when using tag number sequences of length 5 as when using keystring sequences of length 4. The use of tag number sequences achieves a slightly higher true positive rate while keeping a similar accuracy and recall. If the tag-based approach and the keystring-based approach are learning on different patterns, then perhaps combining the two approaches will result in better classifications.

The window size for this experiment is 800 messages and the lead time is 100 messages. Tag sequences of length 5 are used, while keystring sequences are 4 keystrings long. Since the addition of temporal features is useful with keystring based features, time differences are calculated for sequences of length 4.

Approach	Accuracy	Precision	Recall
Tags Without Time	80.999	85.4837	78.668
Keystrings With Time	80.1657	85.567	78.6679
Combination Without Time	77.9995	82.317	74.334
Combination With Time	80.6664	88.567	74.6673

Table 9: A comparison of tag based, keystring based, and combination methods

Table 9 provides a comparison among the best performing tag based approach, the best keystring approach, and a combination approach both with and without time information. Neither a combination of tag and keystring features with or without additional timing information offers any substantial increase in accuracy and both see a dip in recall, which means the combination model predicts fewer of the failures that occur. The recall dips because the message rate does not provide a good indicator of a failure. As a result, the inclusion of time information makes the two classes look more similar. However, this increased similarity results in higher precision. The precision increases because disks which were predicted to fail with low confidence when omitting timing information are now predicted to continue working. Therefore, only disks that have a high confidence score for failure are still predicted to fail. With the combination of approaches and time information, the decrease in the number of failures predicted is balanced by an increased true positive rate, meaning that almost 89% of the disk failures predicted by this approach do fail within the next 30 hours. In fact, this model has the highest true positive rate of any of the experiments, which means that the combination approach in conjunction with timing information provides a useful improvement over other models. Whether or not it is the best model depends on whether a higher recall rate or fewer false positives is the most desired trait in a given situation.

6 Conclusions and Future Work

To determine the overall best method, this section considers the true positive rate as well as the recall. In addition, this section proposes an event logging system which requires less storage space than the current `syslog` utility.

If a high recall is the more important goal, the best approach is to use either tag sequences without timing information or keystring sequences using the 24-keystring dictionary with timing information. Both approaches hit almost 80% recall. In addition, both had very few misclassified failures. If a high true positive rate is the most desired classification trait for a given situation, then there is only one choice: combining tag sequences with keystring sequences and timing information, as this approach has a true positive rate of 89% while still predicting 75% of disk failures.

The PNNL data set used for this experiment contained, on average, 78 `syslog` messages an hour for each node. As a result, there are approximately 699,678,720 messages on the cluster every year, which requires 41.7 GB to store.

Now consider that using only keywords or tag numbers to predict failures is rather effective. If one can predict events using only tag numbers or keywords, then perhaps one could keep only the fields required for these predictions.

While the approach which marries tag numbers and timing information is the best combination of speed and accuracy, combining the keywords with timing information performs the best overall. Keeping keywords provides another benefit over just keeping tags: some amount of semantic data is retained. Assume all words are kept. Keeping all of the words allows the same data to be broken up using a different set of keywords if a user is trying to predict another type of event or if a more effective keyword list for the current problem is found. In this case, the only fields that are necessary are the timing information and the message itself, as the level, facility, and tag numbers add nothing to this prediction approach. As a result, each message will be 31 bytes on average, which will take up 20.2 GB per year for a 51.563% reduction on the overall storage space needed.

Maximizing precision requires that one keep the tag numbers as well. Keeping tag numbers as well as timestamps and the message field uses 22.8 GB per year. Therefore, one would need to keep about 2.6 more GB per year than when using only keystrings to maximize recall. However, this still represents a marked improvement over the space required by standard `syslog` and is the best choice if one wishes to minimize the false positive rate.

There are two branches this research can take immediately. The first direction is to try different classification methods. This research only examines the effectiveness of the SVM approach to classifying nodes as likely to fail. Future work can explore the effectiveness of both unsupervised learning methods and other supervised learning methods.

Another direction this research could move in is to try to predict other events. Perhaps this same approach could be used to predict whether or not an entire node is going to go offline or if a RAID controller is going to fail. One would simply need to find these events in the logs and label each feature vector appropriately before training. Otherwise, the approach, as far as finding sequence numbers or keywords, is exactly the same.

The generalizability of this approach should also be examined by applying the approach to different data sets. For example, a cluster may have a different `syslog` configuration, average message rate, or applications which are installed than those seen in the data set used for this thesis. Perhaps these fluctuations in configuration also affect the usefulness of the proposed approach. As another example, perhaps the tag number distribution in a given set up is different than that of the PNNL data set. In this case, it may be necessary to alter either the alphabet size or the cutoff values for each criticality score.

References

[1] Bruce Allen. Monitoring hard disks with smart. *Linux Journal*, 1(117), January 2004. Available at: http://www.linuxjournal.com/magazine/monitoring-hard-disks-smart. Accessed on April 19, 2010.

[2] Peter Broadwell. Component failure prediction using supervised naive bayesian classification, December 2002. Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.4641. Accessed on April 19, 2010.

[3] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20:273–297, 1995.

[4] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. Predicting computer system failures using support vector machines. In *First USENIX Workshop on the Analysis of Logs (WASL)*, 2008.

[5] Greg Hamerly and Charles Elkan. Bayesian approaches to failure prediction for disk drives. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICLM)*, June 2001.

[6] Andrew Karode. Support vector machine classification of network streams using a spectrum kernel encoding. Master's thesis, Wake Forest University, December 2008.

[7] Christina Leslie, Eleazar Eskin, and William Stafford Noble. The spectrum kernel: A string kernel for svm protein classification. In *Proceedings of the Pacific Symposium on Biocomputing 7*, January 2002.

[8] Yinglung Liang, Yanyong Zhang, Hui Xiong, and Ramendra Sahoo. Failure prediction in ibm bluegene/l event logs. In *Proceedings of the Sevent IEEE International Conference on Data Mining*, 2007.

[9] C. Lonvick. The bsd syslog protocol, 2001. Available at: http://www.faqs.org/rfcs/rfc3164.html. Accessed on: April 19, 2010.

[10] John Makhoul, Francis Kubala, Richard Schwartz, and Ralph Weischedel. Performance measures for information extraction. In *Proceedings of DARPA Broadcast News Workshop*, pages 249–252, 1999.

[11] Joseph F. Murray, Gordon F. Hughes, and Kenneth Kreutz-Delgado. Hard drive failure prediction using non-parametric statistical methods. In *Proceedings of ICANN/ICONIP*, June 2003.

[12] E. Pinheiro, W.D. Webe, and L.A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST '07)*, February 2007.

[13] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 28, 2007.

[14] John Simpson and Edmund Weiner, editors. *Oxford English Dictionary*, volume 1. Oxford University Press, second edition, 1989.

[15] William H. Turkett, Andrew V. Karode, and Errin W. Fulp. In-the-dark network traffic classification using support vector machines. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2008.

[16] Doug Turnbull and Neil Alldrin. Failure prediction in hardware systems, 2003. Available at: http://www.cs.ucsd.edu/ dturnbul/Papers/ServerPrediction.pdf. Accessed on: April 19, 2010.

[17] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kauffman, second edition, 2005.

Log Analysis and Event Correlation Using Variable Temporal Event Correlator (VTEC)

Paul Krizak - Advanced Micro Devices, Inc.
paul.krizak@amd.com

ABSTRACT

System administrators have utilized log analysis for decades to monitor and automate their environments. As compute environments grow, and the scope and volume of the logs increase, it becomes more difficult to get timely, useful data and appropriate triggers for enabling automation using traditional tools like Swatch. Cloud computing is intensifying this problem as the number of systems in datacenters increases dramatically. To address these problems at AMD, we developed a tool we call the Variable Temporal Event Correlator, or VTEC.

VTEC has unique design features, such as inherent multi-threaded/multi-process design, a flexible and extensible programming interface, built-in job queuing, and a novel method for storing and describing temporal information about events, that well suit it for quickly and efficiently handling a broad range of event correlation tasks in real-time. These features also enable VTEC to scale to tens of gigabytes of log data processed per day. This paper describes the architecture, use, and efficacy of this tool, which has been in production at AMD for more than four years.

Tags: security, case study, syslog, log analysis, event correlation, temporal variables

1 Introduction

Log analysis is a critical component for effective automation of large cloud computing environments. As clouds grow, day-to-day operational tasks such as failing hardware become an increasing burden for datacenter operational staff. In addition, emergent behavior in large clouds causes unusual problems that often are difficult to diagnose. These issues require more complex automation techniques in system maintenance and service operation. Modern solutions such as SEC [1,2] and Splunk [3] have done a great job at scaling to large log volumes and making complex correlations feasible, but they have drawbacks. This paper presents an alternative solution we developed at AMD called the Variable Temporal Event Correlator, or VTEC.

AMD designed VTEC with multi-core and multi-system scalability in mind. Virtually every component is multi-process and/or multi-threaded to take advantage of every available CPU cycle on the system. If needed, each component can be isolated on its own machine to distribute load.

VTEC also introduces a novel method for representing temporal event data; these constructs are called *temporal variables*. Temporal variables are constructed to represent temporal data about events, such as frequency and rate of change, in a way that is immediately useful when building event-correlation rules. These rules can make use of the temporal data without the need for extra processing in the rule itself.

Finally, VTEC includes a built-in job scheduler that allows for categorization, scheduling, and prioritization of actions generated in response to events. This gives the user finer control over the sequencing and priorities of actions generated by log analysis than available before.

This paper is organized as follows: Section 2 describes the computing environment and log analysis needs that drove the creation of VTEC. Section 3 describes the internal architecture of VTEC. Section 4 details several example "rule engines" that demonstrate how VTEC can be used to correlate various kinds of events in a computing environment. Section 5 briefly describes some of the useful rule engines implemented at AMD. Section 6 discusses designing for performance and scaling as log traffic increases. Section 7 describes the challenges currently faced using VTEC, and areas for improvement.

2 Background

As compute environments continue to grow in size, it becomes increasingly challenging to keep track of the various events happening in the grid. How does a systems team note and track the failure of a hard disk or memory? Or when one of thousands of compute nodes suffers a kernel panic? An even more complex task is to deal with the inevitable emergent behavior of a massive network of computers. Seemingly innocuous changes to systems (e.g., adding a simple cronjob to grid nodes), can have unexpected consequences (e.g., overloading NIS/LDAP servers).

In this large, ever-changing, and complex computing environment, many organizations (including AMD) have turned to the practice of autonomic computing [4] to reduce the effort that sysadmins must exert to keep the environment stable. There is a system and OS configuration aspect to this, in which tools such as Cfengine [5] can enable autonomic behavior. There is still a gap, however, when it comes to detecting anomalous or interesting events, correlating them, and taking automatic action to alert people or correct the problem.

In the context of large Linux/UNIX compute grids, raw event data is generally available via syslog. Standard syslog daemons, as well as more advanced ones such as syslog-ng [6], are able to send log data to a central server. Thus, collecting enough raw data to analyze is rarely difficult. However, the volume of this data is often a problem: so much data is collected that it becomes difficult to parse and extract useful information from it.

During the past several years, a number of log parsing and event correlation tools have been developed. Some Linux distributions, such as Fedora, ship with the Logwatch [8] utility installed. Logwatch parses system log files regularly and provides useful, human-readable reports via e-mail. When using Logwatch, however, sysadmins are required to wade through e-mails and make event correlations manually, so it does not scale beyond a handful of servers.

One of the first automated log analysis tools used at AMD was Swatch [7]. Swatch is very much a reactionary log analysis system. Events matching a particular regular expression can trigger events, such as an e-mail to an administrator. At AMD, rudimentary flood prevention routines often caused important events to be missed while a rule was stalled waiting for a flood timer to expire. In addition, Swatch is single-threaded, and so was unable to scale to AMD's ever-increasing log volume (about 10GB/day at a typical site).

Around 2006, as AMD's compute grids were growing at a rapid rate, the company determined the aging Swatch installation was no longer effective and chartered a project to implement a replacement log monitoring and analysis system. The parameters for the project were:

- Scale to tens of gigabytes of log data per day
- Take advantage of multiple processors (AMD's strategy is to embrace multi-core computing)
- Be able to correlate events across thousands of systems in real-time (no batch processing)
- Be able to prioritize and queue system repair jobs and e-mail alerts
- Prevent floods of alerts without missing important events
- Correlate events on arbitrary log streams (e.g. FlexLM license daemon logs)
- Ensure correlation rules are easy to read, modify, and create

At the time, SEC [1] and Splunk [3] were popular choices as core components to achieve the goals. AMD tested both thoroughly, and ultimately decided a homegrown system would be best. SEC was (and still is) a very powerful and flexible tool, but the learning curve for writing rules its rules is quite steep. This was demonstrated by [1], which was actually supposed to "demystify" SEC. However, even moderately complex SEC rules were deemed unintelligible by sysadmins unfamiliar with its use.

Splunk did not have an indexing system robust enough to handle the volume of log traffic we expected to run through it. After routing just a few hundred megabytes of syslog data through Splunk, the indexer would stop working properly despite several weeks of tuning with the help of Splunk's developers. Finally, the event correlation features were limited to Swatch-like functionality (this was in 2006: Splunk v1.0).

With its seven design goals in mind, AMD created VTEC from scratch. In the process, we evolved a novel set of data types and data handling methods that have since greatly simplified event correlation and log analysis at AMD.

3 Architecture

VTEC Components

The VTEC system consists of five modules (Figure 1). Each module has a specific, well-defined task to perform, and communicates with the other modules in a well-defined language over TCP sockets. Splitting the system into multiple components generates a number of benefits:

- Bugs/errors in one rule engine generally do not affect the others
- Multiple processes can leverage CPU capacity in multi-core systems
- Standard interfaces between modules simplify the task of tuning modules to optimize performance or add features

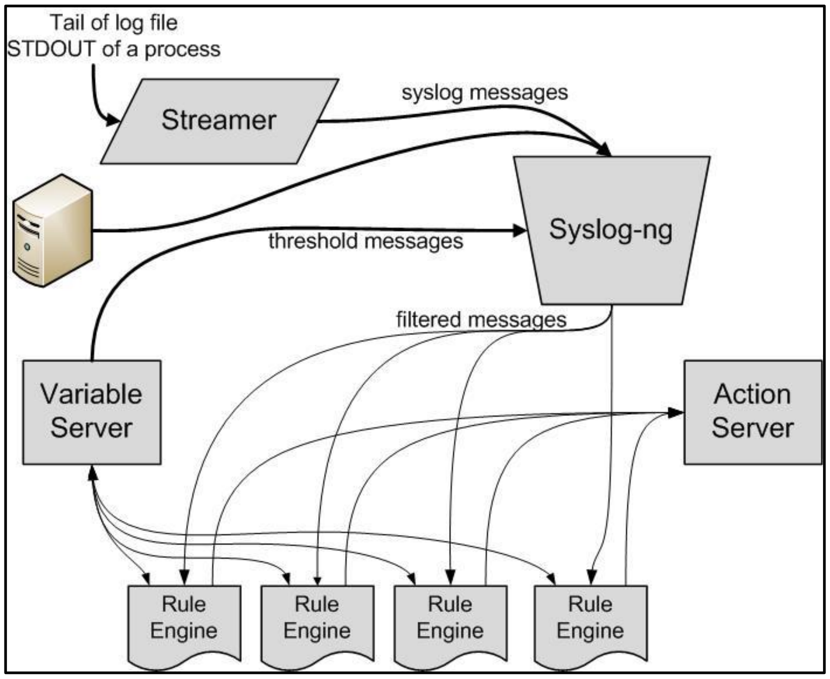


Figure 1: VTEC System Architecture

Streamer - Log data enters VTEC via syslog. The streamer component can tail log files or send arbitrary data from STDOUT of a process to VTEC. Its most useful feature is the ability to insert "heartbeat" messages into the log stream so the absence of event data can trigger actions [10]. Use of the streamer component is optional; most systems at AMD simply route their log data directly to the syslog-ng component without going through a streamer.

Syslog-ng - VTEC uses the powerful syslog-ng system logger [6] as the log router; it is the only non-Perl component. Its purpose is to accept log streams from syslog, TCP/UDP sockets, and streamers. It reformats messages, filters data, and routes messages to the appropriate rule engines and/or archival log files. Control of the filtering is accomplished by including filtering metadata within each rule engine, dictating what log data that rule engine wishes to receive. The VTEC installer then injects this metadata into the `syslog-ng.conf` file, ensuring that each rule engine is tasked with parsing only the log data that is relevant to it. The static parts of the `syslog-ng.conf` file allow for searchable log archives to be created. For example, all log messages could go to
`/var/log/YYYY/MM/DD/hostname-msgs.log.`

Rule engines - These are composed of any executable code that accepts filtered and reformatted log data on STDIN. In practice, these are Perl scripts, created from a template, that include reusable interface modules to both the action server and the temporal variable server (which communicate over a standard TCP protocol). The rule engines are where the event correlation tasks occur. Since the rule engines are open-ended editable custom scripts, a rule engine can do anything your language of choice can do.

Temporal variable server - VTEC hosts all the temporal variables in a separate server daemon. This frees the rule engines from the drudgery of maintaining state across reboots/restarts. It also allows rule engines to share data easily, since the variable server presents a shared namespace to the rule engines; one rule engine can set a temporal variable, and another rule engine can query for it by using the same variable name.

The temporal variable server can also inject special messages into the log stream when certain threshold conditions are met. Rule engines can watch for these threshold messages and take appropriate action without being burdened with having to constantly query the state of variables.

A Perl object interface to the temporal variable server is provided for use in rule engines. Additionally, in cases in which the temporal variable server's features are more than are required, the temporal variable data types are available as Perl objects, meaning that rule engines can instantiate them locally without having to contact the temporal variable server at all.

Action server - When rule engines need to take some sort of action, they have the option of running that task locally (which is not advisable, since this can block the rule engine from processing incoming data) or queuing a job in the action server. Jobs are implemented in a Perl module as subroutines; queuing a job really means sending the name of a subroutine, its parameters, when to run it, and a queue name to the action server over a standard TCP socket interface. A Perl object interface to the action server is provided for use in rule engines.

The action server has a number of job queues with varying priorities. Users can schedule jobs to run immediately or at a specific time (e.g., alert a sysadmin about this event, but not until 10 a.m., when they are awake). The action server processes queues with higher priority first, allowing an emergency page to go out immediately despite a backlog of less urgent repair jobs in lower-priority queues.

The actions that can be queued on the action server are defined in a Perl module as functions. This allows actions to be developed that can be shared by rule engines. Since the actions are implemented in Perl, they can do virtually anything to the environment. They can also communicate with the variable server if needed to get information about how to execute. Some basic actions implemented on AMD's action server are:

- **run_cmd** - executes a command on a remote machine
- **send_mail** - sends an e-mail
- **hopenclose** - instructs AMD's batch scheduling system to open or close a host to jobs
- **inject_log** - injects a syslog message into the log stream (for signaling rule engines about job status)

In addition to the actions themselves, an arbitrary number of queues that can be defined in the action server, with varying levels of priority. The queues are defined in a configuration file. In practice, only three queues are needed:

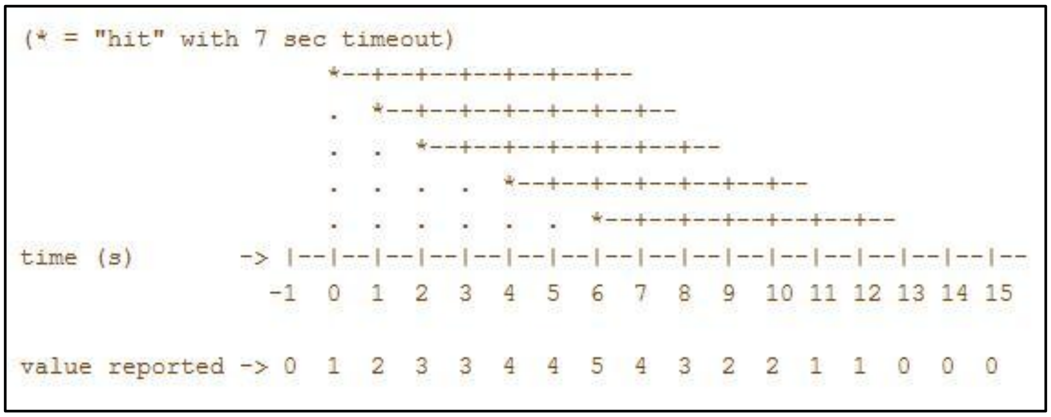
- **crisis** - when any jobs are placed in this queue, all other queues are halted until this queue is empty (good for hotpage events or status checks that are time-sensitive)
- **normal** - normal jobs, such as rebooting or reinstalling a server, or running a script
- **email** - most e-mail jobs get queued here, so that they get processed in parallel with other jobs (e.g. a `run_cmd` job in the `normal` queue won't have to wait for a flood of 500 e-mails to get sent before executing)

Temporal Variable Data Types

The most interesting and novel aspect of VTEC is the temporal variable server and the temporal variables it hosts. There are three data types in the temporal variable server:

Scalar - A scalar consists of two pieces of information: a piece of scalar *data* (such as a string or a number) and a *timeout*. The timeout is set when the scalar is created, and defines the length of time the data is valid. When the scalar is queried, the timeout is checked. If the current time has not exceeded the timeout, the data value is returned. Otherwise, a zero is returned. Scalars are useful for setting alarms, preventing e-mail floods, and storing temporary data.

Incrementer - The incrementer data type builds on the scalar. It is an organized collection of scalars, each with a data value of 1. When an incrementer is instantiated, it is given a timeout value. Every time the incrementer is set (called a *hit* in VTEC), a new scalar is added to the collection with the predetermined timeout and a data value of 1. When queried, the incrementer returns the sum of the values of its constituent scalars. Incrementers are useful for calculating and observing the rate of events over time. Figure 2 demonstrates how the value of an incrementer changes over time as it is repeatedly hit by a rule engine. When read at *time=4 sec*, the reported value of "4" indicates that the current rate of hits is 4 per 7 seconds, or 34.29 per minute.



```
(* = "hit" with 7 sec timeout)
                     *--+--+--+--+--+--+--
                     .  *--+--+--+--+--+--+--
                     .  .  *--+--+--+--+--+--+--
                     .  .  .  .  *--+--+--+--+--+--+--
                     .  .  .  .  .  .  *--+--+--+--+--+--+--
time (s)       -> |--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--
                 -1  0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15

value reported -> 0  1  2  3  3  4  4  5  4  3  2  2  1  1  0  0  0
```

Figure 2: Value of an Incrementer over Time

List - A list is a collection of incrementers that are each referenced by a *key* - in short, a Perl hash of incrementer objects. Lists have the unique property that they can be queried in three different ways:

1. The value of a particular key (the current value of that incrementer);
2. The sum of the current values of all keys; or,
3. The number of non-zero keys.

Lists are useful because they can aggregate event rate data and organize it (e.g., by hostname), then present immediately useful data about that collection of rate data (e.g., the current number of hosts reporting an event, or the total number of those events across the entire environment).

4 Examples

In all of the following code examples, a substantial part of the actual Perl script has been removed for clarity. All Perl-based VTEC rule engines have three major sections:

1. syslog-ng metadata. This information is stored in comments at the top of the rule engine. The VTEC installer parses this metadata and builds syslog-ng filter rules to ensure the desired log messages are passed into the rule engine.
2. Initialization code. This is boilerplate Perl code that includes the appropriate VTEC support modules, instantiates the `$variable_server` and `$action_server` objects, and sets up any thresholds needed by the rule engine in the variable server. This code is not shown in the examples in this paper, for brevity.
3. Log processing. This takes place in the `while(<STDIN>) {}` loop, and is generally the only part of the code that the sysadmin has to actually modify from the template.

While most of a rule engine's code is boilerplate, it was decided to leave it available for modification to ensure that the rule engines would never be limited by any sort of hard-coded framework. Plenty of comments are provided in the template to guide the rule engine author. While most event correlation cases can be covered with a few slight modifications of the template, more complex cases might occasionally require inclusion of extra Perl modules or other initialization code that might be complicated by unnecessary abstraction of the code.

In **Code Example 1,** the most basic type of rule engine is demonstrated; compare it to "Repeat Elimination and Compression" in [1]. The syslog-ng metadata ensures the only log data being passed into the rule engine on

STDIN are messages from host amdftp, where the message portion matches the regular expression disk full error. By performing initial filtering in syslog-ng before the messages even reach the rule engine, the filtering process is only done once, rather than all rule engines having to parse through all the messages looking for matches.

When these messages arrive, the variable server is queried to see if a scalar by the name of ftp_disk_full is set. If not, an outgoing e-mail is queued on the action server, and the ftp_disk_full scalar is set, with a one-hour timeout, to prevent a flood of e-mail from being sent.

Notice that the job submission hash includes a start parameter. This parameter can be set to a delay (e.g. +600 means "10 minutes from now") or to an absolute time by using localtime() to generate an epoch time. Thus with very little added effort, a non-critical alert could be queued and delivered during business hours.

Code Example 1: Simple Event Alert with Repeat Elimination/Compression

```
# syslog-ng metadata
# filter: host("amdftp") and match("disk full error");

while(<STDIN>) {
    if($variable_server->get_scalar("ftp_disk_full") == 0) {
        my %job = (
            start      => time(),
            queue      => "email",
            action     => send_mail,
            parameters => [ $address, $subject, $msg, $name ]
        );
        if($action_server->add(%job)) {
            # scalars can be set to arbitrary values (second parameter), but
            # always return 0 when they have timed out.
            $variable_server->set_scalar("ftp_disk_full", 1, "+3600");
        }
    }
}
```

In **Code Example 2,** a more complex event correlation is demonstrated. Following [1], this code example demonstrates the functionality in both "Report on Analysis of Event Contents" and "Detect Identical Events Occurring Across Multiple Hosts". In this example, VTEC is configured to watch for "NFS server not responding" messages from all hosts, and will alert if the number of unique systems reporting this message for a particular file server exceeds 50 in a given five-minute period.

First, notice the syslog-ng metadata is a bit more involved this time. We are filtering three types of messages into the rule engine: server XXX not responding, server XXX OK, and threshold messages. The threshold messages are important: when the rule engine runs the set_list_threshold method when it starts up, the variable server sets up a watch for that particular list. In this case, we're telling it to watch two lists, filer_down and filer_up, and to send an alert if the value of a single key in either of them exceeds 50. The rule engine can then capture that alert and take action.

Following the code, as log messages flow in from STDIN, they are checked to see if they are one of the three types we are filtering for. In the basic filer down/up case, the variable server is told to update the appropriate list variable, using the affected file server's name as the key. The timeout for each hit is set to five minutes. Conceptually, this means we'll have two groups of incrementers, indexed by the name of the file server. The value of each incrementer tells us roughly how many hosts are simultaneously reporting the given state for that server (we assume each host only reports the "not responding" or "OK" message once every five or more minutes). If the message is a threshold message, all we need to do is take action - in this case, send an e-mail about the affected filer.

The important thing to grasp with this example is the relative simplicity with which this correlation was achieved, especially compared to the same example in Figure 9 of [1]. While the overall functionality is effectively the same, any sysadmin able to read basic Perl should be able to interpret, modify, and use the VTEC rule engine; the same functionality in SEC is much more difficult to grasp, even for a seasoned programmer.

Code Example 2: Correlating Events Across Hosts

```
# syslog-ng metadata
# filter: match(".*server.*not responding.*") or \
#         match(".*server.*OK.*") or \
#         filter(f_thresholds);

# Set up a threshold when we start up.  Thresholds are purged after 48 hours of inactivity.
# VTEC restarts all rule engines nightly (during log rotation) to ensure needed thresholds
# are kept fresh, and retired ones are purged from the variable server.
$variable_server->set_list_threshold("ONE", "filer_down", ">", "50");
$variable_server->set_list_threshold("ONE", "filer_up", ">", "50");

my $window_secs = 300;  # sliding window of 5 minutes
my $email_flood = 1800; # seconds between e-mails about a filer up/down

while(<STDIN>) {
    my %message = parse($_);

    # Filer down messages
    if($message{message} =~ /server (\w+)( is)? not responding/) {
        $variable_server->set_list("filer_down", $1, "+$window_secs");
        next;
    }

    # Filer up messages
    if($message{message} =~ /server (\w+) OK/) {
        $variable_server->set_list("filer_up", $1, "+$window_secs");
        next;
    }

    # Filer up/down threshold exceeded messages
    if($message{message} =~ /THRESHOLD EXCEEDED.*filer_(up|down).*\[(\w+)\].*\((\d+)\)/) {
        my ($type, $filer, $num_messages) = ($1, $2, $3);
        # Create a scalar that we'll use to prevent e-mail flooding for this filer
        my $scalar_name = "email_filer_$type" . "_$filer";
        unless($variable_server->get_scalar($scalar_name)) {
            my %job = (
                start      => time(),
                queue      => "email",
                action     => "send_mail",
                parameters => [ $to_address, "Filer $filer being reported $type!",
    "Filer $filer has been reported $type $num_messages times in the last $window minutes." ]
            );
            Queue(\%job);
            $variable_server->set_scalar($scalar_name, 1, "+$email_flood")
        }
    }
}
```

With a few minor modifications, the code in Example 2 could be updated to assist in security monitoring. If you wanted to check for usernames that are making repeated unsuccessful attempts to login to a machine (brute-force attack), a number of list variables could be created and monitored with thresholds:

- A list using usernames as the key, hit each time a message about an unsuccessful login attempt is seen for that username. Using a LIST ONE threshold would alert when any given username has repeated failed logins (whether on a single host or distributed among many hosts).
- A list using source IPs as the key, hit each time a message about an unsuccessful login attempt is seen for that IP. Using a LIST KEYS threshold would alert when the number of source IPs attempting to connect in a short period of time increases beyond a threshold (e.g. a denial of service attack, or possibly a distributed brute force attack). Using a LIST ONE threshold would alert when a given source IP is making repeated login attempts (e.g. brute force attack).

In **Code Example 3,** the ability to check for the absence of a log message is demonstrated. This functionality is achieved by using an alarm, which is a function provided by the temporal variable server and is implemented using what amounts to an anonymous scalar variable with a threshold attached.

Much like the previous example, the code flows in a way that an average sysadmin with some Perl experience would be able to comprehend. We filter for three types of messages: the job start message, the job finished message, and messages generated by the variable server when thresholds/alarms are generated.

If the message is a job start, the job number is fetched from the message, an alarm is set, and a pair of Perl hashes is used to create a mapping between the job number and the alarm name. If the job finished message arrives before the alarm goes off, the alarm is cleared and no further action is taken. If the alarm message arrives, we use the mapping to see which job is late, and send an appropriate alert. Further, if the late job actually ends up finishing, we can detect that too by noticing there is no alarm set for that particular job.

In Code Example 3, the alert portion has been shortened into a pseudo-function called `queue_alert()`. In a real VTEC rule engine, the rule engine author would write a function that makes a call to the action server and have it queue an e-mail (or some sort of corrective action).

With a few modifications, this code could have the rule engine comprehend multi-line log messages. For example, ECC errors on Linux systems often appear on multiple lines:

```
kernel: CPU 3: Silent Northbridge MCE
kernel: Northbridge status 940c4002:85080813
kernel:     Error chipkill ecc error
kernel:     ECC error syndrome 8518
kernel:       bus  error  local  node  origin,  \
       request didn't time out
kernel:     generic read
kernel:     memory access, level generic
kernel:     link number 0
kernel:     err cpu0
kernel:     corrected ecc error
kernel:     previous error lost
kernel:                NB    error    address    \
       0000001a230571d0
```

By utilizing short alarms and a rudimentary state machine in the rule engine, a robust method for capturing multi-line messages like this can be built, which will function even if the log message is incomplete or missing lines. In fact, at AMD we have implemented just such a rule engine that aggregates machine check errors like these and injects new single-line messages into the log stream so yet another rule engine can look for systems with unusually high rates of ECC errors, and close them for repair. In some cases it can even detect which DIMM has failed (by analyzing the syndrome codes) and add that information to the system log in the asset database.

Code Example 3: Checking for Missing Events

```
# syslog-ng metadata
# filter: match("Job.*started") or match("Job.*complete") or \
#         filter(f_thresholds);

my %alarms_by_job;
my %jobs_by_alarm;
while(<STDIN>) {
    my %message = parse($_);
    # If job started, set an alarm so we can alert if it does not finish in 10 minutes
    if($message{message} =~ /Job ([0-9]+) started/) {
        # set_alarm returns the name of the scalar it created
        $alarms_by_job{$1} = $variable_server->set_alarm("+600");
        $jobs_by_alarm{$alarms_by_job{$1}} = $1;
    }
    if($message{message} =~ /Job ([0-9]+) completed/) {
        my $job = $1;
        if($variable_server->get_scalar($alarms_by_job{$job}) {
            # if the alarm is still active, clear it; we're OK
            $variable_server->clear_alarm($alarms_by_job{$job});
            delete $jobs_by_alarm{$alarms_by_job{$job}};
            delete $alarms_by_job{$job};
        }
        else {
            # the alarm isn't active: the job has finished, but finished late.
            delete $jobs_by_alarm{$alarms_by_job{$job}};
            delete $alarms_by_job{$job};
            queue_alert("Job $job finished late!");
        }
    }
    if($message{message} =~ /TIMEOUT: (\S+) (\S+)/) {
        my ($type, $name) = ($1, $2);
        # One of the jobs didn't finish within 10 minutes, so see which
        # job it was and send an alert.
        queue_alert("Job $jobs_by_alarm{$name} has been running for > 10 minutes!");
    }
}
```

Code Example 4 demonstrates the ability to "chain" rule engines. In this example, we have two low-level rule engines, each checking for a different kind of hardware problem on systems. These rule engines use list variables and thresholds to take action when the rate of each type of hardware error exceeds an acceptable threshold. The action taken in these cases is to inject a log message into the log stream that a third, "master" rule engine intercepts. This third rule engine utilizes the key-counting functionality of lists to determine quickly how many kinds of hardware problems a given system has. In the example, a pseudo-action "A" is taken if a system has only one type of hardware problem, while pseudo-action "B" is taken if a system has both.

The net result is that a fairly daunting correlation task is reduced into its three core correlations. These correlations are easily made individually by utilizing temporal variable constructs in the temporal variable server, making the rule engines easy to write and maintain.

Note the example has been distilled to its core functionality, so is rather simplistic; the same functionality could be achieved in a single rule engine that simply parses more data. However, consider the question of ongoing support and updates. By separating the complex correlation into multiple rule engines, the whole system becomes far easier to maintain over time than with a single, monolithic rule engine. What if you want to start checking for a new class of hardware failure? Rather than modify (and potentially break) a single rule engine, a new (simple) rule engine is created that looks for this new type of failure that injects information about its findings into the log stream. The master rule engine then only needs a minor modification (or, perhaps, none at all) to take advantage of the new data.

Code Example 4: Chaining Rule Engines

```
#### Rule engine 1: Watches for hosts with bad RAM ####
# filter: match("ECC Error") or filter(f_thresholds);
# list threshold will alert when the value of a key exceeds 100.
$variable_server->set_list_threshold("ONE", "ecc_errors", ">=", 100);
while(<STDIN>) {
    my %message = parse($_);
    # an ECC error, hit the appropriate incrementer in the ecc_errors list
    if($message{message} =~ /ECC Error/) {
        # We use a 60-second timeout on the hit, which makes the list threshold
        # above alert at an ECC error rate of >= 100 per minute.
        $variable_server->set_list("ecc_errors", $message{from_host}, "+60");
    }
    if($message{message} =~ /THRESHOLD EXCEEDED.*ecc_errors.*\[(\w+)\].*\((\d+)\)/) {
        # We have found a bad host.  Generate a log message that Rule engine 3 will
        # pick up that indicates just how bad things are.
        my ($bad_host, $count) = ($1, $2);
        # pseudo-function for brevity; in reality this would queue an inject_msg action
        # that injects a message into the log stream at the given facility and priority.
        queue_alert("daemon", "info", "ALERT: $bad_host with $count ECC errors per minute");
    }
}

#### Rule engine 2: Watches for hosts with bad DISKS ####
# filter: match("EXT3 Error") or filter(f_thresholds);
# list threshold will alert when the value of a key exceeds 20.
$variable_server->set_list_threshold("ONE", "ext3_errors", ">=", 20);
while(<STDIN>) {
    my %message = parse($_);
    # an EXT3 error, hit the appropriate incrementer in the ext3_errors list
    if($message{message} =~ /EXT3 Error/) {
        # We use a 3600-second timeout on the hit, which makes the list threshold
        # above alert at an EXT3 error rate of >= 20 per hour.
        $variable_server->set_list("ext3_errors", $message{from_host}, "+3600");
    }
    if($message{message} =~ /THRESHOLD EXCEEDED.*ext3_errors.*\[(\w+)\].*\((\d+)\)/) {
        # We have found a bad host.  Generate a log message that Rule engine 3 will
        # pick up that indicates just how bad things are.
        my ($bad_host, $count) = ($1, $2);
        # pseudo-function for brevity; in reality this would queue an inject_msg action
        # that injects a message into the log stream at the given facility and priority.
        queue_alert("daemon", "info", "ALERT: $bad_host with $count EXT3 errors per hour");
    }
}

#### Rule engine 3: Watches for hosts with bad hardware ####
# filter: facility(daemon) and priority(info) and match("ALERT:");
while(<STDIN>) {
    my %message = parse($_);
    if($message{message} =~ /ALERT: (\w+) with (\d+) (EXT3|ECC) errors per (minute|hour)/) {
        my ($bad_host, $count, $type, $base) = ($1, $2, $3, $4);
        # Use a list variable to keep track of the various types of problems a system has.
        $variable_server->set_list("multi_problem_$bad_host", $type, "+3600");

        # If the system has only one thing wrong with it, we take action A, but if there are
        # two things wrong with it we take action B.
        if($variable_server->get_list_keys("multi_problem_$bad_host") >= 2) {
            queue_action("B"); }
        elsif($variable_server->get_list_keys("multi_problem_$bad_host") >= 1) {
            queue_action("A"); }
    }
}
```

This ability to chain rule engines means that extraordinarily complex correlations can be achieved by reducing them to their constituent parts, then chaining rule engines together to track progress through the correlation. Combine this functionality with the fact that rule engines can share temporal variable information through the variable server, and you have an extremely flexible and powerful system for correlating events and taking appropriate actions that is much easier to create, modify, and troubleshoot than other event correlation tools.

5 Useful Rule Engines at AMD

Since VTEC has been in production at AMD for more than four years, we have amassed a significant number of rule engines that perform useful event correlation and self-healing tasks in our computing environment.

Failed Hardware

One of the earliest uses for VTEC at AMD was to look for systems with bad hardware. We found most healthy systems would occasionally report ECC and EXT3 errors, but systems with truly bad hardware would send these errors at a noticeably higher rate. We implemented a rule engine that checks for these messages (the ECC error check uses a chained rule engine that aggregates the multi-line machine check errors that the Linux kernel reports) and then closes bad machines to new compute jobs. Systems in such bad shape that they are streaming error messages into the logs exceed an "emergency" threshold and a signal is sent to immediately power down the affected machine.

NFS File Server Checks

As summarized and simplified in Code Example 2, we have a rule engine that monitors client systems for "NFS server not responding" messages and alerts the storage and networking teams when the rate of messages exceeds a certain threshold.

Reboot Loops

We have all of our servers at AMD configured to send a syslog message when they have finished booting up. A VTEC rule engine watches for these messages and alerts operational staff if a machine is rebooting more than five times in a 90-minute period. These reboot loops can indicate a multi-bit ECC error, kernel panic, or other system configuration problem that requires the attention of a sysadmin.

Interactive Load Monitor Collator

At AMD we have several interactive login servers that are used by our design engineers as gateways into remote datacenters. Some engineers, instead of using our batch scheduling system, will run their jobs directly on the interactive login server, causing performance problems and occasionally even crashing the systems. We have implemented a cronjob that checks for processes that violate our interactive server usage policy, that sends out syslog messages when it detects a process that violates policy. A VTEC rule engine collects these messages, collates them, and generates a daily report for each engineer that is violating the policy with a summary of all of their processes at each site that are in violation of the policy. Additionally, the whole list of users and processes is sent in a daily digest to our interactive server support team, which can determine if it is appropriate to forcibly kill ill-behaved processes.

Out of Memory Tracking

A common problem in AMD's compute grids is out-of-memory conditions. Leaky jobs, or jobs that simply need more memory than is installed on the machine, will cause the system to run out of memory; the kernel then invokes the out-of-memory killer (OOM killer). The syslog message that the OOM killer generates has very little useful data: just a PID and a process name. To identify the owner of the job that caused the OOM condition, we run a cron job every five minutes on our systems that caches the contents of `/proc/<pid>/stat` into a directory under `/var/spool`. When a rule engine sees the OOM killer event, it queues a job ten minutes into the future (to give the server time to recover from the OOM condition) that logs into the machine and fetches the `/var/spool/proc/<pid>/stat` file that was referenced by the OOM killer message. This file tells us not only who was running the errant job, but also how large it was when it was killed. This information is used to craft an e-mail alert to the user asking them to check the status of their batch job submissions to prevent more problems on other compute nodes.

Automatic System Stress Testing

When bringing new systems online in our compute grid, it is important to stress-test them before allowing engineer's jobs on them. We accomplish this at AMD by using a rule engine that watches for syslog messages indicating that a system has completed its automated installation sequence. It then closes the machine to the batch submission system, kicks off the stress test, and sets an alarm for 12 hours in the future. If the alarm goes off, the stress test must have locked up the system; an operational staff member is alerted to check the system. If the stress test completes (notifying of its status via syslog, of course) the system is either opened for jobs (test was successful) or is left closed (test failed).

Ignoring Flood-Generating Hosts

Sometimes a machine will be so thoroughly broken that it is impossible to shut down the machine, and it just sits there spewing error messages into the syslog stream. This can hamper the VTEC server not only because it can cause it to run out of disk space, but also because it forces syslog-ng to process a flood of irrelevant data. We have implemented a rule engine that watches for hosts sending an excessive volume of log data to the system, and automatically updates the `syslog-ng.conf` file with a filter that drops all of the messages from that host for 24 hours. Once the 24-hour alarm expires, the filter is removed; if the system is still spewing messages, it will be re-ignored within a few minutes, preventing the log flood from adversely affecting performance or data space on the VTEC server.

6 Designing for Performance

The performance characteristics of VTEC can be best described by analyzing the potential bottlenecks of each component, since the slowest single component will likely represent the limiting factor for the overall system. However, due to the multi-core design of VTEC, even if one of the components (usually the variable server) maxes out a CPU, the other components can continue functioning without any degradation on other CPUs.

Syslog-ng

Methods for tuning the performance of syslog-ng are fairly sparsely documented in the reference manual [9]. In the context of VTEC, we are most concerned with the performance of the filtering functions, especially since syslog-ng is a single-threaded process. If syslog-ng spends too much time on the CPU trying to parse through log filters, it could begin dropping log messages.

We have found after much experimentation that syslog-ng's `match()` filter function is much slower than the other message filtering functions. When rule engines are configured with nothing but a `match()` rule, the syslog-ng process spends all of its time on the CPU, even with a fairly modest (1-2GB/day) rate of traffic. Simply adding one of the "fast" functions in addition to the `match()` function returns performance to acceptable levels, presumably by short-circuiting the `match()` function in many irrelevant cases.

In general, this means that a filter rule like this:

```
match(".*some message.*")
```

can be written better as:

```
facility("daemon")  and  priority("info")
and match(".*some message.*")
```

to get acceptable performance from syslog-ng.

A typical VTEC server at AMD runs on a four-vCPU virtual machine with 2.9GHz AMD Opteron^{TM} processors under the hypervisor. The syslog-ng instance filters data into 21 rule engines and four local disk logs. The incoming data rate is about 1,000 messages/sec and 10GB/day. With that level of load, the syslog-ng process consumes about 30% of one CPU core, with occasional spikes to above 50% of one CPU core.

Temporal Variable Server

As demonstrated in the examples, the temporal variable server is a critical part of the speed path as log messages route their way through the system. Since virtually every log message that makes its way to a rule engine results in at least one (and many times multiple) requests to the variable server, it is important that it be able to respond quickly.

The variable server is implemented using Perl threads. This allows for simple sharing of the internal database of temporal variables (a hash for

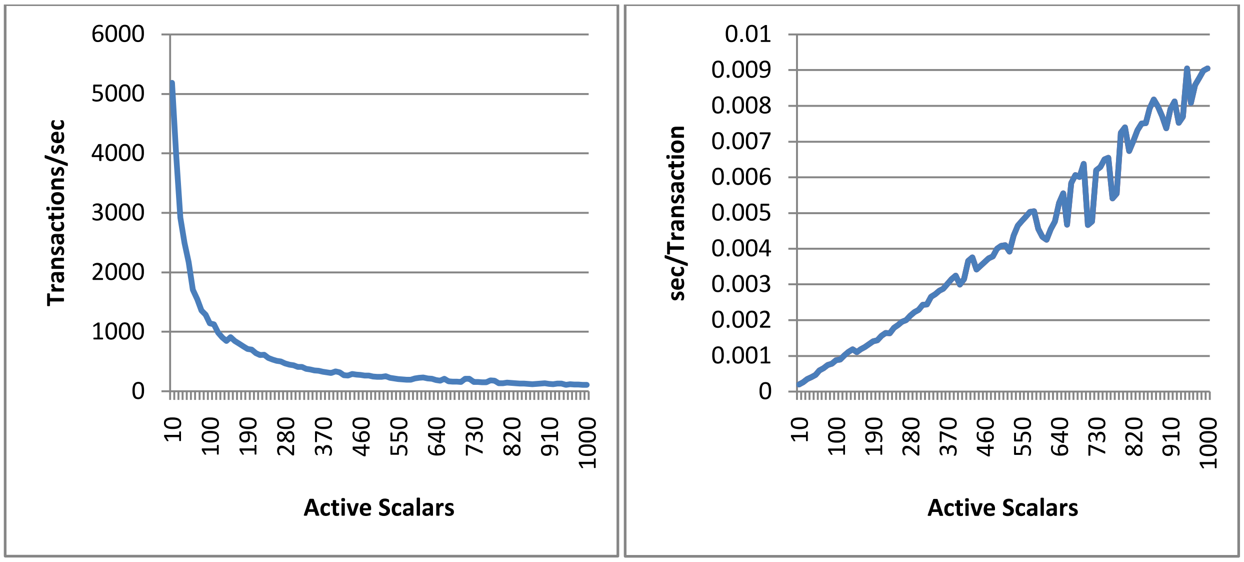


Figure 3: Variable Server Performance Querying a Growing Incrementer

each data type). Six threads run in parallel, using a collection of locks to ensure atomic write access to the internal data structures:

- A server thread that listens for connections (and spawns threads that handle those connections)
- A backup thread that wakes up every ten seconds and backs up the internal state to disk
- Three monitoring threads that wake up every 60 seconds to purge stale data from the internal data structures
- A thresholds thread that wakes up every 60 seconds to scan for variables that have exceeded thresholds.

The persistent threads do not pose any real performance issues; they wake up infrequently, and spend only a few milliseconds to complete their duties before going back to sleep.

The threads that get spawned to service incoming requests are the potential bottleneck. Testing has revealed that for basic requests (e.g. fetch the value of a scalar), performance scales very well. However, as the data structures being queried get more complex (e.g. fetch the value of an incrementer that has 1,000 active scalars in it), performance begins to degrade. Figure 3 shows the transactional performance of the variable server as the size of an incrementer being queried grows in size.

The routine that calculates the value of an incrementer is `O(n)` with the number of active scalars it contains. An updated routine that is `O(log n)` is being tested and shows promising results, but requires significant code changes and so has not been deployed into production yet.

Rule Engines

Obviously the rule engines, since they can have arbitrary code in them, can be a bottleneck. But the idea behind VTEC is for the rule engines to take advantage of the action server, precisely so they don't have to block for long-running tasks. If the rule engines are coded such that they do not block, they do not represent a bottleneck.

7 Challenges

As flexible and effective as VTEC currently is, there is always room for improvement. We currently face two issues with VTEC.

Variable Server Performance

As described in Section 6, the variable server is the major bottleneck. Since virtually all rule engines depend on incrementers and lists (which are collections of incrementers), the fact that the incrementer data type scales so poorly is a significant hindrance. However, there is new code in the works for the incrementer that should greatly improve its scalability and performance.

Feedback from Actions

The action server is currently a "fire and forget" system. When jobs are queued, the rule engine can get a job ID for tracking, but there is currently nothing useful that can be done with it. There is no way to query the status of job, or to collect the output of a job, from within a rule engine. Jobs have the option to use `logger` or some other means to inject syslog messages and "phone home" to their parent rule engine, but this process is awkward and is not used in practice.

A major improvement in this area would be to have the action server cache job status information and output for some period of time after the job completes, so rule engines can query for it. Having the action server automatically inject messages into the log stream when jobs complete would also help, because it would give rule engines a trigger to work with to keep up with the jobs they've queued.

8 Conclusion

AMD has used VTEC since 2006 to monitor and automate maintenance activities on its large compute grids. Log volumes range up to 10 GB/day with VTEC running smoothly on modest two- to four-core virtual and physical machines. VTEC tracks hardware problems such as disk, CPU, and RAM failures and takes appropriate actions (e.g., shut down/close the broken system and create a ticket). VTEC can monitor the environment for trends that indicate events (e.g., *n* systems are unable to contact *m* NFS filers, so there must be a network problem). Most importantly, VTEC enables autonomic computing by allowing intelligent dispatch of repair jobs in response to detected problems. If these repair jobs fail to work, VTEC can notify humans to take over.

In summary, VTEC is a powerful tool for automating log analysis and event correlation. While there are many other tools that perform similar tasks, VTEC's approach to the problem presents a complete, scalable, and intuitive solution that is able to grow and adjust to virtually any workload.

9 Author Biography

Paul Krizak is currently a member of the technical staff at AMD and has been a professional systems engineer for more than seven years. His duties at AMD have generally surrounded infrastructure support of the large engineering research and development grids, most commonly in support of the autonomic Linux OS provisioning and configuration infrastructure. He graduated with a bachelor's degree in computer science from Texas A&M University in 2005.

10 References

1. Rouillard, John P. "Real-time Log File Analysis Using the Simple Event Correlator (SEC)." *Proceedings of LISA XVIII* (2004): 133-49. Print.
2. *SEC - Open Source and Platform Independent Event Correlation Tool*. Web. 22 July 2010. <http://simple-evcorr.sourceforge.net/>.
3. *Splunk | IT Search for Log Management, Operations, Security and Compliance*. Web. 22 July 2010. <http://www.splunk.com>.
4. M. Burgess. On the theory of system administration. *Science of Computer Programming*, 49:1, 2003. Print.
5. *Cfengine - Automatic Server Lifecycle Management*. Web. 22 July 2010. <http://www.cfengine.com>.
6. "Syslog Server | Syslog-ng." *Gateway Solution | Network Security | BalaBit IT Security*. Web. 22 July 2010. <http://www.balabit.com/network-security/syslog-ng/>.
7. Hansen, Stephen E., and Todd Atkins. "Automated System Monitoring and Notification with Swatch." *Proceedings of LISA VII* (1993): 145-52. Print.
8. "Logwatch on SourceForge.net." *SourceForge.net*. Web. 22 July 2010. <http://sourceforge.net/projects/logwatch/files/>.
9. "Syslog-ng V2.0 Reference Manual." *Syslog: Main/Home Page*. Web. 23 July 2010. <http://www.syslog.org/syslog-ng/v2/>.
10. Finke, Jon, "Process Monitor: Detecting Events That Didn't Happen," *Proceedings of LISA XVI* (2002): 145-153. Print.

Chukwa: A system for reliable large-scale log collection

Ariel Rabkin
asrabkin@cs.berkeley.edu
UC Berkeley

Randy Katz
randy@cs.berkeley.edu
UC Berkeley

Abstract

Large Internet services companies like Google, Yahoo, and Facebook use the MapReduce programming model to process log data. MapReduce is designed to work on data stored in a distributed filesystem like Hadoop's HDFS. As a result, a number of log collection systems have been built to copy data into HDFS. These systems often lack a unified approach to failure handling, with errors being handled separately by each piece of the collection, transport and processing pipeline.

We argue for a unified approach, instead. We present a system, called Chukwa, that embodies this approach. Chukwa uses an end-to-end delivery model that can leverage local on-disk log files for reliability. This approach also eases integration with legacy systems. This architecture offers a choice of delivery models, making subsets of the collected data available promptly for clients that require it, while reliably storing a copy in HDFS. We demonstrate that our system works correctly on a 200-node testbed and can collect in excess of 200 MB/sec of log data. We supplement these measurements with a set of case studies describing real-world operational experience at several sites.

Keywords: logging, scale, research

1 Introduction

Almost every distributed service generates logging data. The rise of Cloud computing makes it easier than ever to deploy services across hundreds of nodes [4], with a corresponding increase in the quantity of logs and the difficulty of manual debugging. Automated log analysis is increasing the amount of information that can be extracted from logs, thus increasing their value [26, 30, 3, 16]. Hence, log collection and processing is increasingly important. Scalable data processing is challenging and so it is very desirable to leverage existing tools.

MapReduce is emerging as a standard tool for data-intensive processing of all kinds, including log file analysis [10, 29]. Tasks like indexing and aggregation fit naturally into the MapReduce paradigm. So do more sophisticated analyses, such as machine learning-based anomaly detection using console logs [30].

In this paper, we present Chukwa, a scalable system for collecting logs and other monitoring data and processing the data with MapReduce. Today, an administrator seeking to use MapReduce for system analysis would need to build a great deal of infrastructure to connect data sources with processing tools. Several sites have built such tools [23, 29], but each has been highly tailored to the specific context at hand. All have flawed failure recovery mechanisms, potentially leading to data loss. In contrast, Chukwa is designed to integrate cleanly with a wide variety of legacy systems and analysis applications and to offer strong reliability guarantees. It is available as open-source software and is currently in use at a number of sites, including Berkeley, Selective Media, and CBS Interactive. A previous publication described our initial goals and our prototype implementation [7]. In this paper, we describe Chukwa's design in more detail, present performance measurements, and describe real-world experiences.

1.1 Why distributed log collection is difficult

While MapReduce is a powerful and increasingly popular tool, there is a tension between its performance characteristics and those of many log collection workloads. One of the major design principles of MapReduce is to push computation to the node holding the associated data. This is accomplished by storing the input to a MapReduce job in a distributed filesystem such as the Google File System (GFS) [12], or its open-source counterpart, the Hadoop Distributed File System (HDFS). GFS and HDFS are user-level filesystems that do not implement POSIX semantics and that do not integrate with the OS filesystem layer. Both MapReduce and the underlying filesystems are heavily optimized for the case of large files (measured in gigabytes) [6]. This means that applications must either be modified to write their logs to these filesystems, or else a separate process must copy logs into the filesystem.

This problem would be comparatively easy in a distributed filesystem that allowed multiple concurrent appends and where writes never failed. But such systems are quite difficult to build; no existing filesystem has both properties, and no system available in the open-source world has either. Support for single-writer non-concurrent appends has been in-progress in Hadoop for several years, despite implementation effort by a large population of paid full-time developers.

As a result, the implementation strategy adopted by the open-source world has been to implement this functionality in application code. In the standard approach, processes send their log messages across the network to a daemon, commonly called a collector, that serializes the updates and writes them to the filesystem. Several companies, such as Rapleaf, Rackspace, and Facebook [23, 29, 1], have built specialized log collection systems of this type.

These collection systems have largely treated log collection as just another network service. They expose a narrow interface to clients commonly using remote procedure call (RPC). The monitoring system is responsible for receiving and recording data and plays no role once data has been written to the distributed filesystem. While this separation of concerns is normally an attractive design style, we argue that it is the wrong approach for reliable monitoring of monitoring legacy systems.

A common task for a monitoring system is to collect data from legacy application log files on disk. Ideally, files on disk would be deleted once their contents have been stored durably by the monitoring system. But this is impossible without some way for the monitoring system to report back success or failure. In the event of a transitory failure, data may be buffered by the monitoring system for some time, meaning that a synchronous RPC model, with success or failure reported as soon as data is sent, is insufficient. This problem is perhaps less significant in organizations like Facebook or Google, where legacy code can be rewritten. But in smaller organizations, it looms large as a problem.

1.2 Our innovations

Our system, Chukwa, adopts a different architecture. Rather than expose a narrow interface to the monitoring system, we try to confine as much complexity as possible as close as possible to the application being monitored. This means that the interface between the system being monitored and the monitoring system is highly flexible and can be tailored to a particular context. It also means that the rest of the monitoring system can be simple and optimized for the common case. This enables substantial design simplification and good performance while offering superior reliability guarantees.

In Chukwa, data is collected by a dedicated agent process on each machine being monitored. This process can hold far more application-specific functionality than the simple network services offered by systems such as Scribe. As we show, this enables us to easily support a range of desirable features not found in alternative monitoring systems. Agents are responsible for three important tasks: producing metadata, handling failures, and integrating with existing data sources.

- Unlike other systems, Chukwa has a rich metadata model, meaning that semantically-meaningful subsets of data are processed together. This metadata is collected automatically and stored in parallel with data. This eases the development of parallel, scalable MapReduce analyses.

- We push failure handling and data cleaning to the endpoints of the monitoring system. Each agent is responsible for making sure that data is stored at least once. A MapReduce job removes duplicates. As a result, the interior of the collection system can be much simpler and can optimize for the common case where writes succeed.

- Last, we optimize for the case of log files on local disk. Such logs are common in many environments. Logs on disk are easy to create, easy to reason about, and robust to many failures. They are commonly produced by legacy systems. Chukwa demonstrates that these logs can also be used for low-cost failure recovery. From the point of view of Chuwka agents, data collection is asynchronous and need not be reliable. If a timer expires before data is stored durably, agents re-send using the on-disk log.

While Chukwa is optimized for logs on disk, it can handle many other monitoring tasks. Chukwa can collect a variety of system metrics and can receive data via a variety of network protocols, including `syslog`. Our reliability model encompasses these sources naturally and flexibly. Depending on user preferences, each data source can be buffered to disk pessimistically, buffered on error, or not buffered.

This work is timely for two reasons. The development of automated log analysis (such as [30, 3, 16] has made system logs much more useful. If logs are rarely consulted, then collecting them is a low priority. Now that system logs can be analyzed automatically and continuously, collecting them becomes a much higher priority.

The rise of Cloud computing makes it easier than ever to deploy services across hundreds of nodes [4], with a corresponding increase in the quantity of logs. At that scale, sophisticated storage and analysis tools like Hadoop become very desirable.

We begin, in the next section, by describing our design goals and assumptions and explaining why existing architectures do not adequately meet them. Section 3 describes our concrete implementation and Section 4 presents quantitative measurements. Section 5 discusses deployment experience. We describe related work in Section 6 and summarize our conclusions in Section 7.

2 Design Goals and Alternatives

Many monitoring and log collection systems have been built before Chukwa. In this section, we discuss our goals and why existing systems fail to meet them. These goals were based on design discussions at both Yahoo! and UC Berkeley and reflect real operational needs.

2.1 Supporting Production Use

We first list the core set of requirements needed to monitor production systems.

- The system must support a wide variety of data sources, not just log files. This is needed to collect system metrics and to cope with existing legacy systems that sometimes use other logging protocols, such as `syslog` [15].
- If the monitoring system fails, the system being monitored should continue working without interruption or slowdown.
- The system should scale to handle large numbers of clients and large aggregate data rates. Our target was to support 10,000 hosts and 30 MB/sec, matching the largest clusters currently in use at Yahoo [7].
- The system should impose low overhead. We have often heard 5% described as the largest fraction of system resources that administrators are comfortable devoting to monitoring. Lacking any more principled standard, we have adopted this as our target maximum resource utilization for the monitoring system.
- No matter how intense a bust of log writes, the resource consumption of the monitoring system should remain with its resource bounds.

Some log analysis jobs are very sensitive to missing data. In general, whenever the absence of a log message is significant to an analysis, losing even a small quantity of data can result in a badly wrong answer. For instance, Rackspace uses a MapReduce-based analysis of email logs to determine the precise path that mail is taking through their infrastructure [29]. If the log entry corresponding to a delivery is missing, the analysis will wrongly conclude that mail was lost. The machine-learning based log file analysis developed by Xu *et al.* [30] is another example of a loss-sensitive analysis. And of course, if web access logs are used for billing purposes, lost messages translate directly into lost revenue. To support these sorts of log analysis applications, we made reliable delivery a core goal for Chukwa.

Two of our goals conflict. A system cannot both offer reliable delivery in all circumstances while never having the system being monitored block while waiting for the monitoring system. Local storage is limited, meaning that if the monitoring system is unavailable for a long time, the system being monitored must either discard data or block. To reconcile these goals, we adopted the following reliability standard: if the machine originating the data stays does not fail permanently, data will eventually be delivered.

Making data available to MapReduce in less than a minute or two was not a goal. Chukwa was primarily designed to enable MapReduce processing of log data. Due to scheduling overheads, a Hadoop MapReduce job seldom executes in less than a minute. As a result, reducing data delivery latency below a minute offers limited benefit.

2.2 Why existing architectures are inadequate

Perhaps surprisingly, existing monitoring systems and architectures are inadequate to meet the goals listed above. The oldest and simplest form of logging is writing to local disk. Local disk writes are low-cost, and have predictable performance. Unfortunately, processing data scattered across local disks of a cluster is difficult. Doing so while processing data in-place will result in analysis workloads and production loads conflicting, which is often unacceptable in practice.

Doing processing on a separate analysis cluster requires some way of moving data from source to destination. A shared NFS mount and streaming data via `syslog` are two standard ways to do this. These two approaches make contrasting reliability-availability choices. If the network fails, an NFS write will fail, blocking the application. Syslog, built on UDP, will silently discard data.

That leaves writing data locally, either on failure or before attempting to copy it to HDFS. We discuss each in turn. Several systems, notably Scribe [1] attempt to write data across the network, and buffer locally only on failure. The catch is that clients do not participate in failure recovery. Data loss will be fatal if a crash occurs after data has been handed to Scribe, and before that data has been stored durably. Likewise, data can be lost if a filesystem write returns success before data is serialized. (This can happen because HDFS buffers aggressively before flushing data to remote hosts.) As a result, pessimistic logging by the application is required to achieve high reliability.

For large files, copied periodically and all-at-once, this is simple to implement and works well. For streaming data, however, complexities emerge. There is an "impedance mismatch" between many logging workloads and the optimal performance envelope for scalable distributed MapReduce-friendly file systems like HDFS. Those file systems are designed for a small number of large files, written once and never updated. In contrast, large numbers of small log files updated sporadically are an important kind of monitoring data. Easing this gap requires consolidating logs from many machines into one file. Since HDFS lacks concurrent appends, this requires a separate process to do the merging. This increases the number of points at which failures can occur.

Chukwa responds to this reliability problem in an end-to-end manner, by pushing the retry logic as close to the data source as possible. Data is either stored in log files on local disks, or else in HDFS. No other copies are made, by default. Data transmission is only treated as successful once data from the one source has been successfully copied to the other. The technical challenge is two-fold. Reliability needs to be integrated with legacy applications that may be oblivious to the monitoring system. And this comparison must be performed efficiently and continuously at run-time.

Not all sources or uses of log data require the same degree of reliability. A site might decide that pessimistically recording all system metrics to disk is an unnecessary and wasteful degree of robustness. An additional design goal for us was to avoid imposing excessive costs for collecting this sort of ephemeral data.

2.3 A choice of delivery models

It became clear to us as we were developing Chukwa that in addition to reliability-sensitive applications, there is an another class of applications with quite different needs. It is sometimes desirable to use logs to drive an ongoing decision-making process, such as whether to send an alert to an administrator based on a critical error or whether to scale up or scale down a cloud service in response to load. These applications are perforce less sensitive to missing data, since they must work correctly even if the node that generated the missing data crashes.

Reliable delivery	**Fast-path delivery**
Visible in minutes	Visible in seconds
Writes to HDFS	Writes to socket
Resends after crash	Does not resend
All data	User-specified filtering
Supports MapReduce	Stream processing
In order	No guarantees

Table 1: The two delivery models offered by Chukwa

To support latency-sensitive applications, we offer an alternate "fast path" delivery model. This model was designed to impose minimal delays on data delivery. Data is sent via TCP, but we make no other concession to reliable delivery on this path. Applications using the fast path can compensate for missing data by inspecting the reliably-written copy on HDFS. Table 1 compares these two delivery models.

3 Architecture

In the previous section, we described our design goals. In this section, we describe our design and how it achieves these goals. Like the other systems of this type, we introduce auxiliary processes between the log data and the distributed filesystem. Unlike other systems, we split these processes into two classes. One set of processes, the *collectors*, are responsible for writing to HDFS and are entirely stateless. The other class, the *agents* run on each machine being monitored. All the state of the monitoring system is stored in agents, and is checkpointed regularly to disk, easing failure recovery. We describe each half of the system in turn. We then discuss our data model and the fault-tolerance approach it enables. Figure 1 depicts the overall architecture.

3.1 Agents

Recall that a major goal for Chukwa was to cleanly incorporate existing log files as well as interprocess communication protocols. The set of files or sockets being monitored will inevitably grow and shrink over time, as various processes start and finish. As a result, the agent process on each machine needs to be highly configurable.

Most monitoring systems today require data to be sent via a specific protocol. Both `syslogd` and Scribe [15, 1] are examples of such systems. Chukwa takes a different approach. In Chukwa, agents are not directly responsible for receiving data. Instead, they provide an execution environment for dynamically loadable and configurable modules called *adaptors*. These adaptors are responsible for reading data from the filesystem or directly from the application being monitored. The output

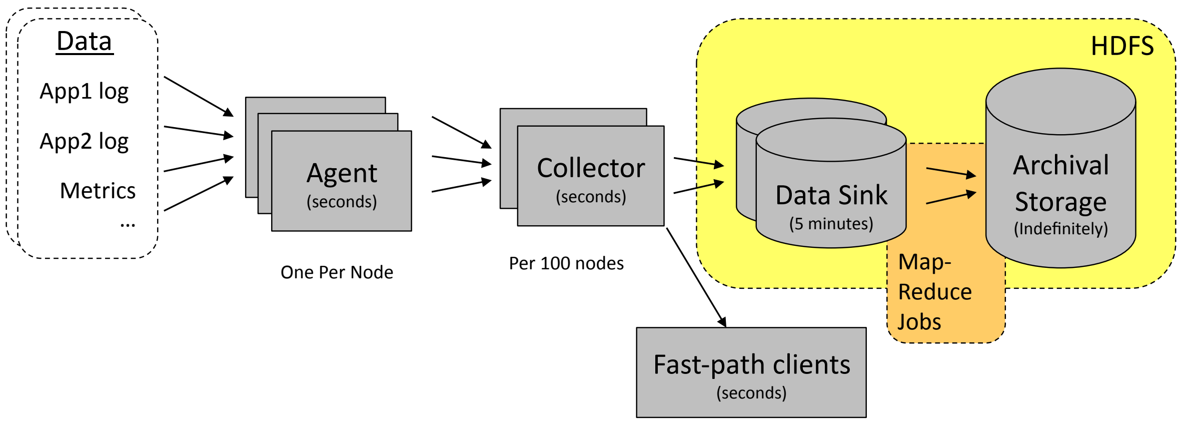


Figure 1: The flow of data through Chukwa, showing retention times at each stage.

from an adaptor is conceptually a stream of consecutive bytes. A single stream might correspond to a single file, or a set of repeated invocations of a Unix utility, or the set of packets received on a given socket. The stream abstraction is implemented by storing data as a sequence of *chunks*. Each chunk consists of some stream-level metadata (described below), plus an array of data bytes.

At present, we have adaptors for invoking Unix commands, for receiving UDP messages (including `syslog` messages), and, most importantly, for repeatedly "tailing" log files, sending any data written to the file since its last inspection. We also have an adaptor for scanning directories and starting a file tailing adaptor on any newly created files.

It is possible to compose or "nest" adaptors. For instance, we have an adaptor that buffers the output from another adaptor in memory and another for write-ahead logging. This sort of nesting allows us to decouple the challenges of buffering, storage, and retransmission from those of receiving data. This achieves our goal of allowing administrators to decide precisely the level of failure robustness required for each data stream.

The agent process is responsible for starting and stopping adaptors and for sending data across the network. Agents understand a simple line-oriented control protocol, designed to be be easy for both humans and programs to use. The protocol has commands for starting adaptors, stopping them, and querying their status. This allows external programs to reconfigure Chukwa to begin reading their logs.

Running all adaptors inside a single process helps administrators impose resource constraints, a requirement in production settings. Memory usage can be controlled by setting the JVM heap size. CPU usage can be controlled via `nice`. Bandwidth is also constrained by the agent process, which has a configurable maximum send rate. We use fixed-size queues inside the agent process, so if available bandwidth is exceeded or if the collectors are slow in responding, then back-pressure will throttle the adaptors inside the process [28].

The agent process periodically queries each adaptor for its status, and stores the answer in a checkpoint file. The checkpoint includes the amount of data from each adaptor that has been committed to the distributed filesystem. Each adaptor is responsible for recording enough additional state to be able to resume cleanly, without sending corrupted data to downstream recipients. Note that checkpoints include adaptor state, but not the underlying data. As a result, they are quite small – typically no more than a few hundred bytes per adaptor. This allows Chukwa to scale to many hundreds or thousands of files being monitored.

One challenge in using files for fault-tolerance is correctly handling log file rotation. Commonly, log files are renamed either on a fixed schedule, or when the reach a predetermined size. When this happens, data should still be sent and sent only once. In our architecture, correctly handling log file rotation is the responsibility of the adaptor. Different adaptors can be implemented with different strategies. Our default approach is as follows: If instructed to monitor log file `foo`, assume that any file starting with `foo.*` is a rotated version of `foo`. Use file modification dates to put rotated versions in the correct order. Store the last time at which data was successfully committed and the associated position in the file. This is enough information to resume correctly after a crash.

3.2 Collectors

We now turn to the next state of our architecture, the collectors. If each agent wrote directly to HDFS, this would result in a large number of small files. Instead, Chukwa uses the increasingly-common collector technique mentioned in the introduction, where a single process multiplexes the data coming from a large number of agents.

Each collector writes the data it receives to a single output file, in the so-called "data sink" directory. This reduces the number of files generated from one per machine or adaptor per unit time to a handful per cluster. In a sense, collectors exist to ease the "impedance mismatch" between large numbers of low-rate sources and a filesystem that is optimized for a small number of high-rate writers. Collectors periodically close their output files, rename the files to mark them available for processing, and begin writing a new file. We refer to this as "file rotation." A MapReduce job periodically compacts the files in the sink and merges them into the archive of collected log data.

Chukwa differs in several ways from most other systems that employ the collector design technique. We do not make any attempt to achieve reliability at the collector. Instead, we rely on an end-to-end protocol, discussed in the next section. Nor do Chukwa agents dynamically load-balance across collectors. Instead, they try collectors at random until one appears to be working and then use that collector exclusively until they receive errors, at which point they fail-over to a new one. The benefit of this approach is that it bounds the number of agents that will be affected if a collector fails before flushing data to the filesystem. This avoids a scaling problem that would otherwise occur where every agent is forced to respond to the failure of any collector. One drawback is that collectors may be unevenly loaded. This has not posed any problems in practice since in a typical deployment the collectors are far from saturated. With a collector on every HDFS node, we have found that the underlying filesystem saturates well before the collectors do.

To correctly handle overload situations, agents do not keep retrying indefinitely. If writes to a collector fail, that collector is marked as "bad", and the agent will wait for a configurable period before trying to write to it again. Thus, if all collectors are overloaded, an agent will try each, fail on each, and then wait for several minutes before trying again.

Collectors are responsible for supporting our "fast path" delivery model. To receive data using this model, clients connect to a collector, and specify a set of regular expressions matching data of interest. (These regular expressions can be used to match either content or the Chukwa metadata, discussed in the next subsection.) Whenever a chunk of data arrives matching these filters, it is sent via a TCP socket to the requesting process in addition to being written to HDFS. To get full coverage, a client needs to connect to every collector. As we will show in the next section, a modest number of collectors are sufficient for the logging needs of large datacenter services. Hence, "every collector" is often only a handful.

Filtering data at collectors has a number of advantages. In the environments we have seen, collectors are IO-bound, not CPU-bound, meaning that CPU resources are available for the pattern matching. Moreover, collectors are stateless, meaning that it is straightforward to spread out this matching across more machines, if need be, by simply adding more collectors.

The fast path makes few reliability promises. Data can be duplicated, if an agent detects a collector failure and resends. Data can be lost, if the collector or the data recipient fails. In some failure scenarios, data can be received out of order. While data is normally delivered to clients as soon as it is received by the collector, it can be delayed if the network is congested. One guarantee the fast path does make is that each individual chunk of data will be received correctly or not at all. As we will see, this guarantee is enough to be useful.

On the regular "reliable path", collectors write their data in the standard Hadoop sequence file format. This format is specifically designed to facilitate parallel processing with MapReduce. To reduce the number of files and to ease analysis, Chukwa includes an "archiving" MapReduce job that groups data by cluster, date, and data type. This storage model is designed to match the typical access patterns of jobs that use the data. (For instance, it facilitates writing jobs that purge old data based on age, source, and type: "Store user logs for 14 days, and framework logs for one year.") The archiving job also detects data loss, and removes duplicate data. Repeated invocations of this job allow data to be compacted into progressively larger files over time.

This stored data can be used in a number of ways. Chukwa includes tools for searching these files. The query language allows regular-expression matches against the content or metadata of the stored data. For larger or more complex tasks, users can run customized MapReduce jobs on the collected data. Chukwa integrates cleanly with Pig, a language and execution environment for automatically producing sequences of MapReduce jobs for data analysis [18].

3.3 Metadata

When agents send data, they add a number of metadata fields, listed in Table 2. This metadata serves two distinct purposes: uniquely identifying a chunk for purposes of duplicate detection, and supplying context needed for analysis. Three fields identify the stream. Two are straightforward: the stream name (e.g. `/var/log/datanode`) and source host. In addition, we also tag data with the "source cluster." In both clouds and datacenters, users commonly allocate virtual clusters for particular tasks and release them when the task is complete. If two different users each use a given host at different times, their logs may be effectively unrelated. The source cluster field helps resolve this ambiguity. Another field, the sequence ID, identifies the position of a given data chunk within that stream.

To these four fields, we add one more, "data type," that specifies the format of a chunk's data. Often, only a subset of the data from a given host is relevant to a given analysis. One might, for instance, only look at Hadoop Task logs. The datatype field lets a human or a program describe the logical content of chunks separately from the physical origin of the data. This avoids the need to separately maintain a table describing the semantics of each file or other physical data source.

Taken together, this metadata set allows MapReduce jobs to easily check if data is missing from a stream. (Data can be missing from a stream either for streams with reliable retransmission disabled, or as a transitory condition before a retransmission.) Missing data will show up as a gap between the sequence numbers for a pair of adjacent chunks, in precisely the same way that TCP sequence numbers allow dropped packets to be detected.

The Chukwa metadata model does not include time stamps. This was a deliberate decision. Timestamps are unsuitable for ordering chunks, since several chunks might be read from a file in immediate succession, resulting in them having identical timestamps. Nor are timestamps necessarily useful for interpreting data. A single chunk might correspond to many minutes of collected data, and as a result, a single timestamp at the chunk level would be misleading. Moreover, such timestamps are redundant, since the content of each chunk generally includes precise application-level timestamps. Standard log file formats include per-line timestamps, for instance.

3.4 Reliability

Fault-tolerance was a key design goal for Chukwa. Data must still arrive even if processes crash or network connectivity is interrupted. Our solution differs substantially from other systems that record logs to distributed storage and is a major contribution of this work. Rather than try to make the writer fault-tolerant, we make them stateless, and push all state to the hosts generating the data.

Handling agent crashes is straightforward. As mentioned above, agents regularly checkpoint their state. This checkpoint describes every data stream currently being monitored and how much data from that stream has been committed to the data sink. We use standard daemon-management tools to restart agents after a crash. When the agent process resumes, each active adaptor is restarted from the most recent checkpoint state. This means that agents will resend any data sent but not yet committed or committed after the last checkpoint. These duplicate chunks will be filtered out by the archiving job, mentioned above.

File tailing adaptors can easily resume from a fixed offset in the file. Adaptors that monitor ephemeral data sources, such as network sockets, can not. In these cases, the adaptor can simply resume sending data. In some cases, this lost data is unproblematic. For instance, losing one minute's system metrics prior to a crash does not render all subsequent metrics useless. In other cases, a higher reliability standard is called for. Our solution is to supply a library of "wrapper" adaptors that buffer the output from otherwise-unreliable data sources. Currently, users can choose between no buffering, buffering data in memory, or write-ahead logging on disk. Other strategies can be easily implemented.

Rather than try to build a fault tolerant collector, Chukwa agents look *through* the collectors to the underlying state of the filesystem. This filesystem state is what is used to detect and recover from failure. Recovery is handled entirely by the agent, without requiring anything at all from the failed collector. When an agent sends data to a collector, the collector responds with the name of the HDFS file in which the data will be stored and the future location of the data within the file. This is very easy to compute – since each file is only written by a single collector, the only requirement is to enqueue the data and add up lengths.

Every few minutes, each agent process polls a collector to find the length of each file to which data is being written. The length of the file is then compared with the offset at which each chunk was to be written. If the file length exceeds this value, then the data has been committed and the agent process advances its checkpoint accordingly. (Note that the length returned by the filesystem is the amount of data that has been successfully replicated.) There is nothing essential about the role of collectors in monitoring the written files. Collectors store no per-agent state. The reason to poll collectors, rather than the filesystem directly, is to reduce the load on the filesystem master and to shield agents from the details of the storage system. On error, agents resume from their last checkpoint and pick a new collector. In the event of a failure, the total volume of data retransmitted is bounded by the period between collector file rotations.

The solution is end-to-end. Authoritative copies of data can only exist in two places: the nodes where data was originally produced, and the HDFS file system where it will ultimately be stored. Collectors only hold soft state; the only "hard" state stored by Chukwa is the agent checkpoints. Figure 2 diagrams the flow of messages in this protocol.

Field	Meaning	Source
Source	Host where Chunk was generated	Automatic
Cluster	Cluster host is associated with	Configured by user per-host
Datatype	Format of output	Configured by user per-stream
Sequence ID	Offset of Chunk in stream	Automatic
Name	Name of data source	Automatic

Table 2: The Chukwa Metadata Schema

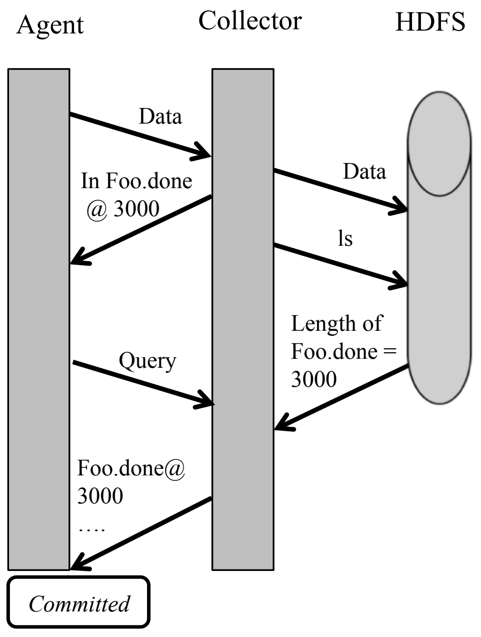


Figure 2: Flow of messages in asynchronous acknowledgement. Data written through collector without waiting for success. Separately, collectors check lengths of written files, and report this back to agents.

4 Evaluation

In this section, we will demonstrate three properties. First, Chukwa imposes a low overhead on the system being monitored. Second, Chukwa is able to scale to large data volumes. Third, that Chukwa recovers correctly from failures. To verify these properties, we conducted a series of experiments at scale on Amazon's Elastic Compute Cloud, EC2. Using EC2 means that our hardware environment is well-documented, and that our software environment could be well controlled. All nodes used the same virtual machine image, running Ubuntu Linux, with a 2.6.21 kernel. We used version 0.20.0 of the Hadoop File System.

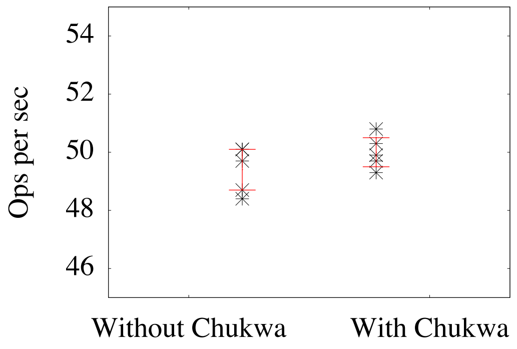


Figure 3: Cloudstone benchmark scores (HTTP requests per second), with and without Chukwa monitoring

4.1 Overhead of Monitoring

To measure the overhead of Chukwa in production, we used Cloudstone, a benchmark [24], designed for comparing the performance of web application frameworks and configurations. Each run takes about ten minutes to complete and outputs a score in requests handled per second for a standardized simulated workload. The version we used starts a large number of Ruby on Rails processors, backed by a MySQL database. We used a 9-node cluster, with Chukwa running on each host. Each node was an EC2 "extra large" (server class) instance. Chukwa was configured to collect console logs and system metrics. In total, this amounted to 60 KB per minute of monitoring data per node.

Our results are displayed in Figure 3. As can be seen, the runs with and without Chukwa were virtually indistinguishable. All of the runs within Chukwa performed within 3% of the median of non-Chukwa runs. This shows that the overhead of monitoring using Chukwa is quite modest. One run each with and without Chukwa failed, due to a bug in the current Cloudstone implementation. These have been excluded from Figure 3.

To test overhead with other workloads, we ran a series of Hadoop jobs, both with and without Chukwa. We used a completely stock Hadoop configuration, without any Chukwa-specific configuration. As a result, our results reflect the experience that a typical system would have

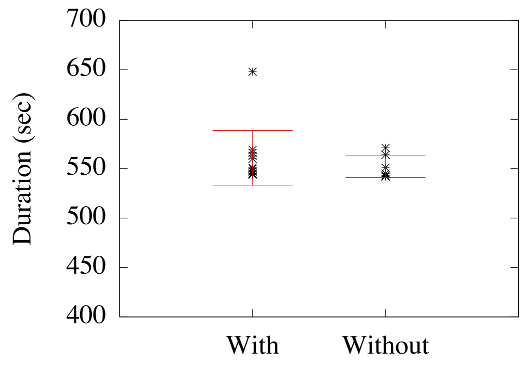


Figure 4: Hadoop job execution times, with and without Chukwa monitoring

when monitored by Chukwa. We used a 20-node Hadoop cluster, and ran a series of random-writer and word-count jobs, included with the standard Hadoop distribution. These jobs are commonly used as Hadoop benchmarks and their performance characteristics are well understood [32]. They first produced, then indexed, 50 GB of random text data. Each pair of jobs took roughly ten minutes to execute. Chukwa was configured to collect all Hadoop logs plus standard system metrics. This amounted to around 120 KB/min/node, and an average of 1296 adaptors per node.

Of this data, roughly two-thirds was task logs, and most of the rest was Hadoop framework logs. This is in accord with the internal Yahoo! measurements quoted in [7]. The IO performance of EC2 instances can vary by a few percent. We used the same instances throughout to control for this. The first job, run with Chukwa, was noticeably slow, presumably due to EC2 disk effects. All subsequent sequences of runs appear indistinguishable. Statistically, our results are consistent with Chukwa imposing no overhead. They effectively rule out the possibility of Chukwa imposing more than a 3% penalty on median job completion time.

4.2 Fan-in

Our next round of experiments was designed to verify that Chukwa collectors could handle the data rates and degree of fan-in expected operationally. Recall that our goal was to use no more than 5% of a cluster's resources for monitoring. Hence, designating 0.5% of machines as Chukwa collector and storage nodes is reasonable. This works out to a 200-to-1 fan-in.

We measured the maximum data rate that a single collector could handle with this degree of fan-in by conducting a series of trials, each using a single collector and 200 agents. In each run, the collector was configured to write data to a five-node HDFS cluster. After 20 minutes, we stopped the agents, and examined the received data.

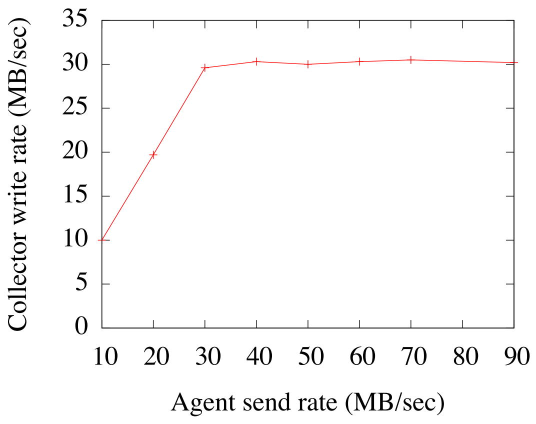


Figure 5: Throughput as a function of configured maximum send rate, showing that Chukwa can saturate the underlying filesystem. Fan-in of 200-1.

As can be seen in Figure 5, a single collector is able to handle nearly 30 MB/sec of incoming data, at a fan-in of 200-to-1. However, as the data rate per agent rises above that point, collector throughput plateaus. The Hadoop filesystem will attempt to write one copy locally, meaning that in our experimental setup, Collector throughput is limited by the sequential-write performance of the underlying disk. From past experiments, we know that 30 MB/sec is a typical maximum write rate for HDFS instances on EC2 in our configuration. Chukwa achieves nearly the maximum possible data rates on our configuration. We checked for lost, duplicate, and corrupted chunks — none were observed.

4.3 Scale

Hadoop and its HDFS file system are robust, mature projects. Hadoop is routinely used on clusters with thousands of nodes at Yahoo! and elsewhere. HDFS performs well even with more than a thousand concurrent writers, e.g. in the Reduce phase of a large distributed sort. [19]. In this section, we show that Chukwa is able to take advantage of these scaling properties. To do this, we started Hadoop clusters with a range of sizes, and a Chukwa collector on each Hadoop worker node. We then started a large number of agents, enough to drive these collectors to saturation, and measured the resulting performance. The collectors and HDFS DataNodes (workers) were hosted on "medium CPU-heavy" instances. The agent processes ran on "small" instances.

Rather than collect artificial logs, we used the output from a special adaptor emitting pseudorandom data at a controlled rate. This adaptor chooses a host-specific

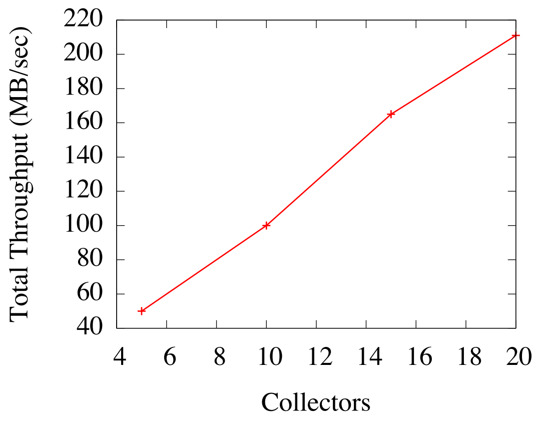


Figure 6: Aggregate cluster data collection rate, showing linear scaling.

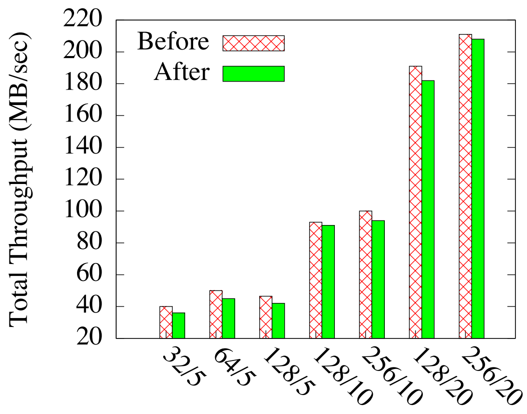


Figure 7: Performance before and after killing two collectors, showing modest degradation of throughput. Labels represent numbers of agents/collectors.

pseudorandom seed, and stores it in each chunk. This allows convenient verification that the data received came from the expected stream and at the expected offset in the stream.

Our results are displayed in Figure 6. Aggregate write bandwidth scales linearly with the number of DataNodes, and is roughly 10 MB/sec per node — a very substantial volume of log data. This data rate is consistent with our other experiences using Hadoop on EC2. In this experiment, the Chukwa collection cluster was largely IO-bound. Hosts had quite low CPU load and spent most of their time in the `iowait` state, blocked pending disk I/O. Chukwa is saturating the filesystem, supporting our assertion above that collector processes will seldom be the bottleneck in a Chukwa deployment.

Recall that our original goal was for Chukwa to consume less than 5% of a cluster's resources. The experiments presented here demonstrate that we have met this goal. Assume that monitoring imposes a 3% slowdown on each host. That would leave 2% of the cluster's resources for dedicated collection nodes. Given a thousand-node cluster, this would mean 20 dedicated Chukwa collectors and a 50-to-1 fan-in. Given the data rates observed in [7], each collector would only be responsible for 130 KB/sec; slightly over 1% of our measured collection capacity on a 20-node HDFS cluster. We conclude that, given 5% of a cluster's resources, Chukwa is able to easily keep up with real-world datacenter logging workloads.

4.4 Failure Tolerance

Fault-tolerance is a key goal for Chukwa. We ran a series of experiments to demonstrate that Chukwa is able to tolerate collector failures without data loss or substantial performance penalty. The configurations in this experiment were the same as described above, with a Chukwa collector on every HDFS node.

We began by testing Chukwa's response to the permanent failure of a subset of collectors. Our procedure was as follows: After running a test cluster for 10 minutes, we killed two collectors, and then let Chukwa run for another 10 minutes. We then stopped the agents and analyzed the results. We repeated this experiment with a variety of cluster sizes. In each case, all data had been received correctly, without missing or corrupted data. Figure 7 plots performance before and after stopping the two collectors. Having fewer collectors than Datanodes degraded performance slightly, by reducing the fraction of writes that were local to the collector.

We also tested Chukwa's response to a transient failure of all collectors. This models what would happen if the underlying filesystem became unavailable, for instance if the HDFS Namenode crashed. (The HDFS Namenode is a single point of failure that sometimes crashes, resulting in the filesystem being unavailable for a period from minutes to hours.) We began our experiment with 128 agents and 10 collectors running. After five minutes, we turned off the collectors. Five minutes later, we turned them on again. We repeated this process two more times.

We plot data received over time in Figure 8. As can be seen, data transfer resumes automatically once collectors are restarted. No data was lost during the experiment. The data rate quickly jumps to 100 MB/sec, which is consistent with the maximum rates measured above for clusters of this size.

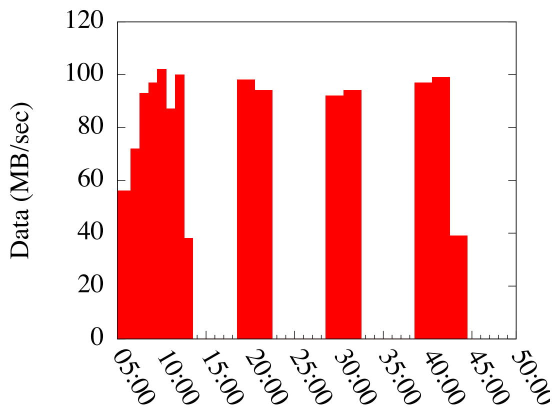


Figure 8: Data rate over time with intermittent collectors. Data transfer resumes automatically whenever collectors are available.

5 Case Studies

In this section, we discuss operational experiences using Chukwa in several contexts: web log analysis at several technology companies and real-time Cloud monitoring at Berkeley. We show that Chukwa is a natural solution for these disparate problems.

5.1 Web Log analysis

Web access logs are a particularly important class of logging data. These logs are typically line oriented, with one line per HTTP request. Automatically analyzing these logs is a core technical underpinning for web content and advertising companies. This makes analysis a good fit for Chukwa: the data volumes are large and short-turnaround automated analysis is important. We describe the experiences of two different companies: CBS Interactive and Specific Media.

CBS Interactive manages a wide range of online content, including the CBS News web site. Short-turnaround analysis allows the news room staff to monitor the popularity of stories from minute to minute, helping them gauge reader interest in particular topics. It also allows them to track the source of referrals to each story. Chukwa is a key piece of infrastructure for the underlying analysis. Content is served from a cluster of app servers, each of which writes its logs to local disk. Chukwa then copies this data into a small Hadoop cluster, where a series of Pig jobs aggregate this data and store it into a MySQL database. This database, in turn, is used by an internal web application to render data for users. Chukwa has been in use for several months and is functioning smoothly. The total volume of data is several gigabytes per day. A single collector is able to keep up with this load.

Specific Media is a leading online advertising vendor, responsible for placing advertisements on affiliate sites. Short-turnaround analytics are essential to make sure that ads are placed on sites with high likelihood of click-throughs from the advertiser's target demographic. The click-through data totals over a terabyte per day, before compression.

Chukwa is a natural fit for these uses. The data rates and past data volumes are high enough that distributed computing is necessary. Hadoop, with its easy scale-out, is a natural choice. However, blocking the production websites because of a monitoring or analysis system failure is unacceptable. As a result, the loosely-coupled Chukwa log tailing strategy is a good approach.

Both of these deployments made modifications to Chukwa to cope with site-specific needs. (These changes have been contributed back to the project.) Developers found it convenient to use the Chukwa agent process to manage parts of their logging workflows; notably CBS Interactive contributed the ability to trigger an HTTP post after every demux run in order to trigger further downstream processing.

5.2 Near-real-time Adaptive Provisioning

Chukwa was originally targeted at system analysis and debugging. But it can also be used for applications requiring lower latency in data delivery. One such application is adaptively provisioning distributed systems based on measured workload. SCADS, the Scalable Consistency-Adjustable Data Store, is an ongoing research project aiming to develop a low-latency data store with performance-safe queries [5]. A key component of SCADS is the "Director," a centralized controller responsible for making data placement decisions and for starting and stopping storage nodes in response to workload. Internally, SCADS uses X-Trace reports [11] as its data format. The total data volume varies from 60 to 90 KB/sec of data per node.

The SCADS development team opted to use local UDP to send the reports to Chukwa. Using TCP would have meant that SCADS might block if the Chukwa process fell behind and the kernel buffer filled up. Using the filesystem would have imposed unnecessary disk overhead. Each X-Trace report fits into a single UDP packet and in turn is sent through Chukwa as a single Chunk. This means that the Director will always see complete reports. The price for using UDP is that some kernels will discard local UDP messages under load. Some data loss is acceptable in this context, since the Director merely requires a representative sample, rather than every report.

Rather than wait for data to be visible in HDFS, the Director receives updates via fast path delivery. On boot, the Director connects to each collector, and requests copies of all reports. Once received, the reports are used to detect which hosts were involved in each read and write operation, and how long each host took to respond. Using this information, the Director is able to split up the data stored on an overloaded host, or consolidate the data stored on several idle ones. Data is typically delivered to the Director within a few seconds of being generated.

Using Chukwa in this scenario had a significant advantages over a custom-built system. While seeing data immediately is crucial to the Director, having a durable record for later analysis (potentially with MapReduce) is very helpful in tuning and debugging. Chukwa supports both, and can guarantee that all data that appeared once will eventually be stored. Using Chukwa also meant that the code for receiving data locally could be shared between this application and others.

5.3 Machine learning on logs

As mentioned in the introduction, one of our key goals was to enable various log analysis techniques that cannot gracefully tolerate lost data. We give an example of one such technique here. This illustrates the sort of automated log analysis Chukwa was intended to facilitate and shows why unreliable delivery of logs can poison the analysis.

Xu *et al.* have developed a machine learning approach able to detect many subtle error conditions by inspecting logs [30]. In a nutshell, their technique works as follows. Categorize the messages in a log and group them together based on whether they have a shared identifier (an ID number for an object in the system, such as a task ID.) Compare the number of messages of each type mentioning each identifier. For instance, on the Hadoop filesystem, a fixed number of "writing replica" statements should appear for each block. Seeing an unexpected or unusual number of events is a symptom of trouble.

Imperfect log statements can easily throw off the analysis. There is no easy way to differentiate a message dropped by the collection system event report from a message that was never sent because of an application bug. To conduct their experiments, Xu *et al.* copied logs to a central point at the conclusion of each experiment using `scp`. This would be unsuitable in production; logs grow continuously and the technique requires a consistent snapshot to work correctly. As a result, the largest test results reported for that work were using 200 nodes running for 48 hours. Experimental runs needed to be aborted whenever nodes failed in mid-run. There was no easy way to compensate for lost data.

Copying data off-node quickly, and storing it durably, would significantly enhance the scalability of the approach. Chukwa does precisely this, and therefore integrating this machine-learning approach with Chukwa was of practical importance. (This integration took place after the experiments described above had already been concluded.)

Adapting this job to interoperate with Chukwa was straightforward. Much of the processing in this scheme is done with a MapReduce job. We needed to add only one component to Chukwa — a custom MapReduce "input format" to hide Chukwa metadata from a MapReduce job and give the job only the contents of the collected chunks of log data. Aside from comments and boilerplate, this input format took about 30 lines of Java code. The analysis job required only a one-line change to use this modified input format.

6 Related Work

The Unix `syslogd` deamon, developed in the 1980s, supported cross-network logging [15]. Robustness and fault-tolerance were not design goals. The original specification for `syslogd` called for data to be sent via UDP and made no provision for reliable transmission. Today, `syslogd` still lacks support for failure recovery, for throttling its resource consumption, or for recording metadata. Messages are limited to one kilobyte, inconveniently small for structured data.

Splunk [25] is a commercial system for log collection, indexing and analysis. It relies on a centralized collection and storage architecture. It does not attempt high availability, or reliable delivery of log data. However, it does illustrate the demand in industry for sophisticated log analysis.

To satisfy this need, many large Internet companies have built sophisticated tools for large-scale monitoring and analysis. Log analysis was one of the original motivating uses of MapReduce [10]. Sawzall is a scripting language, designed for log analysis-type tasks, that simplifies writing big-data queries and that uses MapReduce as its execution engine [21]. A query language is only useful if there is data to query. While the MapReduce and Sawzall query tools have been described in the open literature, the details of log collection and management management in enterprise contexts are often shrouded in secrecy. For instance, little has been published about Google's "System Health infrastructure" tools, beyond mentioning their existence [22]. Chukwa is more comparable to these data sources, rather than to the query languages used to process collected data.

In the introduction, we mentioned a number of specialized log collection systems. Of these, Scribe is the best documented and has been used at the largest scale. Scribe is a service for forwarding and storing monitoring data. The Scribe metadata model is much simpler than that of Chukwa: messages are key-value pairs, with both key and value being arbitrary byte fields. This has the advantage of flexibility. It has the disadvantage of requiring any organization using Scribe to develop its own metadata standard, making it harder to share code between organizations.

A Scribe deployment consists of one or more Scribe servers arranged in a directed acyclic graph with a policy at each node specifying whether to forward or store incoming messages. In contrast to Chukwa, Scribe is not designed to interoperate with legacy applications. The system being monitored must send its messages to Scribe via the Thrift RPC service. This has the advantage of avoiding a local disk write in the common case where messages are delivered without error. It has the disadvantage of requiring auxiliary processes to collect data from any source that hasn't been adapted to use Scribe. Collecting log files from a non-Scribe-aware service would require using an auxiliary process to tail them. In contrast, Chukwa handles this case smoothly.

As mentioned above, Scribe makes significantly weaker delivery guarantees than Chukwa. Once data has been handed to a Scribe server, that server has responsibility for the data. Any durable buffering for later delivery is the responsibility of the server, meaning that the failure of a Scribe server can cause data loss. There can be no end-to-end delivery guarantees, since the original sender does not retain a copy. Clients can be configured to try multiple servers before giving up, but if a client cannot find a working Scribe server, data will be lost.

Another related system is Artemis, developed at Microsoft Research to help debug large Dryad clusters [9]. Artemis is designed purely for a debugging context: it processes logs *in situ* on the machines where they are produced, using DryadLINQ [31] as its processing engine. The advantage of this architecture is that it avoids redundant copying of data across the network, and enables machine resources to be reused between the system being analyzed and the analysis. The disadvantage is that queries can give the wrong answer if a node crashes or becomes temporarily unavailable. Artemis was not designed to use long-term durable storage, which requires replication off-node. Analysis on-node is also a poor fit for monitoring production services. Analyzing data where it is produced risks having data analysis jobs interfere with the system being monitored. Chukwa and Scribe, in contrast are both designed to monitor production services and were designed to decouple analysis from collection.

Chukwa is flexible enough to emulate Artemis if desired, in situations with large data volumes per node. Instead of writing across a network, agents could write to a local Hadoop filesystem process, with replication disabled. Hadoop could still be used for processing, although having only a single copy of each data item reduces the efficiency of the task scheduler [20].

Flume is another, more recent system developed for getting data into HDFS [2]. Flume was developed after Chukwa, and has many similarities: both have the same overall structure, and both do agent-side replay on error. There are some notable differences as well. In Flume, there is a central list of ongoing data flows, stored redundantly in Zookeeper. Whereas Chukwa does this end-to-end, Flume adopts a more hop-by-hop model. In Chukwa, agents on each machine are responsible for deciding what to send.

There are also a number of more specialized monitoring systems worth mentioning. Tools like Astrolabe, Pier, and Ganglia [27, 14, 17] are designed to help users query distributed system monitoring data. In each case, an agent on each machine being monitored stores a certain amount of data and participates in answering queries. They are not designed to collect and store large volumes of semi-structured log data, nor do they support a general-purpose programming model. Instead, a particular data aggregation strategy is built into the system.. This helps achieve scalability, at the cost of a certain amount of generality. In contrast, Chukwa separates the analysis from the collection, so that each part of a deployment can be scaled out independently.

7 Conclusions

There is widespread interest in using Hadoop to store and process log files, as witnessed by the fact that several systems have been built to do this. Chukwa improves on these systems in several ways. Rather than having each part of the monitoring system be responsible for resuming correctly after a failure, we have demonstrated an end-to-end approach, minimizing the amount of state that needs to be stored in the monitoring system. In recovering from failures, Chukwa takes advantage of local copies of log files, on the machines where they are generated. This effectively pushes the responsibility for maintaining data out of the monitoring system, and into the local filesystem on each machine. This file-centered approach also aids integration with legacy systems. Chukwa also offers the flexibility to support other data sources, such as `syslog` or local IPC.

Chukwa is efficient and practical. It was designed to be suitable for production environments, with particular attention to the cloud. Chukwa has been used successfully in a range of operational scenarios. It can scale to large data volumes and imposes only a small overhead on the system being monitored.

We have shown that Chukwa scales linearly up to 200 MB/sec. If sufficient hardware were available, Chukwa could almost certainly match or exceed the highest reported cluster-wide logging rate in the literature, 277 MB/sec. [9]. While few of today's clusters produce remotely this much data, we expect that the volume of collected monitoring data will rise over time. A major theme in computer science research for the last decade has been the pursuit of ever-larger data sets and of analysis techniques to exploit them effectively [13]. We expect this to hold true for system monitoring: given a scalable log collection infrastructure, researchers will find more things worth logging, and better ways of using those logs. For instance, we expect tracing tools like XTrace and DTrace to become more common [11, 8]. Chukwa shows how to build the necessary infrastructure to achieve this at large scale.

Availability

Chukwa is a subproject of Hadoop and is overseen by the Apache Software Foundation. All code is available under permissive license terms. At present, the Chukwa website is `http://hadoop.apache.org/chukwa`; releases can be obtained from there.

Acknowledgments

We thank Bill Graham and Gerrit van Vuuren for describing how Chukwa is being used at CBS Interactive and Specific Media, respectively. This paper has focused on the data collection and transport side of Chukwa. Eric Yang and Jerome Boulon were responsible for the processing and display components, which are also important aspects of the overall system. The students and faculty of the RAD Lab at UC Berkeley supplied a great deal of feedback and advice.

This research is supported in part by gifts from Sun Microsystems, Google, Microsoft, Amazon Web Services, Cisco Systems, Cloudera, eBay, Facebook, Fujitsu, Hewlett-Packard, Intel, Network Appliance, SAP, VMWare and Yahoo! and by matching funds from the State of California's MICRO program (grants 06-152, 07-010, 06-148, 07-012, 06-146, 07-009, 06-147, 07-013, 06-149, 06-150, and 07-008), the National Science Foundation (grant CNS-0509559), and the University of California Industry/University Cooperative Research Program (UC Discovery) grant COM07-10240.

References

[1] Scribe. http://sourceforge.net/ projects/ scribeserver/, 2008.

[2] Cloudera's flume. `http://github.com/cloudera/flume`, June 2010.

[3] M. Aharon, G. Barash, I. Cohen, and E. Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, Bled, Slovenia, September 2009.

[4] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, et al. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report 2009-28, UC Berkeley, 2009.

[5] M. Armbrust, A. Fox, D. A. Patterson, N. Lanham, B. Trushkowsky, J. Trutna, and H. Oh. SCADS: Scale-Independent Storage for Social Computing Applications. In *Fourth Conference on Innovative Data Systems Research*, Asilomar, CA, January 2009.

[6] D. Borthakur. HDFS Architecture. http://hadoop. apache. org/common/ docs/r0.20.0/ hdfs_design.html, April 2009.

[7] J. Boulon, A. Konwinski, R. Qi, A. Rabkin, E. Yang, and M. Yang. Chukwa, a large-scale monitoring system. In *First Workshop on Cloud Computing and its Applications (CCA '08)*, Chicago, IL, 2008.

[8] B. Cantrill. Hidden in Plain Sight. *ACM Queue*, 4(1), 2006.

[9] G. F. Creţu-Ciocârlie, M. Budiu, and M. Goldszmidt. Hunting for problems with Artemis. In *First USENIX Workshop on Analysis of System Logs (WASL '08)*, San Diego, CA, December 2008.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, Volume 51(Issue 1):107–113, 2008.

[11] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. XTrace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, Cambridge, MA, April 2007.

[12] S. Ghemawat, H. Gobioff, and S. Leung. The Google file system. In *19th Symposium on Operating Systems Principles (SOSP)*, 2003.

[13] A. Halevy, P. Norvig, and F. Pereira. The Unreasonable Effectiveness of Data. *IEEE Intelligent Systems*, 24:8–12, 2009.

[14] R. Huebsch, J. Hellerstein, N. Lanham, B. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. *Proceedings of 19th International Conference on Very Large Databases (VLDB)*, pages 321–332, 2003.

[15] C. Lonvick. RFC 3164: The BSD syslog Protocol. http://www.ietf.org/rfc/rfc3164.txt, August 2001.

[16] A. Makanju, A. N. Zincir-Heywood, and E. E. Milios. Clustering event logs using iterative partitioning. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

[17] M. Massie, B. Chun, and D. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7):817–840, 2004.

[18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1099–1110. ACM New York, NY, USA, 2008.

[19] O. O'Malley and A. C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. http://sortbenchmark.org/ Yahoo2009.pdf, April 2009.

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A Comparison of Approaches to Large-Scale Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, Providence, RI, 2009.

[21] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming Journal*, Volume 13(Number 4/2005):277–298, 2003.

[22] E. Pinheiro, W. Weber, and L. Barroso. Failure Trends in a Large Disk Drive Population. In *5th USENIX Conference on File and Storage Technologies (FAST '07)*, San Jose, CA, 2007.

[23] Rapleaf, inc. The collector. http://blog.rapleaf .com/dev/?p=34, October 2008.

[24] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Cloud Computing and Applications*, 2008.

[25] Splunk Inc. IT Search for Log Management, Operations, Security and Compliance. http://www.splunk.com/, 2009.

[26] J. Stearley. Towards informatic analysis of syslogs. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[27] R. Van Renesse, K. Birman, and W. Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM TOCS*, 21(2):164–206, 2003.

[28] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th Symposium on Operating Systems Principles (SOSP)*, 2001.

[29] T. White. *Hadoop: The Definitive Guide*, pages 439–447. O'Reilly, Sebastopol, CA, 2009.

[30] W. Xu, L. Huang, M. Jordan, D. Patterson, and A. Fox. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[31] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

[32] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, CA, December 2008.

How to Tame Your VMs: an Automated Control System for Virtualized Services

Akkarit Sangpetch asangpet@andrew.cmu.edu
Andrew Turner andrewtu@cmu.edu
Hyong Kim kim@ece.cmu.edu
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA

Abstract

Modern datacenters contain a large number of virtualized applications and services with constantly changing demands for computing resources. Today's virtualization management tools allow administrators to monitor current resource utilization of virtual machines. However, it is quite challenging to manually translate user-oriented service level objectives (SLOs), such as response time or throughput, to suitable resource allocation levels. We presented an adaptive control system which automates the task of tuning resource allocations and maintains service level objectives. Our system focuses on maintaining the expected response time for multi-tier web applications. Our control system is capable of adjusting resource allocation for each VM so that the applications' response time matches the SLOs. Our approach uses individual tier's response time to model the end-to-end performance of the system. The system helps stabilize applications' response time. It can reduce the mean deviation of the response time from specified targets by up to 80%. Our system also allows the physical servers to double the number of VMs hosted while maintaining the target response time.

Tags: VMs, research, control, resource, allocation

1. Introduction

Modern datacenters contain a large number of virtualized applications and services; with constantly changing demands for computing resources. These virtual workloads are executed on multiple virtual machines (VMs) which can be consolidated onto a smaller number of physical hosts. Today's virtualization management tools allow administrators to monitor current resource utilization of virtual machines. Management capabilities such as adjustable resource allocation [9] are also provided as a way to configure the underlying resource to meet applications' demands.

However, it is quite challenging to manually translate user-oriented service level objectives (SLOs), such as response time or throughput, to suitable resource allocation levels. Such tasks demand experience administrators and significant amount of time. Moreover, virtualized applications are often distributed and dependent on each other. It is imperative that the administrators understand the complex behaviors of the applications before they are able to manually tune them effectively.

In this work, we developed an adaptive control system which automates the task of tuning resource allocations and maintains service level objectives. Our system initially focuses on the expected response time for multi-tier web applications as our primary SLOs. Our control system is capable of adjusting CPU share allocation for each VM so that the applications' response time matches the SLOs. Our approach uses individual tier's response time to model the end-to-end performance of the system. This allows our model to capture systems' dynamics without relying on just their resource utilization level.

Our system helps stabilize applications' response time. It can reduce the mean deviation of the response time from specified targets by up to 80%. The system also allocates only the required amount of resource to satisfy the SLOs for each VM. Without over-provisioning, our system can increase the number of hosted applications by up to 100%.

The capabilities provided by our system are useful for administrators. It provides a way to express levels of services in terms of actual application performance. Our system can be applied to a cloud-based service provider model as well as in smaller clusters where resources are limited and applications may have different priorities. Our controllers can allocate just enough resource to satisfy the level of service required, allowing individual host to process more workloads.

We deployed our system on Linux and Kernel Virtual Machine environment in a local cluster. Our results suggest that the system can maintain the service response time for different VMs running on the host. Our system can also adapt to the level of workload changes and adjust system parameters in order to match the service response time.

We will explain the overall design of our system in section 2. The detailed specification on each component could be found in section 3. We evaluate our system's performance in section 4. The related works are reviewed in section 5. The discussion and ongoing works are explained in section 6.

2. System Overview

Our control system consists of four components; sensor, actuator, modeler, and controller as shown in Figure 1. Our design resembles a closed-loop control system. During each control interval, the sensors collect application-related performance (such as response time) from VMs hosting the controlled application. The collected information is fed to the modeler and is used to update the application's performance model. The modeler creates a performance model for targeted applications by adjusting the model parameters based on sensor inputs. The model obtained can be later used to predict the applications' performance for possible system configurations. The controller then uses the model to find the optimized system configurations and send the result to actuators. The actuators then adjust the system parameters accordingly. The impact on the applications' performance can be measured during subsequent intervals by the sensors, forming a closed-loop control system.

We currently use Linux and KVM as our hypervisor. However, the system can be extended to support other environments.

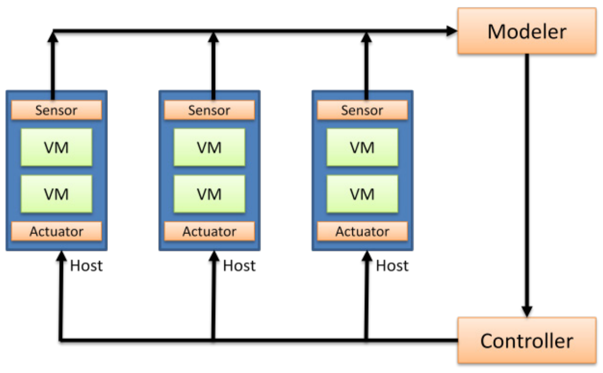


Figure 1: Control System Overview

Our initial system design primarily focuses on the response time as our controlled objective. The system tries to control the CPU share allocation for each VM in order to match the specified response time objectives. The system can automate the task of finding suitable CPU allocation for each VM tier. By controlling the number of shares allowed for each VM, we are able to increase the number of VMs running on a host without impacting the response time of the controlled applications. This allows the overall cluster to be more efficient and able to accept more workloads while maintaining existing SLOs.

The detailed description of our system is discussed in the next section.

3. System components

The components of our system could be described as followed.

3.1 Sensors

Sensors utilize packet filtering and capturing tools to analyze packets intended for the controlled VM. Our sensors can extract response time from the target applications' components. The response time is the time from the moment the last packet of the request is sent to the moment the last packet of the response arrives.

We collect the application performance metrics from different application tiers. For our initial system design, our sensor try to determine the application performance based on network packets going through the VMs. We are currently using the response time collected from each application tier. However, the sensor can also be used to collect other performance-oriented metrics such as the application's throughput, or number of concurrent requests.

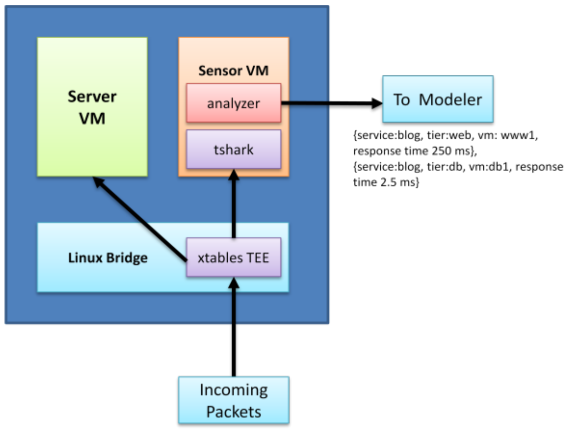

Server VM
Sensor VM
analyzer
tshark
To Modeler
{service:blog, tier:web, vm: www1, response time 250 ms},
{service:blog, tier:db, vm:db1, response time 2.5 ms}
Linux Bridge
xtables TEE
Incoming Packets

Figure 2: Sensor implementation

Our current implementation of the sensor is a combination of packet filtering and capturing tools which capture packets intended for the concerned server (as shown in Figure 2). The sensor is a guest VM running on the controlled hosts. This allows us to deploy and modify the sensor without too much modification on the physical host. The sensor utilizes *tshark* (packet analyzer) and *pcap* (packet capture driver) to extract the response time of the controlled applications.

The applications' response time is determined by recording the timestamp of packets (belonging to the same connection) with matching request parameters on the specified port number. Currently, the administrators have to supply URIs' pattern for HTTP requests/responses, or MySQL command for database queries as the request parameters.

Since all VMs in the host share a single virtual network bridge, we can filter only packets destined to controlled VMs (with *iptables*) and forward copies of the packets (with *xtables-TEE* target) to the sensor VM. This reduces the overall number of packets that our sensor has to process and analyze. We also avoid placing pcap driver directly on the host because it can only capture packets that actually pass through the machine's network interface. By placing pcap driver in the guest VM, we can intercept packets from dependent VMs communicating within the same host.

Our sensors periodically generate a response time summary for each VM. The summary consists of the name of service being monitored, its application tier, VM server, and its response time.

As the sensor is located on the host, the response time is measured starting at the moment when a packet has arrived on the host and stopping when a response packet has been observed by the sensor. In our test environment, the network propagation time is negligible since all hosts are located on the same local area network.

3.2 Actuators

Actuators are small agents installed on the host. They adjust the hypervisor parameter as specified by the controller. Currently our actuators can control the number of CPU shares allocated for VMs on physical hosts. It is possible to extend the actuator to control other system parameters.

In our test environment we adjust the scheduler level of CPU share for each VM using Linux Control Groups subsystem (cgroups.) Cgroups allows us to set the CPU share for each process running on the host. By default, KVM utilizes the Linux kernel's Completely Fair Scheduler (CFS.) The scheduler's behavior is configurable via cgroups cpu share (*cpu.shares*).

Cgroups allows us to set the CPU share for each process running on the host. We use the default Linux Completely Fair Scheduler (CFS) configurable via cgroups CPU share (*cpu.shares*). In the CFS scheduler, each process (or a virtual CPU) is given 1024 shares, unless configured otherwise. The portion of time scheduled for the process is calculated as a ratio between the number of the shares given to the process and the sum of all shares given to every runnable process (on the same physical CPU).

Moreover, the CFS scheduler exhibits work-conserving behavior. This means that if a process happens to be the only one running on a CPU, it gets all available CPU time regardless of the number of shares allocated. Such behavior also indicates that the share configured for CFS does not constitute CPU limits for the process.

In a system with multiple CPUs, the scheduler also utilizes a load-balancer which tries to balance the amount of workload equally amongst each CPU. However, the load-balancer can move a virtual CPU of a VM after it is assigned a preferred number of shares. Such behavior can lead to inaccurate measure between the number of share allocated and the observed applications' performance. In order to effectively control the scheduling parameter, we also have to pin CPUs of all controlled VMs on the same physical core. This makes the relationship between the number of share allocated and the measured response time to be more stable. Our actuators then only have to set the share to match a number specified by the controller. It is the controller's task to find the best possible share for the current workload.

3.3 Modeler

The modeler creates a performance model for controlled applications by resolving its internal parameters based on sensor inputs. The obtained model can later be used to predict the application's performance for specified system configurations.

Our modeler updates a prediction model for the application performance based on the sensor inputs. The model is based on observations between the measured response times from different application tiers. The model uses control signals (CPU share) and measured input (individual tier response time from the sensors.) Although building an accurate model may be a time-consuming process and could be applied only to a specific application, we found that an intuitive model based on application tier relationship could be used to derive a practical performance prediction model.

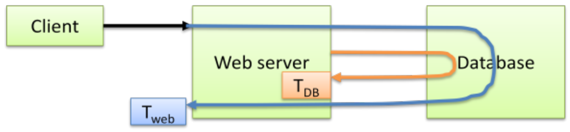


Figure 3: Two-tier web application model

Consider a generic two-tier web application shown in Figure 3; we could build an empirical model for the end-to-end system response time as a linear combination of the time spent in the database and web tier. When a client requests a (dynamic) web page, the web server will make additional requests to the database. The web server then processes the responses before returning the value to the client. Assuming that our concerned requests exhibits similar behavior, the relationship between the web response time (T_{web}) and the database response time (T_{web}) could be represented by $T_{web} = A \cdot T_{DB} + B$.

For example, in Figure 4, the response time used to access a Wordpress home page (a popular blogging web application) exhibits a linear relationship with its database server response time. When the time takes to process database requests increases (due to additional load from another VMs residing on the same host with the database server), the overall web response time also increases.

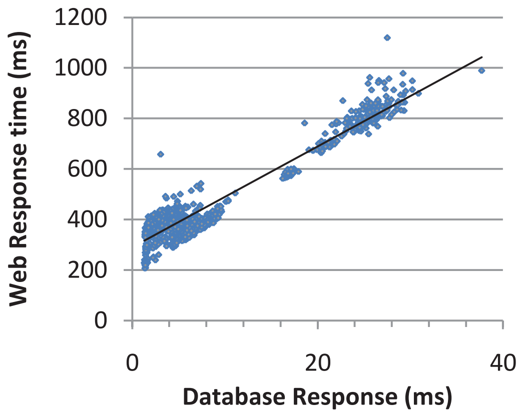


Figure 4: Linear relationship between web and database server response time

We can use this linear model to predict possible performance values for the next sensor interval. Given previous measurement values for the web (T'_{web}) and database response time(T'_{DB}), we can represent the current measurement from our sensor as $T_{web} = T'_{web} + A \cdot (T_{DB} - T'_{DB})$. By performing an ordinary least-square regression on multiple data points (obtained from the sensors), we can estimate the common coefficient A and use the same equation to predict the web response time for the next sensor interval.

If we want to be able to adjust the number of shares for the database VM, we also have to find a relationship between the database server CPU share allocation (S_{CPU}) and the database response time(T_{DB}). On a physical host with very high CPU utilization, which represents a worst-case scenario for consolidation, we found that the relationship between the database response time and its CPU share could be represented by a power law curve ($T_{DB} = a \cdot S_{CPU}^{b}$). Figure 5 shows the observed relationship between the response time of the database server and the number of CPU shares allocated for the database VM. We can also obtain the relationship coefficient by fitting a least square on the log-scale of T_{DB} and S_{CPU} , i.e. $\ln T_{DB} = a + b \ln S_{CPU}$

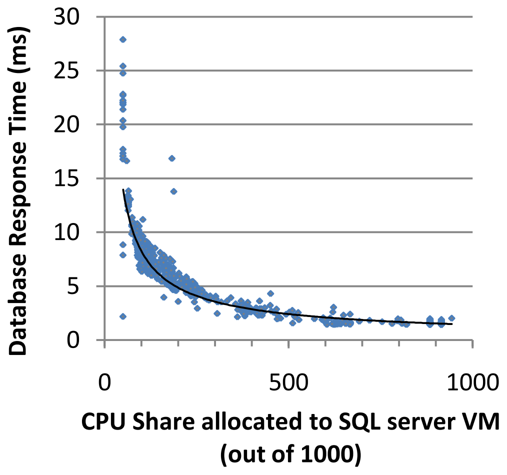


Figure 5: Power law relationship between the database server's response time and its allocated shares

Note that the model given in this section might initially seem to be very specific to our scenario. However, such scenarios are quite common in actual deployments. For example, the linear relationship can be directly applied to many existing web applications. The relationship between allocated CPU shares and the database server response time can also be used to approximate other scenarios where a controlled VM is placed on a very busy host.

Additionally, our model parameter can be obtained on-line by periodically updating the regression parameters with recent measurements. This also allows our system to dynamically adapt its model based on the current level of workloads. However, since our current model relies on many past sensor readings, its ability to adjust the models for sudden change of workload levels will be limited.

3.4 Controller

The controller is the final component which glues all the pieces of our system together. Our controller takes the updated model obtained by the modeler and sensor inputs from the current interval. It then tries to find the minimal virtual CPU allocation that yields the response time closest to the one defined in the SLOs.

Our controller also utilizes both long-term and short-term prediction. The long-term prediction uses the moving average value generated from previous sensor readings as the input for the model. The short-term prediction uses the most recent sensor reading as the model input. The controller primarily determines the number of shares based on the long-term prediction to maintain system stability. However, the short-term prediction is utilized when the sensors' reading shows SLO violations. This allows the system to avoid immediate SLO violations while still maintaining stability.

Once the control decision has been made, the controller forwards the result to the actuator which actually adjusts the system based on the control signal.

In more complex scenarios, we may have to optimize for a large number of potential parameters. We could view such scenario as a state-space search problem and additional heuristic will be needed. Alternatively, we are currently exploring methods used in classical control theory which could be applied to our linear models.

4. Evaluation

4.1 Experimental Setup

Although the framework of our system design is generic, we deployed a proof-of-concept system on an example system as shown below. The targeted application is a blogging web application (Wordpress) which consists of a web server (Apache and PHP) and a database server (MySQL).

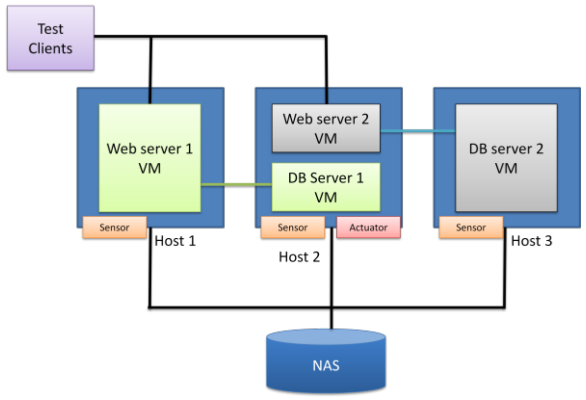


Figure 6: Test setup for evaluation

Our test system setup is shown in Figure 6. We run two instances of the described web application. Each instance may have different service level requirement. This represents differentiate levels of service demanded in actual infrastructure deployment and will be described in the evaluation.

We use Linux KVM as our hypervisor in the experiment. The deployed operating system on all systems is Fedora 12 with Linux 2.6.33 kernel. The host systems contain Intel Quad Core Q6600 with 4GB of memory. Each VM is allocated 1 GB memory with one virtual CPU. The VM image is storing on a dedicate NFS server on the local network. As our test workload fits in the system memory, the storage system does not cause a bottleneck in our test scenario. The web server is more CPU-intensive, compared to the database. Client loads are generated from other machines located on the same local area network. Each client is associated with a single web server VM. Every 1,000ms, the client generates a request to the home page on its associated web server.

We placed sensor on all participating hosts. For the purpose of evaluation, we only concern with the actuator on Host 2 where potential contention could occur. The actuator needs to arbitrate the amount of CPU shares allocated between the web and the database server for two different services with different service response time objectives. Both sensor and control interval are set to 5,000ms.

4.2 Evaluation Result

Our evaluations suggest that our control system can be used to maintain the service level objectives for the hosted applications. It could also react to change in workload level.

4.2.1 Multiple SLOs

Our control system allows multiple service level objectives to be achieved. In this experiment, we set the demand so that the expected end-to-end response time for the blogging web application instance 1 should be 800ms while the response time for instance 2 should be 4,000ms. Note that this represents differentiate level of services, and the objective is described in term of the end-user experience. We expect that our control system will try to adjust the CPU allocation on host 2 so that both SLOs could be met. As for the driving workload, instance 1 was serving 2 concurrent clients. Instance 2 was serving 15 concurrent clients.

The system response time and its target for each instances is shown in Figure 7 and 8. Figure 7 shows the result when we do not use our control system and each VM is allocated the default number of CPU shares. Figure 8 shows the result when we enable our control system. The absolute mean deviation from the target response time for each instance is shown in Table 1.

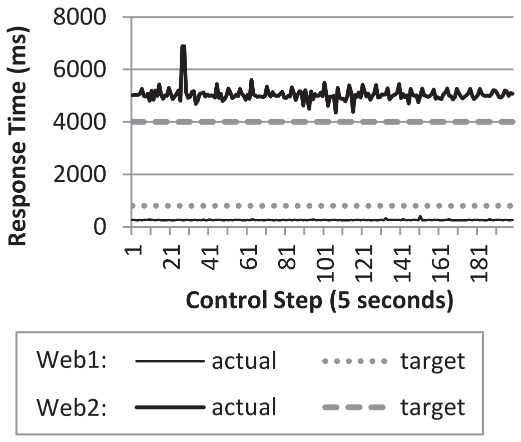


Figure 7: Response time without the control system (static workload)

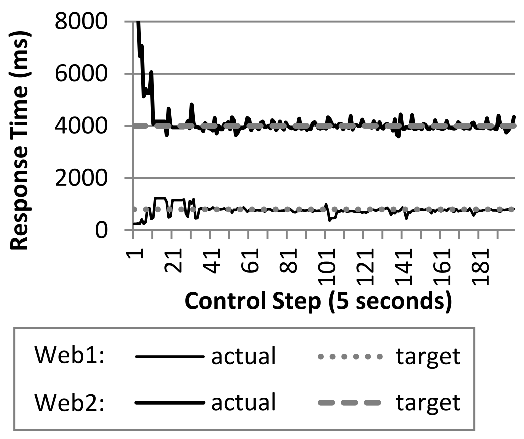


Figure 8: Response time with the control system (static workload)

Application Instances	Mean Deviation from SLOs	
	Without Control	With Control
Instance 1	540 ms	109 ms
Instance 2	1043 ms	282 ms

Table 1: Mean deviations for static workloads

Without the control system, the default number of shares allocated for the database VM instance 1 is too high. Therefore, its response time is much lower than the expected value. However, the response time for instance 2 is also much higher than the SLO specifies as too many resources are given to instance 1. Such system fails to meet the given service demands for instance 2.

With our control system, both instances can be satisfied as the controller adjusts the share to track the expected response time. As a result, both instances can operate within the demanded response time. The share allocated for the database VM for instance 1 is shown in Figure 9. Initially, the adjusted allocation will have high variance as it attempts to find the stable operating points.

Once the operating points have been found, it is also possible that the controller will react to unanticipated system event (such as disk paging, or periodic system maintenance tasks.) These events are indicated by occasional spikes in the response time and the share graph. However, the control system will finally try to revert back to its normal operating points in order to maintain the SLO. The current controller takes about one minute to readjust after such event occurs.

Our current implementation of the controller only attempts to match the actual and the expected response time. As a result, some of the requests may go over the given requested time. In actual deployment, it is possible to specify a lower expected response time than the wanted limits to account for the variances.

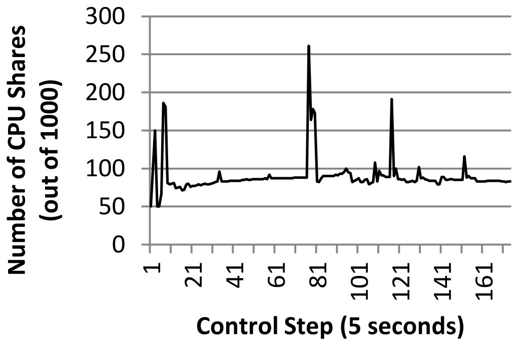


Figure 9: Number of CPU shares allocated by the controller for instance 1's database server

Note that it is also possible for system administrators to manually analyze the workload characteristics and preset the allocation accordingly. However, such task is time-consuming and the administrator may not be able to react as quickly to changes in workload or other system events.

4.2.2 Dynamic Workload

Another benefit of having the control system is it can adapt the allocation for dynamically changing workload. The result in this section shows the effectiveness of the control system while workload level changes for instance 1. The workload level for the instance is shown in Figure 12. The system setup is the same as in previous section. The differences are the number of concurrent clients for instance 1. Also, due to higher overall load, we set the response time required by instance 1 to 1,000ms. For non-controlled system, the number of CPU shares for DB server 1 has been set to satisfy the average load over the evaluation period.

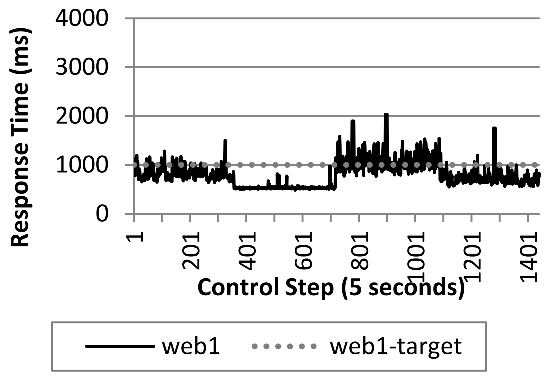


Figure 10: Response time without the control system (dynamic workload)

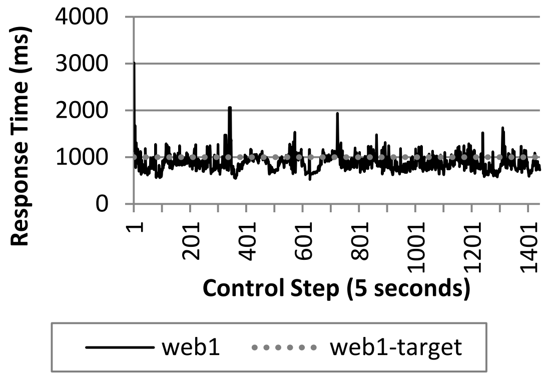


Figure 11: Response time with the control system (dynamic workload)

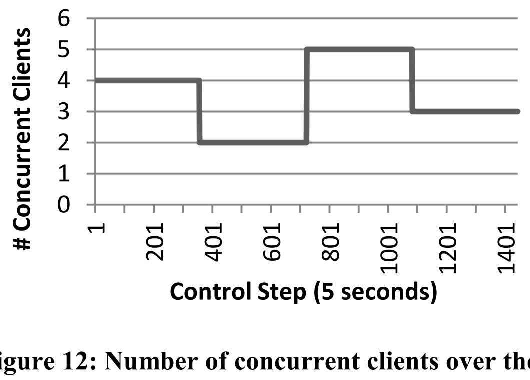


Figure 12: Number of concurrent clients over the test period

Application Instances	Mean Deviation from SLOs	
	Without Control	With Control
Instance 1	276 ms	182 ms

Table 2: Mean deviations for dynamic workloads

Figure 10 shows the response time of the web server when the control system is not enabled. Figure 11 shows the response time of the web server when the control system is enabled. The absolute mean deviation from the target response time for instance 1 is also shown in Table 2.

Without the control system, it is possible that, for particular level of workload, the SLO could be easily met because the workload level is well below the average. However, when the level of work load is higher than the average, the instance also fails to meet the SLOs provided.

With the control system, the response time tracks more closely to the expected response. However, the controller may not react as quickly as in the static workload cases. This behavior happens because our model relies on past sensor readings to build up the system models. After the workload level has been changed, the system has to readjust itself and settle to a new model. However, the system can still maintain the target workload level, although it shows higher degree of variances.

5. Related works

Existing commercial solutions remain focused on the resource utilization aspect of VMs, not the applications' performance. Current management tools are capable of reacting to high levels of resource utilization by performing live migrations [7] to reduce the hardware usage.

Existing tools could assist in capacity planning by profiling the hardware utilization level and forecasting future resource demands in datacenters [8]. However, such tools do not directly address the problem of managing the applications' performance.

Previous works have been done to investigate the behavioral model of multi-tier application using profiling-based methods [2] [6]. Such model only predicts long-term statistical behavior and is applicable for static workloads.

Control systems have been used in tuning computer applications' parameters [4] [5]. Researchers have applied control-theoretical approach to address VMs' resource allocation [1] [3]. Such system adjusts its model by observing only the clients' response time whereas our system also responds to performance changes in related application tiers. Feedback-controlled systems have also been investigated in order to improve the system utilization for CPU throughput-based applications [11]. Our work could compliment such method as we are focusing on achieving target response time for delay-based applications.

6. Discussions & Future Works

Our results suggest that it is possible to use a control system to maintain target SLOs on virtualized system and also able to react to changes in workload levels. With the control system, administrators will be able to deploy virtualized workload without concerning about low level system-configuration such as CPU shares.

6.1 Workload modeling

Actual enterprise applications could be much more complicated than the current model given in this paper. The linearity assumptions may not be held for complex chain of dependent VMs. We are interesting to explore possible composition models (such as Markov Chains) that can be used to approximate the response time performance of such distributed applications. It should be possible to derive a more accurate performance model for complex application based on a composition of simpler models such as those described in this paper.

Our initial model only captures direct dependency between application tier (e.g. a web server directly makes request to a database server.) We also investigate the performance behavior and its relation between particular types of behaviors. These include partitioned requests, load-balance, or aggregate behavior of the application tiers. Such model could give us more insight into the relationship between the performance and application's composition which allows us to generate a model for complex applications.

6.2 Control Parameters

In this paper, we have been only experimented with controlling the CPU share allocation. In actual system, more control parameters could be used to change the behavior of the VMs. For example, it is possible to associate traffic for different VMs with multiple network traffic classes. This allows the system to have more control over the queuing and priority for behavior of the VMs' network traffic. Similarly, for local storage control, it is possible to use I/O-controller [10] to control the share for disk I/O access.

7. Conclusions

We presented an automated control system for virtualized services. Our system suggests that it is possible to use intuitive models based on observable response time incurred by multiple application tiers as a model for the overall performance. The models are also used in conjunction with a control system to determine the optimal share allocation for the controlled VMs. Our system helps maintain the expected level of service response time while adjusting the allocation to meets the demand for different level of workloads. Such behavior allows administrator to simply specify the required end-to-end service-level response time for each application, without the need of constant monitoring or understanding complex behavior of the applications. Our system helps simplifying the task of managing the performance of many VMs already exists in today's datacenter.

8. References

[1] P. Padala, K.Hou, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. EuroSys 2009.

[2] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. Middleware 2008.

[3] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal. Optimal Multivariate Control for Differentiated Services on a Shared Hosting Platform. IEEE CDC 2007.

[4] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, A control theory foundation for self-managing computing systems, IEEE journal on selected areas in communications, Dec. 2005

[5] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server. Network Operations and Management Symposium, 2002.

[6] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource overbooking and application profiling in shared hosting platforms. OSDI 2002.

[7] VMware,Inc. VMware Infrastructure: Resource Management with VMware DRS. http://www.vmware.com/pdf/vmware_drs_wp.pdf

[8] VMware Inc. VMware vCenter CapacityIQ. http://www.vmware.com/products/vcenter-capacityiq/

[9] VMware, Inc. vSphere Resource Management Guide. http://www.vmware.com/pdf/vsphere4/r40/vsp_40_resource_mgmt.pdf

[10] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. FAST, February 2009.

[11] Nathuji, R., Kansal, A., and Ghaffarkhah, A. 2010. Q-clouds: managing performance interference effects for QoS-aware clouds. EuroSys, April 2010.

Empirical Virtual Machine Models for Performance Guarantees

Andrew Turner
andrewtu@cmu.edu

Akkarit Sangpetch
asangpet@andrew.cmu.edu

Hyong S. Kim
kim@ece.cmu.edu

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

Existing Virtual Machine (VM) management systems rely on host resource utilization metrics to allocate and schedule VMs. Many management systems only consolidate and migrate VMs based on hosts' CPU utilizations. However, the performance of delay-sensitive workloads, such as web services and online transaction processing, can be severely degraded by contention on numerous of the hosts' components. Current VM management systems typically use threshold based rules to decide when to migrate VMs, rather than using application-level performance. This means that they cannot easily provide application-level service level objective (SLO) guarantees. Providing SLO guarantees is even more difficult when considering that today's enterprise applications often consist of multiple VM tiers.

In this paper we show how the performance of a multi-tiered VM application can be empirically captured, modeled and scaled. This allows our management system to guarantee application-level performance, despite variable host utilization and VM workload levels. Additionally, it can predict the performance of an application at host utilization levels that have not been previously observed. This is achieved by performing regression analysis on the previously observed values and scaling the applications performance model. This allows the performance of a VM to be predicted before it is migrated to or from a new host. We have found that by dynamically, rather than statically, allocating resources, average response time can be improved by 30%. Additionally, we found that resource allocations can be reduced by 20%, with no degradation in response time.

1. INTRODUCTION

Modern data centers contain a large number of virtual machines (VMs). Additionally, internet Cloud services use VMs to run multiple applications across multiple physical servers, under the premise that the Cloud is a single resource pool. While hypervisor vendors such as VMware [1], Citrix [2] and Microsoft [3] tout the potential benefits of VMs, these benefits are not always fully realized. This is typically due to increased overheads and resource contention cause by other VMs. In this paper we show how application-level performance can be guaranteed for multi-tier VM applications. Additionally, we show how hardware utilization can be increased over current VM management systems by more densely packing VMs than threshold based systems. Finally, we show that the overall performance of the applications in a datacenter can be improved by dynamically setting resource allocation levels.

VMs were originally deployed as a way to increase resource utilization levels. This is achieved by consolidating multiple machines that have low resource utilization levels onto a single physical host, saving both hardware capital and energy costs. This is possible as VMs are isolated from each other by the hypervisor, allowing them to share the same physical resources. Additionally, modern VMs can be live-migrated [21] and will run on heterogeneous hardware. While consolidating under-utilized applications is easy, consolidating even moderately used applications can be difficult. This is because VMs are not entirely isolated from each other, and virtualization adds additional overhead. Thus, two VMs running on the same physical host can have an impact on each other's performance; as shown in [12] and [13].

To achieve the greatest amount of capital and energy savings, VMs must be placed to minimize the number of physical hosts required. However, a placement scheme must also ensure that the applications' performances remain at an acceptable level. To achieve this, VMs must be placed in such a way as to minimize the performance impact they have on each other. Current placement schemes primarily focus on setting utilization threshold levels. However, resource utilization levels can be a poor indicator of application level performance**.** This suggests that VM placement schemes should not be solely based on the idea of bin-packing resource utilization levels.

Commercial VM placement technologies, such as VMware Distributed Resource Scheduler (DRS)[4], place VMs based on resource utilization levels. DRS uses the VMs' CPU utilization level and RAM usage commitment to automatically decide which VMs should be placed on which physical hosts. VMs are then migrated between hosts as resource utilization levels change. Placement schemes such as this rely on the assumption that resource utilization levels reflect application-level performance. However, as resource utilization levels do not always reflect application level performance, such a scheme cannot easily guarantee application-level SLO.

In this paper we show that a VM management system can model multi-tiered applications to guarantee application-level SLO. This would allow system administrators to choose performance guarantees, such as response time < 500ms, without having to manually configure resource allocations. We show how the applications' model can be scaled to unobserved utilization levels, to allow SLO guarantees despite varied host workloads. Additionally, scaling the applications' model can predict the performance impact on an application before migrating VMs to or from a host where one of the application's tiers resides. Lastly, we show that modeling applications can help to more effectively and flexibly place VMs over a threshold based approach. For example, an application's VMs tiers can be placed to minimize power usage, or to minimize the risk of a certain response time being exceeded.

The paper is organized as follows: In Section 2 we discuss related works. In Section 3 we describe our system. In Section 4 we describe our experimental setup. In Section 5 we evaluate our results, followed by our conclusion in Section 6.

2. Related Works

There are many works on maximizing resource utilization levels and increasing efficiency in the virtual environment [5], [6], [7]. Existing commercial products are also available to facilitate the task of managing and relocating VMs. For example, VMware DRS [4] monitors the CPU and memory usage of VMs and migrates them to balance utilization levels. Similarly, VMware Distributed Power Management [4] minimizes the power usage of a data center by migrating VMs from lowly-utilized hosts and powers them off. Both systems focus on maintaining CPU and memory usage. Our work focuses on service level performance.

Recent efforts such as [8], [14], [15] and [16] have attempted to further increase resource utilization levels by migrating VMs. Each VM's resource utilization level is monitored and VMs are migrated to new hosts such that host resource utilization is maximized, and no host is overloaded. Kochut et. al [14] consider both autocorrelation and a periodogram to decide which VMs are best candidates to be placed together. Ideally, colocated VMs should have a low probability of overloading the host. Hermenier et. al. [15] consider the order that the migrations occur in addition to which VMs to migrate to minimize the impact of the migrations on system performance.

Another method to maximize resource utilization levels is overbooking resources. Urgaonkar et. al. [9] shows that a 500% increase in utility can be achieved by overbooking hosts by 5% of their peak load values. This only causes a 4.6% decrease in overall throughput. However, the study focuses on a shared hosting environment, not a virtual one, and considers neither contention nor the overhead caused by a virtual environment.

To maintain end-to-end service level performance, Stewart et.al [11] offers a response time prediction model. The model is based on an identified trait model for multi-tier applications. Their work focuses on predicting the service response time, based on pre-identified trait model relationships between processor properties and observed response time. Liu et. al. [18] use an autoregressive model to control CPU allocation. This allows VMs to be assigned a certain resource level as to normalize multiple applications' performance. Padala et. al. [19] and [20] have further used an autoregressive moving average to assign VM multiple resources.

3. Motivation

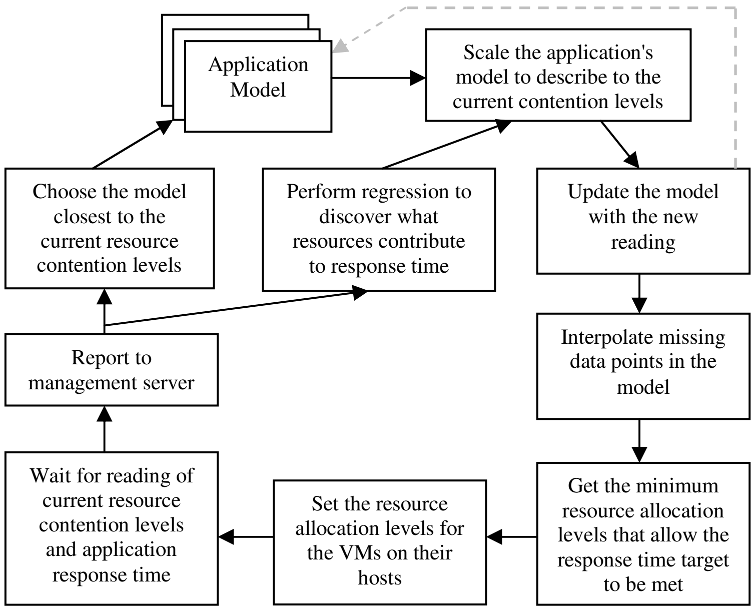
The motivation behind our work is to remove the need for administrators to perform resource allocation in the virtual environment. Our system aims to achieve SLOs by automatically allocating resources when they are required by a VM. Resources are then taken away and reallocated to other VMs as resource needs change. In a non-virtualized datacenter, applications avoid performance degradations by being isolated and run on dedicated hardware. However, this typically means low resource utilization levels, resulting in high hardware and energy costs. It is therefore attractive to place applications within VMs to reduce these costs. However, once applications are placed in a virtual environment, they must contest for resources as they are no longer entirely isolated. This can cause applications to suffer from performance degradations.

To ensure applications perform satisfactorily, Virtual Machine Monitors can be set to allocate a certain amount of hardware resources to each VM. There are however, a number of problems with current VM management systems. Firstly, administrators typically need to set the resource allocation levels manually. This requires administrators to monitor their applications' performances, and set each VMs' resource allocation and priority in the VM management system. This task can be made more difficult if the VMs' resource requirements frequently change. Secondly, the resource allocation levels only guarantee that a VM will receive a certain share of a resource. They do not provide any application-level performance guarantee. This can lead to lower hardware utilization levels, as administrators will typically over-provision resource allocations to ensure satisfactory performance. Lastly, administrators must manually set the utilization levels at which VMs will be migrated to and from hosts. This can again lead to lower hardware utilization as migration thresholds must be set low enough to ensure application-level performance does not suffer due to high resource contention.

To address these problems, our system monitors application-level performance and automatically allocates VMs the minimum level of resources they need to meet an application-level SLO guarantee. Our system works by monitoring the applications' performances at various user, resource allocation, and resource contention levels. Resource contention occurs on a host when multiple VMs require the use of the same resource. Once our system has multiple readings at different values, it can interpolate the minimum resource allocations needed to achieve a certain response time.

Figure 1 shows the basic flow of information in the management system. The process starts by an application reporting its response time and the level of resource contention on each host where one of its VMs resides. The management system then chooses the model that best describes the application's response time based on the current resource contention levels. Initially, this model will be empty as the management system does not have any data about the application. The model is then stretched based on how far the readings in the model are from the current resource contention levels. The missing data points in the model are then interpolated from the data that is available. The minimum resource allocation levels that allow the application to meet its response time target are then found in the interpolated model. Finally, the resource allocations are set on the hosts, and the hosts wait to take a new reading to report to the management system.

The applications' performance models are created automatically by analyzing the performance achieved at the various resource allocation levels. Although such models could contain millions of potential data points, we have found that a model can be constructed with only 10's of data points. Although each application will have a unique model, in future work it may be possible to apply a generic model to different types of applications, and then quickly tailor them with even fewer data points.



Application Model
Scale the application's model to describe to the current contention levels
Choose the model closest to the current resource contention levels
Perform regression to discover what resources contribute to response time
Update the model with the new reading
Report to management server
Interpolate missing data points in the model
Wait for reading of current resource contention levels and application response time
Set the resource allocation levels for the VMs on their hosts
Get the minimum resource allocation levels that allow the response time target to be met

Figure 1: Management system flow

In addition to interpolating the minimum resource allocations needed to achieve a certain response time, our system can also interpolate a response time value for a given resource contention level. This can help predict the performance of a VM before it is migrated to or from a host. This allows migration decisions to be made more flexibly, as they can be based on VM performance, rather than occurring at a fixed threshold.

As many of today's datacenter applications rely on multiple tiers, our system allows for this. Our system sets the resource allocations at each tier, such that the total response time experienced by the end user is below the SLO target. This allows an administrator to configure a single SLO value for an entire application stack. This is in contrast to current management systems, where the resource allocation must be configured individually for each tier.

3.1 Monitoring

To collect the data we need for our system we record the application's response time at its first tier; as shown in Figure 2. Throughput based applications can be monitored in a similar fashion, with throughput per time period recorded rather than response time. While monitoring response times at each individual tier could possibly provide a more accurate model, such monitoring would incur a significant overhead. Additionally, monitoring at intermediate tiers does not always reflect the overall performance characteristics experienced by the end-users.

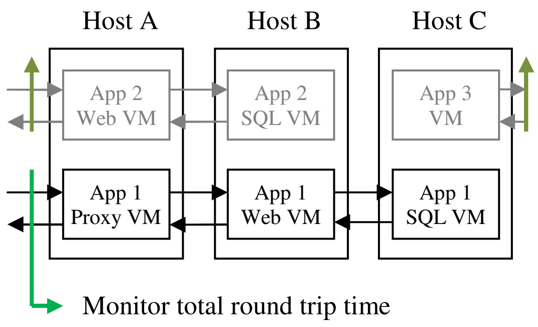


Figure 2: Response time monitoring

The data we capture are the applications' average total response time, CPU utilization, and storage and network throughput. All of the data are captured outside of the VMs, thereby not requiring a client to be inside the VMs. To allow our system to react quickly to changes, we take a reading every 10 seconds. This period could be increased or decreased as needed, depending on the system being controlled.

After the data is captured, it is passed to a server and added to our model. The model then interpolates the resource allocations that each VM should receive to meet a specified response time and chooses the minimum value. These resource allocations are then set on each host so that each VM receives the amount of resources calculated by the model, as shown in Figure 3.

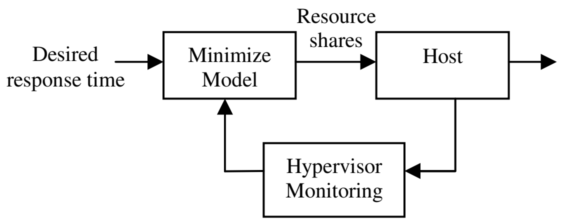


Figure 3: Control flow

3.2 Model Interpolation

Once the data is reported to our management system it is added to an application system model. An application's system model describes the previous response time values that we have observed for an application at various user, resource allocation, and hardware contention levels. We then use this model to predict the minimum resource allocations an application's VMs require to meet a certain response time.

To predict the required resource allocation levels we must first identify trends in the data. Figure 4 shows the effect of CPU contention on the host containing the web tier of TPC-W. The CPU contention is the total CPU utilization minus the amount used by the VM itself. As shown, the response time curve follows an exponential distribution. As the data closely fits an exponential distribution very few points are needed during run time to interpolate estimated resource allocation values.

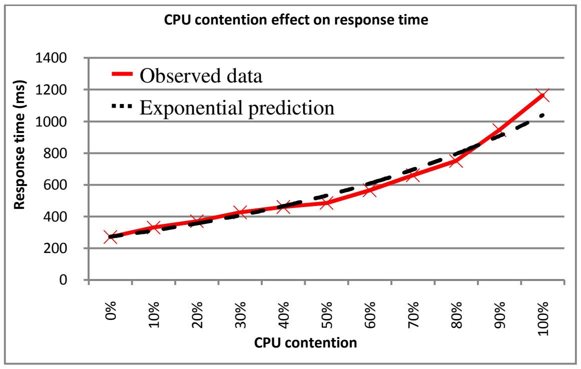


Figure 4: CPU contention and response time degradation

Figure 5 show the response time of TPC-W as the web tier has its CPU allocation changed from 10% to 100%. The resource allocation levels are currently capped to a minimum of 10% in our system as we have found that response times quickly approach infinity (the website crashes) for extremely low resource allocation values. Both the proxy and SQL tiers were set to 80% CPU allocation. As shown, for 45%-100% CPU allocation the response time for all four contention levels can be roughly predicted by the same linear function. For allocation values less than 45%, each contention level follows its own steeper linear function. This occurs as the web server tier is not the bottleneck of the application until it receives less than 45% CPU allocation.

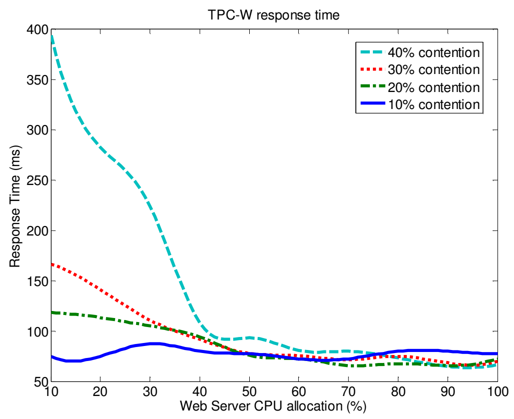


Figure 5: TPC-W response time with proxy and web server set at 80% CPU allocation

If the SLO guarantee we are trying to fulfill is 100ms, for example, Figure 5 would suggest that we allocate the web tier 45% of the CPU share if the CPU contention on the host is 20% or above. However, this only considers a single tier of the application. Figure 6 shows the response time curves when the web tier's CPU allocation is changed from 10% to 100%, but the proxy tier's allocation has been reduced to 30%. In this situation, there is no way to meet the 100ms response time goal if the contention on the host is more than 10%. This is because the proxy tier has become the application's bottleneck, so assigning more resources to the web tier will not significantly improve the response time. Because of this, it is clear that to minimize the resource allocations the model must include every tier of the application as a dimension.

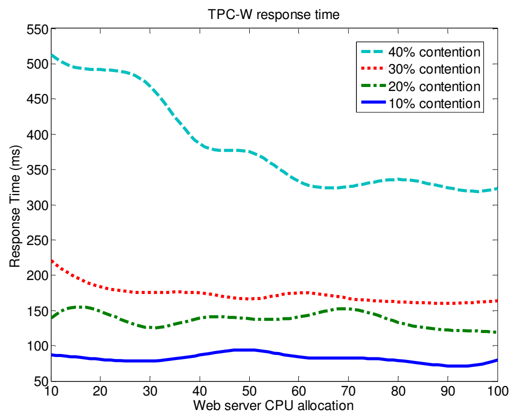


Figure 6: TPC-W response time with proxy set at 80% CPU allocation

Figure 7 shows the surface plot for the TPC-W proxy and web tiers with 300 active users and 40% CPU contention on each host. It should be noted that our system uses data from every application tier and from multiple hardware components. However, displaying graphs with more than three dimensions is difficult.

While Figure 7, 8 and 9 contain hundreds of data points to show the complete resource allocation to response time model, the runtime model does not require this much data. If, for example, the administrator has set 150ms as the SLO target, each model will contain points around that response time, but only a few points for the rest of the model. For example, in Figures 8 and 9 the model will mostly need to record data points between the dotted lines. In addition to having to store less data points, being able to characterize the application with fewer data points helps the model converge and adapt to changes quickly.

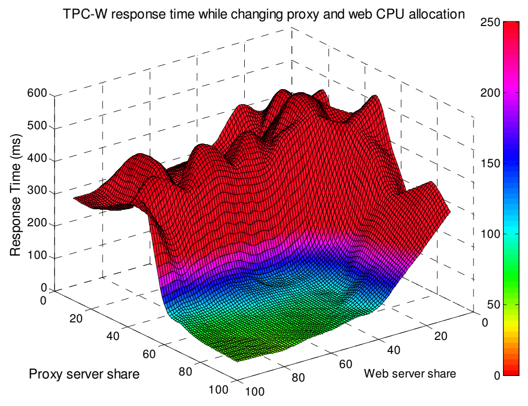


Figure 7: Proxy and Web tier CPU allocation response times for 40% CPU contention

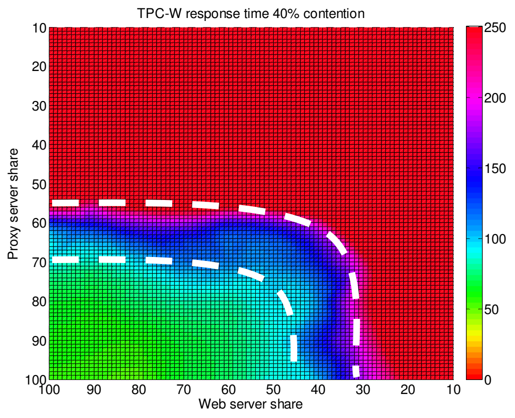


Figure 8: Proxy and Web tier CPU allocation response times for 40% CPU contention

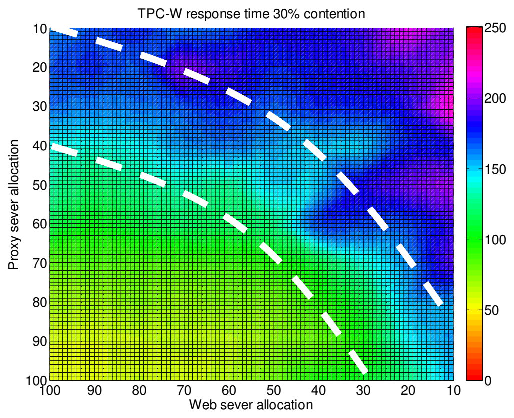


Figure 9: Proxy and Web tier CPU allocation response times for 30% CPU contention

3.3 Dimensional Reduction

As there are potentially thousands, or even millions, of resource contention combinations, it is infeasible to keep a model for every combination we encounter. Instead, we keep a subset of models, and scale the response time values to fit the current contention levels. To achieve this scaling we use the same data used in the resource allocation to response time models (Figures 7, 8 and 9), but instead interpolate contention to response time for a given resource allocation level. We use piecewise multiple linear regression to estimate the value that each point in the model should be scaled by. When we are estimating resource allocation values for a resource contention level that we do not have a model for, the management system needs to choose the model that most accurately represents the current resource contention levels. The model chosen to be scaled is the one with the smallest Euclidean distance from the current resource contention levels.

Figure 10 shows the actual response time for TPC-W and the estimated response time calculated using the regression coefficients. The data shown is a subset of data points where the CPU contention is between 10% and 40% for each tier. As can be seen in Figure 10, the estimated and actual response times are highly correlated, as would be expected given the fit of the data shown in Figure 4.

Figure 11 shows the response time increases as the number of users increase; in this case there is 10% CPU contention on each of the hosts where the TPC-W VMs are placed. As can be seen, the response time increases exponentially with the number of users. This can be accurately represented by linearly scaling three copies of an application's resource allocation model.

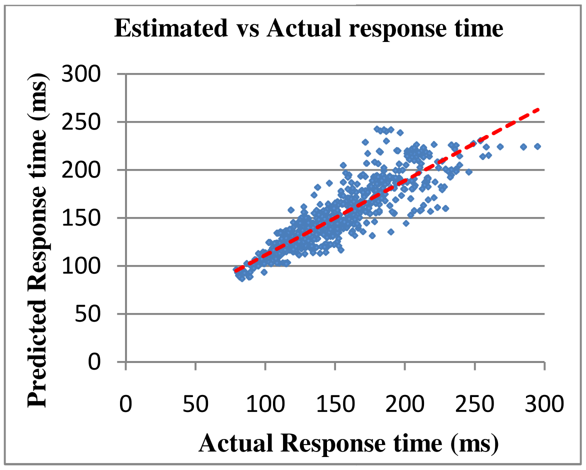


Figure 10: Estimated and Actual TPC-W response time

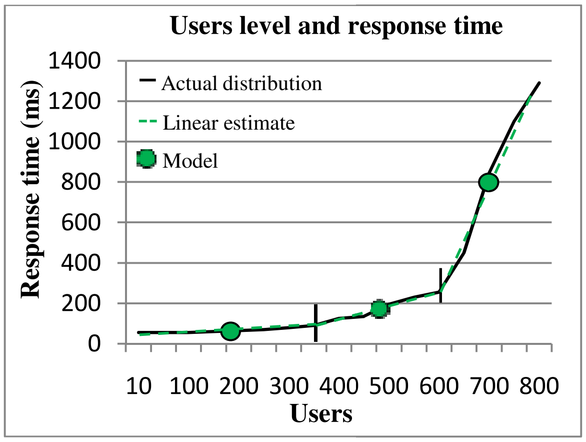


Figure 11: Response time increase vs. user level

Figure 12 shows the degredation in TPC-W's responce time at various CPU contention levels. The responce times shown are when the web tier is assigned 50% or 10% CPU allocation. At 50% CPU share allocation the CPU contention has little affect on the response time. This is because the web tier recieves CPU cycles very frequently, and is not the application's bottleneck. At 10% CPU share allocation the response time quickly degrades to almost a 50% increase in response time with a 10% increase in CPU contention. Even though the CPU had over 40% free cycles, the web tier does not receive its cycles promply enough, cauing it to become the bottleneck tier and causing degraded response time.

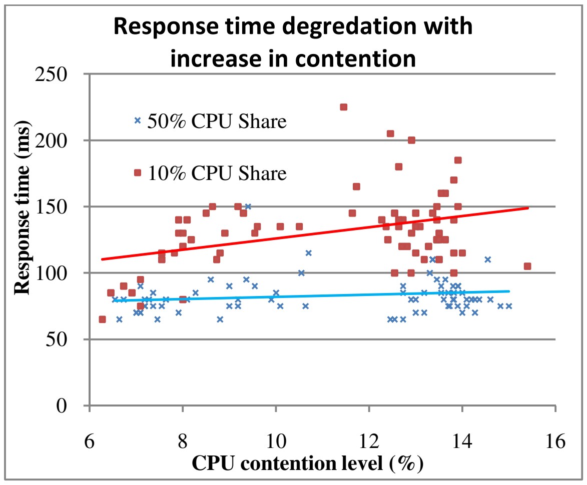


Figure 12: Regression values used to stretch a model

To scale an application's performance model, we multiply the model from the current contention level to the new one for each resource allocation level. For example, if we wanted to know the response time at 20% CPU contention and 50% CPU share allocation we would estimate 73 + 1.1 * 20 = 95ms. If our SLO target is 100ms, we would know that we could place the web tier on a host with 20% CPU contention if it could receive 50% of the CPU share allocation. However, if the host only had 40% CPU share allocation remaining, the estimated response time would be 80 + 1.15 * 20 = 103ms. Therefore, we would not expect that we could palce the web tier on that host.

4. Experimental Setup

4.1 Infrastructure

Our experiments are setup on a flat local area network using commodity hardware. The host operating system is Fedora 12 with Linux kernel 2.6.31. We use KVM as our hypervisor. The VM hosts consist of three nodes with tri-core 2.1 GHz CPU, 4GB RAM. The test clients consist of two nodes with quad-core 2.66 GHz CPU, 4GB RAM. The storage node contains a dual-core 2.8 GHz CPU, 4GB RAM.

The network topology we use is two flat-networks each with one switch: the user data network and the management network. Each physical host has two network interface cards (NICs). One NIC is connected to a user data network using a 24-port Gigabit switch. The user network carries all of the user workload and benchmark traffic. The other NIC is connected to a management network using a separate Gigabit switch as shown in Figure 13. The management network carries management-related commands and network attached storage traffic for the VMs' virtual disk images.

The storage system is hosted on two-spindle RAID-0, 2TB, 7200rpm hard disks. The storage server exports an NFS share. All virtual machine images are served from this location. To ensure network storage was not the bottleneck in our system, we benchmarked the network storage and found it more than capable of handling all of the VMs' disk traffic.

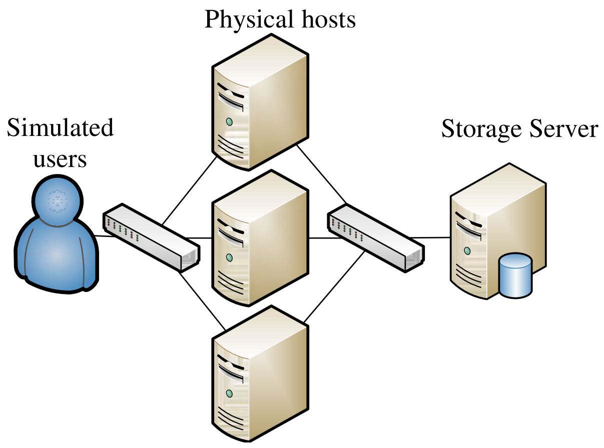


Figure 13: Test bed setup

4.2 Workloads

To test our system we use the TPC-W benchmark suit [22]. We use TPC-W as a test of a real-world delay-sensitive application. TPC-W mimics an online-bookstore application. It consists of an Apache web proxy front-end, a Tomcat application server, and a MySQL database back-end. There are 15 types of page requests. The benchmark client is a closed-loop client which simulates multiple users concurrently accessing the server. TPC-W's performance is measured based on response time for each action performed.

5. Results

In this section we discuss the results from our system. We test our system by running the TPC-W benchmark with each of the application’s tiers on a separate host. Each host also contains another VM running an Apache web server hosting computationally intensive web pages. The additional VMs are used to create resource contention on the hosts. They represent other applications that would undoubtedly also be running in a shared virtual environment. The number of requests per second to each Apache server was varied throughout the experiments to change the resource contention levels.

5.1 Meeting SLO target

Figure 14 shows the resulting response times of TPC-W when the resource allocation levels are set manually and when they are controlled by our system. When the resource allocations are set manually, each tier receives the same resource allocation on each host. For example, in the 50% resource allocation experiment, each tier has a fixed 50% resource allocation throughout the experiment.

As can be seen in Figure 14, when using our system TPC-W's response time closely follows the SLO target that is set. It is expected that the response time will oscillate above and below the SLO target as our system attempts to make the median response time equal to the SLO target. It is also evident from Figure 14 that the response time when using our system is usually faster, rather than slower, than the SLO target, and therefore averages to faster than the required SLO value. This is due to the resource allocation optimizer being cautious in its estimates. This is a conscious design decision, as a system that constantly over performs is more useful than a system that constantly under performs.

It can also be seen in Figure 14 that setting the resource allocation levels manually does not always produce a consistent response time. This is because resource contentions may increase over time, but the resource allocations do not. When the resource allocation is set to 50%, TPC-W's response time is faster for a longer period of time than when the SLO target is set to 150ms. However, at time period 480, a 50% resource allocation is no longer sufficient to continue providing that fast response time. However, with a dynamically set resource allocation our system can keep providing the same response time despite the CPU contention increase.

Test	**RT average**	**Resource allocation average**	**Apache VM average**
SLO = 100ms	89ms	48%	125ms
SLO = 150ms	127ms	35%	107ms
50% resource allocation	150ms	50%	120ms
10% resource allocation	355ms	10%	83ms

Table 1: Response time for TPC-W and contention workload

As can be seen in Table 1, despite the 50% resource allocation test having a faster response time for a longer period of time than the SLO 150ms test, its final average response time is greater. Additionally, the SLO 150ms test uses on average 15% less resources to achieve this faster average response time. As TPC-W uses less resources in the SLO 150ms test, the Apache workload on the host receives a greater share of resources; thus reducing its average response time from 120ms to 107ms. This is because the optimizer does not needlessly overprovision TPC-W, allowing the host scheduler to allocate remaining resources as needed. This shows that dynamically setting the resource allocation levels can not only guarantee a specified response time, but is also a more efficient use of resources. In this case, both applications have benefited from faster response times, despite our system only guaranteeing one of them.

Comparing the two tests with the closest resource allocation levels, we find that dynamic resource allocation helps achieve a faster average response time while using overall fewer hardware resources. Even excluding the final 120 readings, where the 50% allocation test performed poorly, dynamic allocation still performs faster, with an average response time of 89ms vs. the static allocation average of 106ms.

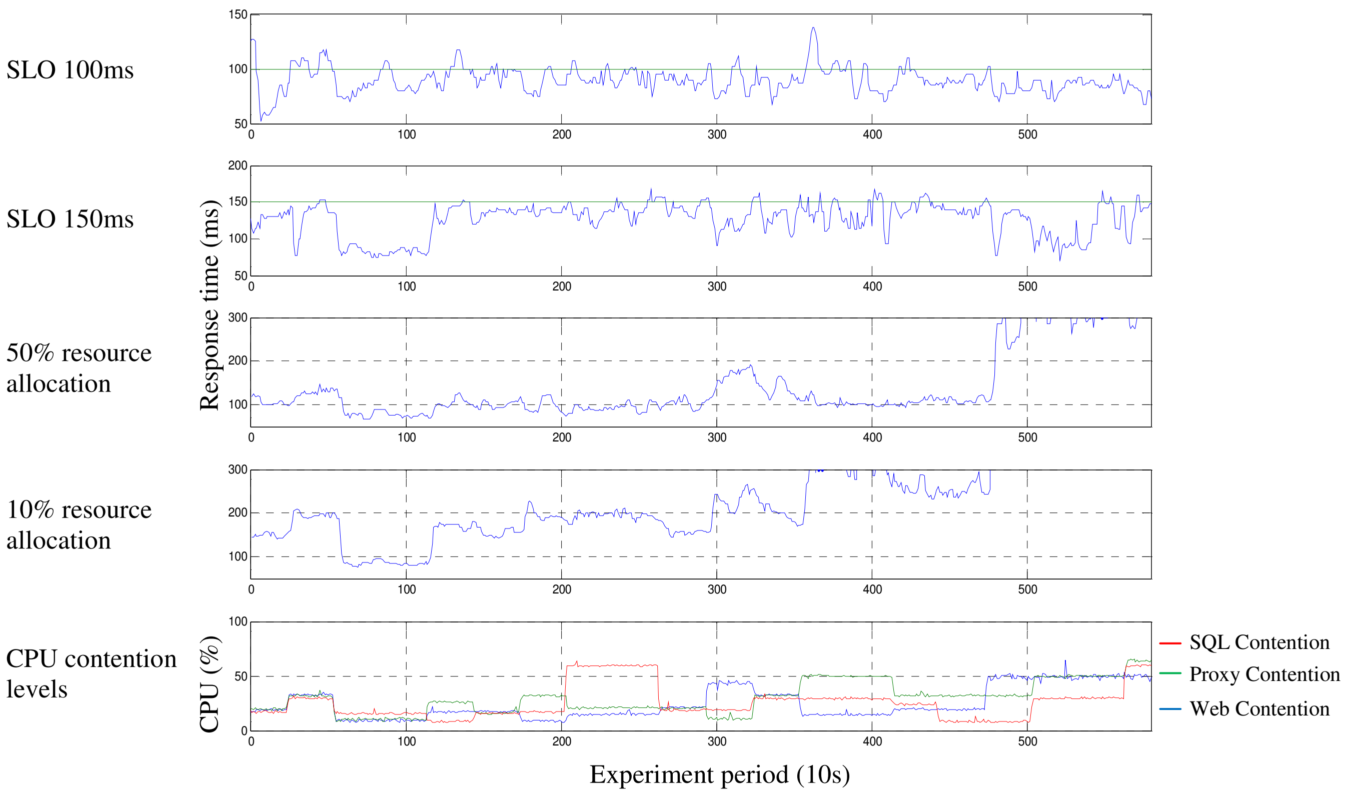


Figure 14: Response time results for dynamic and static resource allocations, changing CPU contention

5.2 Resource Allocation

Figure 15 shows the resource allocation levels that TPC-W received for the SLO 100ms and 150ms tests. The other two tests remain at 50% and 10% allocation throughout and are not shown.

At time period 200 it can be seen that the CPU contention on the SQL VM's host jumps 40%; however, the resource allocation only increases roughly 10%. This shows an advantage of modeling and predicting the application's performance over a more simple resource control scheme, such as increasing the resource allocation by a fixed factor of CPU contention. The regression analysis identifies that the CPU contention on the SQL VM's host does not cause large increases in response time. Therefore, when a model is used to predict the resource allocations for the new contention level, the scaling factor is low. This is in contrast to time period 110, when the CPU contention on the web server VM's host increases by 10%. In this case, the resource allocation increases by 20% in the SLO 150ms test and by 30% in the SLO 100ms test. This is because the model has correctly predicted that increased CPU contention on the web server VM's host will cause an increase in response time and has scaled the resource allocation model accordingly. We can see that the system predicted the correct resource allocation increases in both cases, as the response times for the SLO tests in Figure 14 both change to the configured SLO level at time period 110.

5.3 Change in user levels

Figure 16 shows the TPC-W response time when the number of users is varied during the experiment. We again configure our system to meet either a 100ms or 150ms response time SLO. We also experiment with the VMs resource allocations set statically to either 50% or 10%.

It can be seen from Figure 16 that our system can dynamically adjust resource allocations to meet an SLO despite a varying user level. Our system keeps the response time near the SLO target, whereas the static resource allocation causes response time to vary from 100ms-400ms.

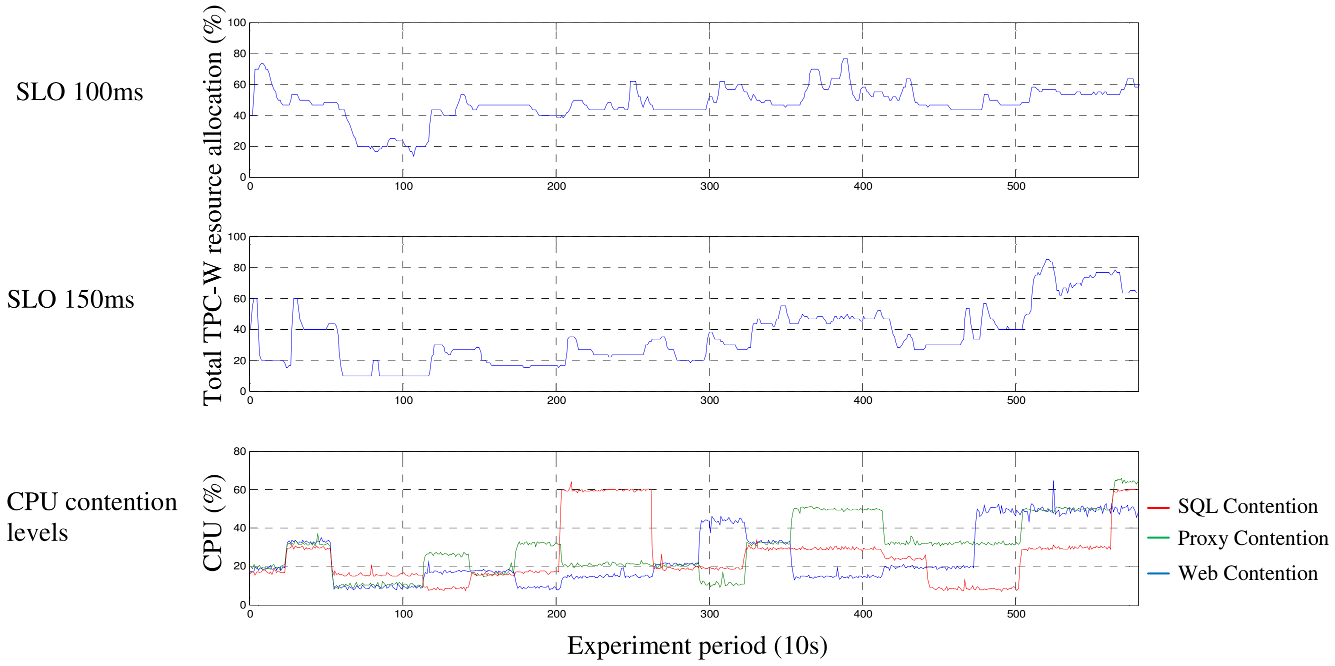


Figure 15: Allocated resource levels for dynamic resource allocation test

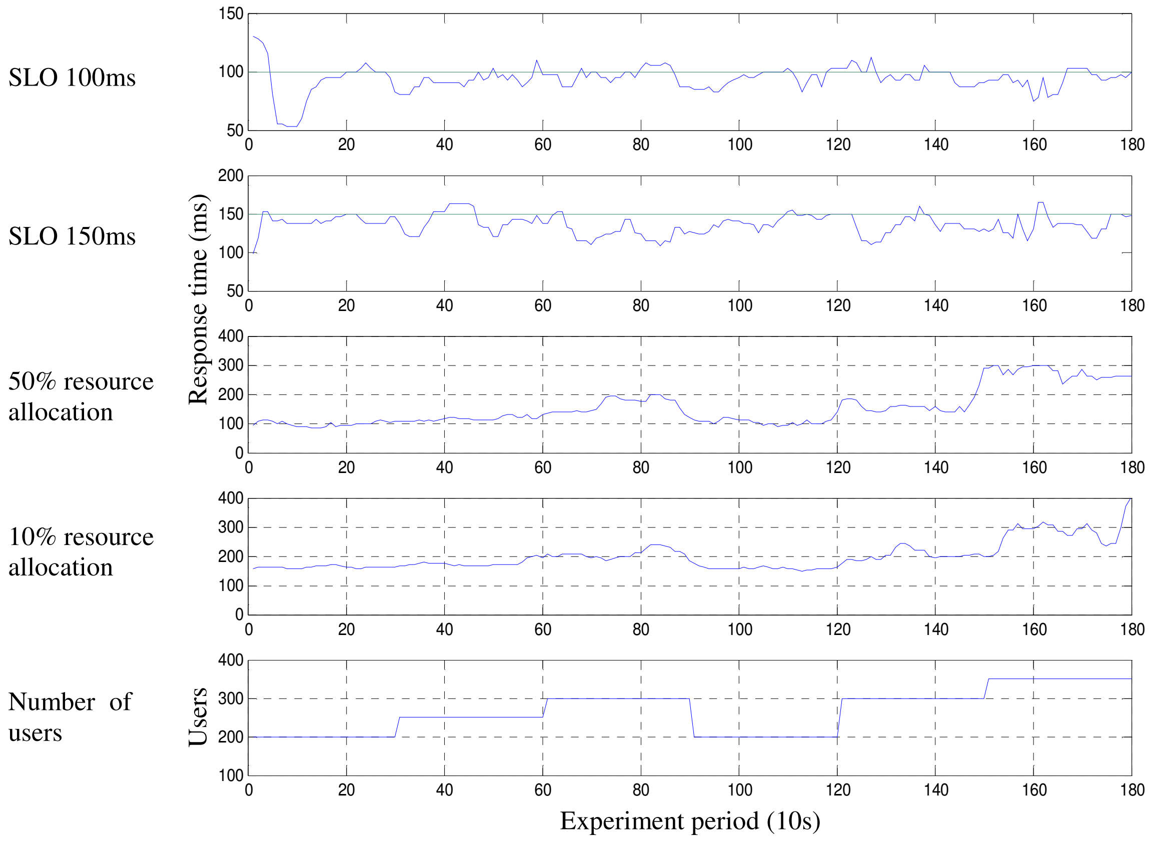


Figure 16: Response time results for dynamic and static resource allocations, changing user level

6. Conclusion and Future Work

In this work we have shown that applications comprised of multiple VM tiers can meet SLOs by dynamically allocating host resources. We show that by capturing an application's previous performance, we can model and predict the minimum amount of resources it needs to meet an SLO. Additionally, we show that these models can be stretched to changes in host utilization levels. This allows the resource allocation to be quickly altered when resource utilization levels change.

We evaluate our system using TPC-W and setting response time SLO targets. The host utilization is then varied throughout the experiments. Our system adapts to the changes in host utilization levels, and helps maintain TPC-W's response time within the SLO target. Our system also assigns the minimum amount of resources required to meet the SLO, allowing the other application running on the same hosts to improve its performance.

Although our system allows applications to meet SLOs, minimizing the total amount of resources used by each application may not be the most desirable goal in a data center. As VM migration causes both performance degradation and increased utilization, assigning resources in such a way as to lower the number of migrations may achieve lower global resource utilization than attempting to minimize resource allocation alone. Additional study would be needed to analyze the application specific performance degradation caused by migration.

While our current control scheme ensures that VMs receive the correct amount of resources to meet an SLO, it does not actually provide a hard guarantee about the number of violations. In future work we will bound the number and severity of SLO violations to provide administrators with hard guarantees about application level performance.

Additionally, rather than starting with a blank slate for each application, we hope to identify common traits between applications. This will allow performance models to be created and adapted more quickly, and could allow for different modes of control for different application types. This could potentially make the task of bounding the number of SLO violations easier.

7. REFERENCES

[1] VMware vSphere. www.vmware.com/products/vsphere/

[2] Citrix XenServer. http://www.xensource.com/

[3] Microsoft Hyper-V Server. http://www.microsoft.com/hyper-v-server/

[4] VMware Infrastructure: Resource Management with VMware DRS.

[5] Carrera, D. et. al. Utility-based placement of dynamic Web applications with fairness goals, NOMS 2008.

[6] Karve, A., et. al. A. Dynamic placement for clustered web applications. WWW 2006.

[7] Madhukar Korupolu, Aameek Singh, Bhuvan Bamba, Coupled placement in modern data centers, SPDP 2009.

[8] Choi, et.al. Autonomous learning for efficient resource utilization of dynamic VM migration. ICS 2008.

[9] Urgaonkar, B., Shenoy, P., and Roscoe, T. Resource overbooking and application profiling in shared hosting platforms. SIGOPS Oper. Syst. Rev. 36, SI Dec. 2002

[10] Karve, A., et. al. A. Dynamic placement for clustered web applications. WWW 2006.

[11] Stewart, C. et. al. A dollar from 15 cents: cross-platform management for internet services. USENIX 2008.

[12] Cherkasova, L, et. al. Comparison of the Three CPU Schedulers in Xen

[13] Somani, G. Chaudhary, S., Application Performance Isolation in Virtualization, CLOUD `09, IEEE International Conference on Cloud Computing, 2009

[14] Kochut, A., Beaty, K., On Strategies for Dynamic Resource Management in Virtualized Server Environments, IEEE MASCOTS, 2007

[15] Hermenier, F., et. al., Entropy: a consolidation manager for clusters, ACM/Usenix International Conference On Virtual Execution Environments, ACM SIGPLAN/SIGOPS, 2009

[16] Verma, A., Ahuja, P., Neogi, A., pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems, Middleware 2008, 2008

[17] Bobroff, N.; Kochut, A.; Beaty, K., Dynamic Placement of Virtual Machines for Managing SLA Violations, Integrated Network Management, 2007. IM '07

[18] Lui, X., et. al., Optimal Multivariate Control for Differentiated Services on a Shared Hosting Platform, Decision and Control, 2007

[19] Padala, P., et. al., Adaptive control of virtualized resources in utility computing environments, ACM SIGOPS Operating Systems Review, Volume 41 , Issue 3, 2007

[20] Padala, P., et. al., Automated control of multiple virtualized resources, ACM European conference on Computer systems, Cloud Computing, 2009

[21] Clark, C., et. al., Live migration of virtual machines, USENIX Networked Systems Design & Implementation - Volume 2, 2005

[22] TPC-W, http://www.tpc.org/tpcw/default.asp

RC2 – A Living Lab for Cloud Computing

Kyung Dong Ryu, Xiaolan Zhang, Glenn Ammons, Vasanth Bala, Stefan Berger, Dilma M Da Silva
Jim Doran, Frank Franco, Alexei Karve, Herb Lee, James A Lindeman, Ajay Mohindra, Bob Oesterlin
Giovanni Pacifici, Dimitrios Pendarakis, Darrell Reimer, Mariusz Sabath

IBM TJ Watson Research Center
Yorktown Heights, NY

Abstract

In this paper we present our experience in building the Research Compute Cloud (RC2), a cloud computing platform for use by the worldwide IBM Research community. Within eleven months of its official release RC2 has reached a community of 631 users spanning 34 countries, and serves on average 350 active users and 1800 active VM instances per month. Besides offering a utility computing platform across a heterogeneous pool of servers, RC2 aims at providing a living lab for experimenting with new cloud technologies and accelerating their transfer to IBM products. This paper describes our experience in designing and implementing a flexible infrastructure to enable rapid integration of novel ideas while preserving the overall stability and consumability of the system.

1 Introduction

Cloud Computing has become synonymous with ways to contain and manage IT costs for enterprises. Cloud Computing is a paradigm where compute capacity is made available to users in an on-demand fashion through a shared physical infrastructure. The expectation is that sharing hardware, software, network resources, and management personnel would reduce per unit compute cost for enterprises. Several vendors such as Amazon EC2, Google, and Rackspace have been providing commercial Cloud offerings. Though not enterprise-grade level yet, Cloud Computing has piqued the interest of several large enterprises, which have started deploying and experimenting with the technology for their test and development environments. IBM Research has developed and deployed a Cloud Computing platform called Research Compute Cloud (RC2) for use by the worldwide IBM Research community. The goals of RC2 are to establish an "innovation" platform for the IBM Research community and to serve as a "living" lab for the research technologies developed by the IBM Research community. The platform has been purposefully architected to facilitate collaboration among multiple research groups and encourage experimentation with cloud computing technologies. The platform also serves as a showcase of new research technologies to IBM customers and business partners.

The IT infrastructure of the IBM Research division resembles that of a global enterprise having many different lines of business spread across multiple geographies. IBM Research is a geographically distributed organization, consisting of several thousand research personnel spread across 9 research laboratories worldwide. Each IBM research lab operates its own local data center that is used predominantly for lab-specific research experiments. In addition, a portion of the data center infrastructure is collectively used for production workloads such as email, employee yellow pages, wikis, CVS servers, LDAP servers, etc. Critical production workloads can be replicated across different lab data centers for purposes of failover. This infrastructure is a substantial investment built over many years, and is very heterogeneous in its make up. For instance, IBM's POWER series systems and System Z mainframes are mixed with many generations of commodity x86 blade servers and IBM iDataplex systems.

The Research Compute Cloud (RC2) is an infrastructure-as-a-service cloud built by leveraging the existing IT infrastructure of the IBM Research division. Its goals were two-fold: (a) create a shared infrastructure for daily use by the research population, and (b) provide a living lab for experimenting with new cloud technologies. There were several challenges that the team needed to address to meet the two goals. The first challenge was to design and build a consistent infrastructure-as-a-service interface over a heterogenous infrastructure to meet the needs of the Research user community. The second challenge was to enable a true living lab where the Research community could develop and test new technologies in the cloud. The architecture of RC2 had to be flexible enough to enable experimentation with different cloud technologies at the management, platform, and application layers. All this had to be achieved without any disruptions to the stability and consumability of the overall infrastructure.

In this paper, we present our experience in building RC2. In Section 2, we present the architecture of RC2 to meet the two design goals. Next, we discuss the implementation of RC2 in Section 3. Section 4 presents our experience specifically in the context of pluggability and extensibility of the environment. We conclude the paper by discussing related work in Section 5 and future work in Section 6.

2 Architecture

As mentioned in Section 1, RC2 aims to provide a research platform where exploratory technologies can be rapidly introduced and evaluated with minimal disruption on the operation of the cloud. This requirement calls for a componentized, extensible cloud architecture.

Figure 1 shows the architecture of RC2 which consists of a cloud dispatcher that presents an external REST API to users and a collection of managers that provide specific services. For each manager, the dispatcher contains a proxy whose role is to marshal requests to and responses from the manager itself.

This architecture enables a loose coupling between the managers. Any manager only knows its corresponding proxy; there is no direct communication between managers. Different groups within IBM Research can work on different managers without anyone mandating how their code should integrate. Groups only need to agree on the APIs that the manager proxies will expose within the dispatcher.

The dispatcher is driven by an extensible dispatch table that maps request types to manager proxies. When a request enters the dispatcher (whether from an external source like an RC2 user or an internal source like one of the managers), the dispatcher looks up the request's signature and dispatches it to the manager proxy responsible for that type of request. A new manager can be added simply by adding its request type and mapping information to the table.

Another benefit of this design is that, because all requests pass through the dispatcher, features such as admission control, logging and monitoring can be implemented easily in the dispatcher. A potential drawback is that the dispatcher becomes a bottleneck, but this problem can be solved by distributing requests among multiple dispatcher instances.

Figure 1 shows the managers that currently exist in RC2. The user manager authenticates users. The image manager catalogs, accesses, and maintains virtual-machine images. The instance manager creates, deploys, and manipulates runnable instances of the image manager's images. The security manager sets up and configures the network isolation of cloud tenants' security domains for communication both outside the cloud and with other security domains inside the cloud.

Distribution of functionality implies distribution of the system's state among individual components. This distribution makes it difficult to obtain a complete and consistent view of the system state during long-running requests (for example, instance creation), which complicates failure detection and recovery. Our architecture tackles this problem by requiring each manager to maintain and communicate the states of the objects it manages. For example, both images and instances have associated states, which can be queried by sending a "describe image" or "describe instance" request to the appropriate manager. Long-running requests are processed in two stages. The first stage synchronously returns an object that represents the request's result, and the second stage asynchronously completes the time-consuming remainder of the request and updates the object's state accordingly. Request completion (or failure) can be determined by querying the object's state.

Another challenge was to design a set of infrastructure-as-a-service APIs that could be implemented consistently across a heterogeneous pool of servers. Differences among platforms can be huge. For example, consider two different server platforms: an IBM xSeries blade server and an IBM pSeries blade server. The former runs a software virtual-machine monitor (in RC2, Xen or KVM) on commodity x86 hardware, while the latter runs a firmware hypervisor (PHYP), on IBM Power hardware (also referred to as System P). These two platforms support different operating systems, different image formats, and different mechanisms for creating instances from images.

For example, for Xen and KVM based VM instances, the images exist in raw (block) disk format. Deploying those images into new VM instances requires copying the disks onto the host's storage and mounting the file system of those disks to customize the images. The process is completely different for the AIX operating system that runs on the pSeries servers – the images exist in a special backup (tar-like) format, and are referred to as *mksysb* backups. Deploying those images into new PHYP instances requires booting off an initial *install kernel* which in turn drives the installation of the image files into the newly booted VM. The installation is achieved through a special central server called the Network Installation Manager (NIM), which creates filesystems for the instance, with files restored from the *mksysb* backups.

Our design supports multiple platforms by requiring

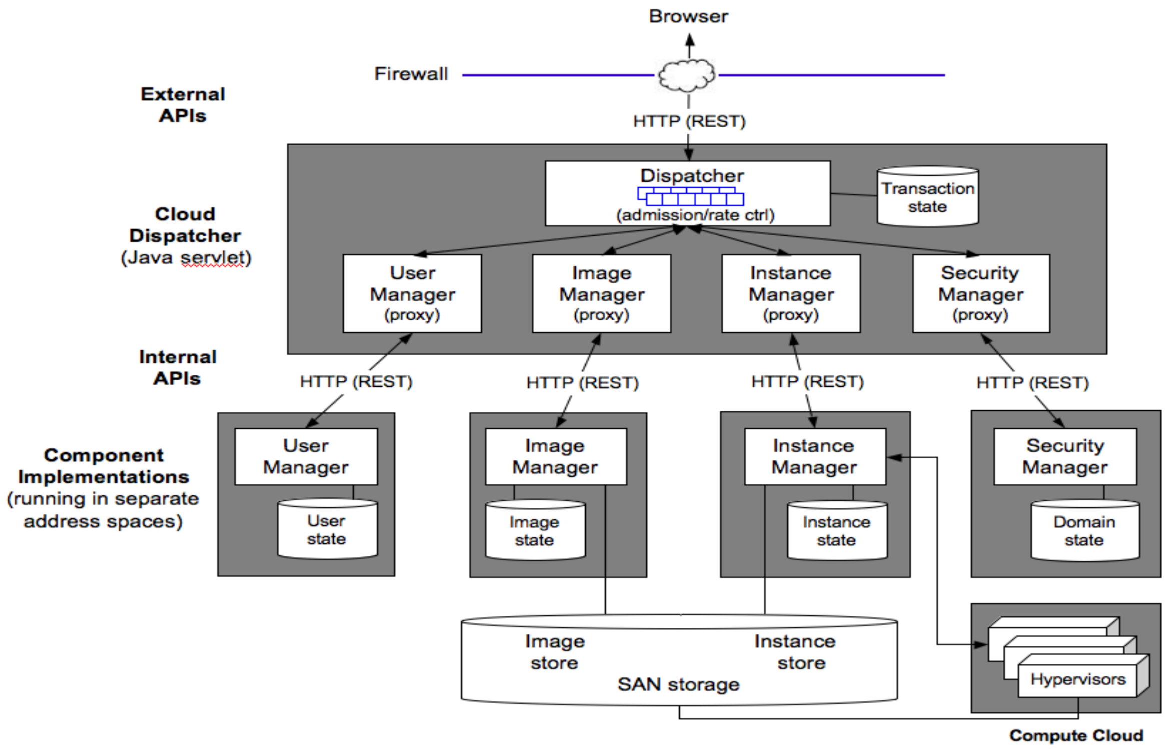


Figure 1: RC2 Architecture

that requests avoid platform-specific parameters. For example, both types of images are stored in the repository with the same format, has identical list of attributes, and can be queried in an identical manner. Similarly, the same instance creation API is used to start an instance for both image types (although the parameter values vary). The requester is not required to know the specific platform type of the image or instance that she is operating on. This approach minimizes the complexity of supporting multiple platforms, as only the instance manager, which receives requests for creating instances, must concern itself with differences among platforms.

3 Implementation

The RC2 hardware infrastructure is comprised of management servers, the host server pool, and the storage subsystem. The management servers host RC2 management services, provision and capture of virtual machines, and http access for users. The host pool houses the provisioned instances and consists of a collection of IBM iDataplex blades varing in size from 32GB-4way to 128GB-8way systems. The storage pool consists of a SAN subsystem that is shared across both the host servers and the management servers.

3.1 Dispatcher

The RC2 cloud dispatcher is composed of three layers: a REST servlet, the cloud manager, and several manager proxies. The REST servlet provides an HTTP-based REST interface to cloud operations. The interface can be used by programs as well as through a web-based graphical user interface. The manager proxies decouple interactions between user and cloud dispatcher and communication between the dispatcher and managers. This separation promotes flexibility of managers while allowing uniform cloud interfaces to users. Although, in the current implementation, all managers are accessed through REST APIs, they can be easily replaced with implementations that use different communication mechanisms such as Java Message Service (JMS).

The cloud manager sits between the REST servlet and the manager proxies, providing admission control and rate control using dispatch queues and request-handler threadpools. There are currently two types of dispatch queues: synchronous request queues and asynchronous request queues. The former handles short-lived cloud requests such as looking up image information and listing an owner's instances whereas the latter handles long-lived cloud requests such as creating an instance or capturing an image. The threadpool size of the synchronous request queue is typically set to a large value to allow more requests to be processed concurrently while that of the asynchronous request queue is limited to a number that matches the rate at which the back-end manager can process requests. The configuration of dispatch queues such as queue length and threadpool size can be changed at runtime through an administrative REST interface, which is designed to allow feedback-based rate control in the future.

3.2 Instance Manager

The instance manager keeps track of the cloud's virtual-machine instances. An instance-manager implementation must provide several basic services: "start instance", which adds a running instance to the cloud; "stop instance", which stops an instance; "delete instance", which removes an instance from the cloud; and a query service for listing instances and their state. Only the implementation of "start instance" is described here because it is the least straightforward.

Starting an instance involves four tasks: selecting a target host, creating and configuring an instance on that host (which includes choosing a security domain), retrieving and configuring the virtual-machine image, and finally starting the instance. Each task is implemented by plugins, so as to support a variety of host and image types.

The instance manager selects a host with the proper resources to run the user-requested instance. The current implementation uses a best-fit algorithm [7] that considers memory, cpu, disk, network connectivity, and host-specific requirements such as the host's architecture and virtual-machine monitor. Selecting the host also binds some instance parameters, including the IP address of the new instance.

The instance manager retrieves the image from the image manager and configures it for execution. Image configuration sets both user-specific parameters, such as ssh keys, and instance-specific parameters, such as the IP address. Some parameters are set by modifying the retrieved image before startup while others are set at startup-time by embedding an "activation engine" [3] in the image that runs the first time the instance boots. The instance-specific parameters are provided through a virtual floppy drive. The activation engine is designed for extensibility and can configure operating system, middleware, and application parameters.

Next, the instance manager instructs the chosen host to allocate a new instance. The details are host-specific; the current implementation includes plugins for AIX hosts and for x86 hosts based on Xen and KVM.

Finally, the instance manager starts the instance. The user is notified and a description of the new instance is sent to a database for compliance tracking.

3.3 Image Manager

The image manager maintains a library of images. The image manager cooperates with the user manager to control access to images and with the instance manager to create runnable instances of images and to capture images of runnable instances as images.

Each image has a unique image identifier, which names the image for access-control purposes. The library stores one or more versions of each image and each version has a version identifier, which names both data and metadata. The data consists of a set of files, including disk images and other files required to create a runnable instance. The metadata is a set of version attributes, such as a name, a description, and the identifier of the version's parent.

Version data is immutable. Therefore, if a disk image is modified by a running instance, it can be captured back to the library only as a new version, whose parent will be the version from which the instance was created. Some version attributes are mutable, such as the description, while others are immutable, such as the parent identifier. The access-control information associated with an image is mutable.

The most important image manager services are "checkout" and "checkin". Given a version or image identifier and a target URL, checkout creates a runnable instance from a version; if an image identifier is supplied, the most recent version of that image will be checked out. The target URL identifies a directory on the SAN where the instance manager expects to find the runnable instance and to which the image manager copies the version's data files. The image manager also places control files in the directory that, among other things, identify the source version.

Given a source URL, which identifies a directory on the SAN that was populated by a checkout, checkin creates a new version. Note that the directory's data files, including its disk images, may have been modified by instance execution. There are two kinds of checkin calls: the first creates a new version of the original image while the second creates the first version of a new image. Currently, only the second kind is exposed to RC2 users.

Both checkout and checkin are asynchronous calls. The instance manager invokes these two interfaces and tests for completion by polling a status file, which the image manager updates on completion, or by supplying the URL of a callback, which the image manager invokes on completion.

The image manager controls access to images in the library. Each image has an owner, a list of users and groups with checkout access, and a list of users and groups with both checkout and checkin access. Only the owner may update the lists. Each image manager call includes an owner and a list of groups to which the owner belongs, which the manager uses to verify that the caller has the required access for the call. The image manager assumes that a call's owner and group list is correct: the user manager is responsible for user authentication and the cloud dispatcher ensures that calls do not forge user or group names.

The image manager provides other services besides checkin and checkout. These include calls that list, describe, and delete images and versions, plus calls that update access-control lists. Deleted versions retain their metadata but lose their data files.

The image manager uses a file-granularity, content-addressable store (CAS) to maintain the image content [9]. The CAS saves space by guaranteeing that the same item is never stored twice. It also keeps the reference information necessary to garbage collect deleted image data.

3.4 Security Manager

RC2 has been architected with several mechanisms to provide security in a cloud environment. In particular, the security manager provides support for isolation between different cloud user's workloads in a heterogeneous, multi-tenant environment. The isolation model follows our established concepts of Trusted Virtual Domains (TVDs) [2] and a Trusted Virtual Data Center (TVDc) [10]. A TVD is a grouping of (one or more) VMs belonging to the same user that share a trust relation and common rules for communicating among themselves as well as with the outside world.

The security manager exports a broad API through the cloud dispatcher and provides extensive functionality for life-cycle management of security domains and their runtime configuration.

The security manager is currently built on top of modifications to the Xen daemon for the establishment and runtime configuration of firewall rules on virtual machines' interfaces in Domain-0. Our architecture makes use of the fact that in the Xen hypervisor all virtual machines' network packets pass through the management virtual machine (Domain-0) and firewall rules can be applied on the network interface backends that each VM has in that domain. This allows us to filter network traffic originating from and destined to individual virtual machines.

The extensions to the Xen daemon provide functionality for the application of layer 2 and layer 3 network traffic filtering rules using Linux's ebtables and iptables support. While a VM is running, its layer 3 network filtering rules can be changed to reflect a user's new configuration choices for the security domain a virtual machine is associated with. We support a similar network traffic filtering architecture with the Qemu/KVM hypervisor where we implemented extensions to the libvirt management software providing equivalent functionality as the extensions to the Xen daemon.

Functionality that the security manager provides for support of security domain life cycle management involves the following:

- Creation and destruction of security domains.
- Renaming and configuration of parameters of security domains.
- Retrieval of security domain configuration data.
- Modifications of security domains' network traffic rules.
- Establishment of collaborations between security domains of the same or different cloud tenants.

Altogether, the security manager adds 17 new commands to the dispatcher API.

The realization of the security domains concept drove extensions to several other existing components in the architecture. Extensions were implemented in the cloud dispatcher layer to make the new APIs visible to other management components as well as external entities. The instance-manager request that creates a virtual machine instance was extended with optional parameters describing the security domain into which a virtual machine is to be deployed. A new internal request was added to the instance manager for deployment of filtering rules associated with VM instances. Several previously existing workflows, which are part of the instance manager, were modified to notify the security manager of VMs' life cycle events as well as to perform configuration in the Xen management virtual machine (Domain-0).

3.5 Chargeback

We implemented a simple allocation-based pricing model to experiment with users' behavior in resource consumption under different pricing models. Users are charged for compute capacity based on a billable unit of "per instance hour consumed". This begins with instance creation and ends when an instance is destroyed. At this time, the same charges are incurred whether the instance is active (that is, running) or inactive (stopped). Rates differ by system type (XEN, PHYP, and KVM) and configuration (small, medium, large, and extra large). In addition, there are separate charges for end-user initiated transactions that lead to state changes of their instances (Initialize, Start, Stop, Reboot, Destroy). Charges are calculated on an hourly basis and are integrated with IBM's existing internal billing systems.

3.6 User Manager

The user manager authenticates users by invoking services available in the IBM infrastructure and associating authentication with sessions. It also manages user-specific information, such as ssh public keys, that can be queried by other managers during provisioning.

4 Experience

RC2 was released in a Beta version in early April, 2009, and officially released to IBM Research world-wide in early September, 2009. In this section, we present global usage statistics of RC2 and our experiences using RC2 as a research platform to experiment with new cloud technologies.

4.1 RC2 Usage

Within 11 months of its official production release, RC2 has served 631 distinct users spanning 34 countries. The image library has accumulated a collection of 2286 images, all of which derive from just three root images that were imported into the libary at the beginning. The number of images in the library grew starting about a week after the Beta release. The library grew modestly during the Beta testing period but has been experiencing faster growth since the official release in early September. The number of instances has grown at a similar rate; Figure 2 shows this growth since the production release.

The average number of active instances per month is also growing, reaching 1800 in the most recent month. This includes 102 instances of the System P type. On the average there are about 350 distinct active users per month, who consume a total of 600,000 virtual-machine hours.

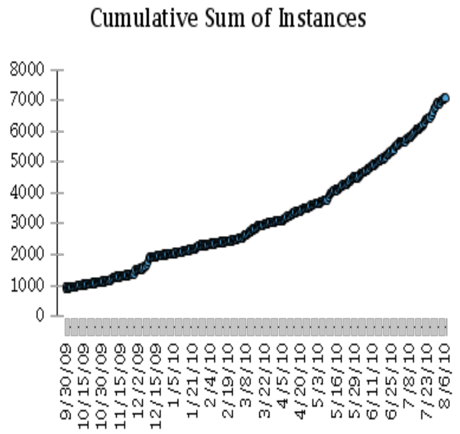


Figure 2: Instance Growth

RC2 was first released free of charge. When charges for instance ownership were introduced in early October, it had a dramatic impact on user behavior, as shown in Figure 3. There was a significant drop in the number of instances right after users received their first statements, leading to a drop in memory utilization. Interestingly, the number quickly bounced back, and memory utilization again approached pre-chargeback levels. We consider this to be a strong endorsement from our user community about the value of the service provided by RC2.

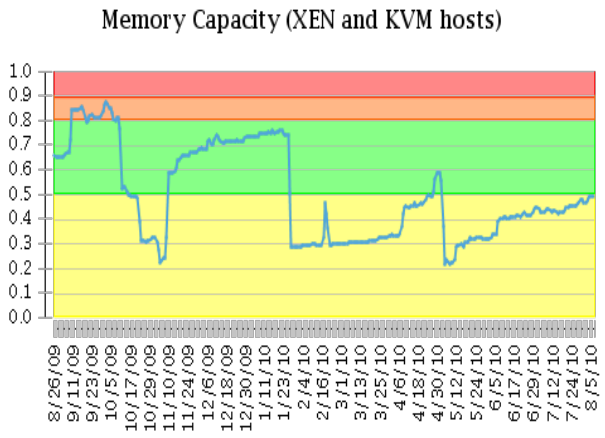


Figure 3: Cloud Memory Utilization
Percentage of memory allocated for instances as a ratio of total available memory.

4.2 RC2 as a Living Lab

In addition to its role as a production-quality IT offering for IBM's research organization, RC2 also serves as an experimental testbed for innovative cloud management technologies. To this end, we show how RC2's architectural emphasis on extensiblity and pluggability has helped facilitate these experimental activities.

The initial version of RC2 consisted of three managers: the image manager, the instance manager, and the user manager. The security manager was added to provide stronger isolation between multiple tenants' workloads in the cloud. While the security manager presented significant functionality enhancements, the core RC2 architecture remained essentially the same given that it was designed to be extensible from the start and most changes were contained at the cloud dispatcher.

The pluggable cloud dispatcher architecture enabled us to deploy an image manager based on the Network File System (NFS) for Research sites that lack a SAN storage environment. For these sites, we reimplemented the image manager interfaces using NFS as the backing store. As with the SAN, the file system is mounted on each hypervisor node so that images are locally accessible. The instance manager required no change as the NFS-based image manager supports the same set of requests as does the SAN-based image manager. The flexibility of RC2 allowed researchers to experiment with alternate implementations without requiring changes to other components.

Being a living lab implies that sometimes RC2 needs to deal with unusual infrastructure-level changes that are typically not seen in production systems. One such example is change of supported hypervisor types. Initially, RC2 adopted Xen as its only x86 hypervisor. Later on there was a strategic decision to switch to KVM, which means that RC2's entire Xen image collection needs to be converted to KVM.

Because RC2 is a production system, the conversion needs to be accomplished with minimal disruption to the user. This translates into two concrete requirements. First, the contents of all existing Xen images as well as instances need to be preserved. Users should just be able to start their existing images as usual without even noticing that the images will be in fact running on a KVM hypervisor. Similarly, when existing Xen instances are captured, they should automatically be converted to KVM without any user intervention. Second, conversion of the entire Xen image/instance collection needs to be achieved with zero downtime (except for the regularly scheduled maintenance window). During the conversion period, both Xen and KVM provisioning must be supported.

Our solution required multiple enhancements to be made to both the instance manager and the image manager. The instance manager, upon receiving a capture request, performed an on-the-fly conversion of the captured Xen image to a KVM image. The image manager was enhanced with a *locking* functionality that hid newly converted KVM images from the end user until the images were ready to be seen by the users. Again, the decoupled architecture of the RC2 system allowed individual component to be separately tested and replaced, making it possible to achieve the conversion without any disruption of the system.

The RC2 team successfully converted the entire Xen image collection (419 images) to KVM. The migration process started on May 6th, 2010 and ended on June 14th. During this whole period, the RC2 production system was continuously running with all functionalities enabled and no noticeable performance slowdown. The process was also completely transparent to the users. All conversion activities were shielded from the end users. End users did not notice any change of their images until the "conversion" day, at which point the newly converted images (with new image numbers) appeared on user's login view. Advance notice was sent to the users a few days earlier so they were prepared for this change on "conversion" day.

5 Related Work

Current cloud computing offerings focus on building an optimized, homogeneous environment for delivery of compute services to customers. Amazon's EC2 [1] and IBM's Developer Cloud [4] are examples of such offerings. By contrast, our work focuses on heterogeneity and providing a pluggable and extensible framework to serve as a living lab for cloud technologies. The open source project Eucalyptus [8] provides capabilities similar to those of Amazon's EC2 and could be used in a living lab, as developers can modify and extend the source. However, the project lacks support for heterogeneous platforms and a pluggable architecture.

6 Conclusion and Future Work

The RC2 project succeeded in achieving its two main goals: (1) it delivers high-quality cloud computing services for the IBM Research community and (2) it provides an effective framework for integration of novel ideas into a real cloud platform, rapidly enriching the evaluation of new technologies by offering meaningful, realistic user experience and usage/performance data. Many of these new technologies were adopted by newly announced IBM products in 2009 such as Websphere Cloudburst Appliance [6] and VM Control [5].

The current RC2 system is implemented only in the New York area data center. However, the RC2 services are available to all of the worldwide IBM Research Labs. In 2010, we plan to create RC2 zones in at least two other labs on two different continents.

The current RC2 production system has numerous monitoring probes installed at different points in the infrastructure and in the management middleware that runs the data center. These probes provide a rich set of real-time monitoring data, which is itself available as a service provided through a collection of REST APIs. We plan to use this feature to provide a simulated data center environment over the RC2 production environment, for experimental purposes. The simulated environment will behave as if it is the actual production environment underneath, by tapping into the real-time monitoring data provided by the probes.

7 Acknowledgments

To keep a system such as RC2 in production as it evolves rapidly and drastically can not be done without the contributions of many talented engineers and IT specialists. We thank the entire RC2 team for the intense effort.

We also thank our anonymous reviewers, and our shephard Matthew Sacks, for their insightful reviews and comments.

References

[1] AMAZON. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2`.

[2] BUSSANI ET AL. Trusted Virtual Domains: Secure Foundations for Business and IT Services. Technical Report RC23792, IBM Research, November 2005.

[3] HE, L., SMITH, S., WILLENBORG, R., AND WANG, Q. Automating deployment and activation of virtual images. White paper, IBM developerWorks, August 2007. `http://www.ibm.com/developerworks/websphere/techjournal/0708_he/0708_he.html`.

[4] IBM. IBM Cloud Computing. `http://www.ibm.com/ibm/cloud`.

[5] IBM. VM Control Enterprise Edition. `http://www-03.ibm.com/systems/management/director/plugins/syspools/index.html`.

[6] IBM. Websphere Cloudburst Appliance. `http://www-01.ibm.com/software/webservers/cloudburst/`.

[7] KWOK, T., AND MOHINDRA, A. Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications. *Services-Oriented Computing – ICSOC 5364* (2008).

[8] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The Eucalyptus open-source cloud-computing system. In *Proceedings of CC-Grid'09: the 9th IEEE International Symposium on Cluster Computing and the Grid* (Shangai, China, May 2009).

[9] REIMER, D., THOMAS, A., AMMONS, G., MUMMERT, T., ALPERN, B., AND BALA, V. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *The 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)* (March 5-7, 2008).

[10] STEFAN BERGER ET AL. Security for the cloud infrastructure: Trusted virtual data center implementation. *IBM Journal of Research and Development 53*, 4 (2009).

PeerMon: A Peer-to-Peer Network Monitoring System

Tia Newhall, Jānis Lībeks, Ross Greenwood, Jeff Knerr
Swarthmore College Computer Science Department, Swarthmore, PA, USA

Abstract

We present PeerMon, a peer-to-peer resource monitoring system for general purpose Unix local area network (LAN) systems. PeerMon is designed to monitor system resources on a single LAN, but it also could be deployed on several LANs where some inter-LAN resource sharing is supported. Its peer-to-peer design makes PeerMon a scalable and fault tolerant monitoring system for efficiently collecting system-wide resource usage information. Experiments evaluating PeerMon's performance show that it adds little additional overhead to the system and that it scales well to large-sized LANs. PeerMon was initially designed to be used by system services that provide load balancing and job placement, however, it can be easily extended to provide monitoring data for other system-wide services. We present three tools (smarterSSH, autoMPIgen, and a dynamic DNS binding system) that use PeerMon data to pick "good" nodes for job or process placement in a LAN. Tools using PeerMon data for job placement can greatly improve the performance of applications running on general purpose LANs. We present results showing application speed-ups of up to 4.6 using our tools.

1 Introduction

General purpose LANs of workstations are systems where multiple machines (nodes) are connected by a network. Each machine runs a stand-alone operating system (OS) and typically runs a network file system and may support a few other types of networked resource sharing. These types of systems are common at universities and other organizations where machines in offices and labs are connected to allow some system-wide resource sharing, but where most of a machine's resources are under the control of its local OS. Typically, these systems do not implement any kind of centralized scheduling of networked resources; resource scheduling is done locally by the OS running on the individual nodes.

In general purpose LANs multiple users can log into individual nodes and use the networked resources to run any workload including batch, interactive, sequential and parallel applications. The workload in such systems is much more dynamic and not as well controlled as in cluster systems that typically run system-wide job scheduling software that users must use. As a result, there are often large variations in system-wide resource usage and large imbalances in the use of computational resources in general purpose LANs [3].

To perform computational tasks efficiently it is often key to have some knowledge of resource availability and resource load. For example, it would be ideal to choose the node with the lowest CPU load, the largest amount of free RAM, and the fewest number of users to run a computationally intensive sequential program. For parallel applications (such as MPI) running on a network of workstations, performance is usually determined by the slowest node. If a user had a tool that could easily identify the best nodes on which to run a parallel job, avoiding heavily loaded nodes, the result could be a dramatic improvement in execution time of the application.

Because general purpose networked systems do not provide system-wide resource scheduling, it is up to users to either guess at good placement or gather current usage information on their own to make better informed job placement options. In some cases, this can require a fair amount of effort; in others, it may not be possible. For example, a system may be set up so that individual nodes cannot be specified for remote ssh. Instead, the DNS server may use a round-robin mapping of a generic name like `lab.cs.swarthmore.edu` to one of the nodes in the system. In this case, a user can end up on a heavily loaded node, her only recourse being to log out and hope for better placement when she tries again.

A network resource monitoring system that efficiently provides system-wide usage data could be used to better distribute users and program workloads across the system. This would result in more balanced resource usage across the system, better system-wide resource utilization and, thus, better average system-wide performance.

We designed PeerMon to efficiently provide system-wide resource usage information to tools that implement load balancing functions in general purpose LAN systems. Each node in the system runs a PeerMon daemon peer that periodically collects system usage statistics about its own node and sends its information about system-wide resource usage to a fixed number of peers (currently three). The peers are chosen based on heuristics designed to maintain accurate system-wide data and a high degree of P2P network connectivity while at the same time minimizing network overheads.

PeerMon's peer-to-peer design solves problems associated with more centralized client-server monitoring systems like those based on Simple Network Management Protocol (SNMP), namely the single server bottleneck and single point of failure. Because there is no central authority for system-wide resource information, there is no central server that can become a bottleneck as systems grow to larger numbers of nodes. Applications that use PeerMon data access it locally on the nodes on which they run by interacting with their local PeerMon daemon. This ensures that system-wide resource usage data are always available and can be accessed quickly through a local service on each node. Additionally, since it is not necessary that system-wide resource usage data be consistent across all peers for the data to be useful, our system is designed to avoid costly peer data synchronization and peer data recovery.

PeerMon is also fault tolerant. Each PeerMon peer is equal and provides system-wide usage information to clients on its local node. If a node fails, PeerMon daemons on other nodes just stop receiving data about the failed node, but continue to provide system-wide resource information for non-failed resources.

To demonstrate how PeerMon resource monitoring data can be used, we implemented three tools that make use of its data. The first tool, smarterSSH, uses data collected from the peer monitor process to select the best machine to ssh into. Currently, we support selecting the "best" machines based on combinations of CPU load, RAM load, and number of CPU cores. The second tool, autoMPIgen, uses PeerMon data to automatically generate MPI host files based on system-wide resource capabilities and usage. The third tool is dynamic DNS binding based on system-wide resource usage. Using data provided by the PeerMon daemon running on the DNS server, our tool sets bindings so that a single name is mapped to the current set of "best" nodes in the system. A user who remotely ssh's into `cslab.cs.swarthmore.edu` will be logged into one of the "best" machines in our LAN. The result is that we better distribute remote logins across machines in our system.

Currently PeerMon runs on the Swarthmore Computer Science Department's LAN of about 60 Linux 2.6/x86 machines. All three tools that make use of PeerMon data are available to the users of our system.

The remaining parts of the paper are organized as follows: Section 2 discusses related work; Section 3 discusses the design of PeerMon; Section 4 discusses PeerMon's current implementation, configuration, and running details; Section 5 discusses our three example tools that we designed that make use of PeerMon data; Section 6 presents performance results of PeerMon and our example tools; and Section 7 concludes and discusses future directions for our work.

2 Related Work

Our work is most closely related to other work in network management and network resource scheduling. There has been a a lot of work on network management systems that are designed to obtain usage information and manage networked resources. Most of these are centralized systems based on the Simple Network Management Protocol (SNMP) framework [9]. The framework is based on a client-server model in which a single central server periodically sends requests to clients running on each node to send back information about the node. In addition, SNMP allows the manager to send action requests to clients to initiate management operations on individual nodes. The centralized design allows for a single central authority to easily make system-wide resource management decisions; however, it also represents a single point of failure in the system and a bottleneck to scaling to large-sized networks.

To address the fault tolerance and scalability problems associated with a centralized design, there has been work in distributing network management functionality. Some work uses a hierarchical approach to network management [8, 20, 10]. In these systems, one or more top-level managers communicate with distributed mid-level managers to perform resource management activities. Because the mid-level managers are distributed over the network, these systems scale better than centralized systems. Additionally, there is typically some support for handling failure of one or more manager processes.

There have also been systems proposed using a P2P design for networked management systems [2, 18]. In particular, Panisson et al. [15] propose a modification to the SNMP protocol whereby the network contains nodes of three different roles, two types of managers as well as an agent processes.

Our work is similar in that we use a P2P design to solve the fault tolerance and scalability problems with centralized solutions. However our work differs in two fundamental ways. First, PeerMon is designed to provide system-wide resource monitoring and resource usage data collection only. It is not a network management system, but provides lower-level monitoring and data collection. Thus, its design is much less complicated than this other work and as a result, can be better optimized for its system-wide data collection task independently of how its data may be used by higher-level services. A higher-level resource management system could be implemented as a client of PeerMon data rather than being integrated into PeerMon. The second difference is that every PeerMon peer is an equal peer. The result is a purer P2P design than this other work; one that provides a more layered and flexible architecture for designing resource management systems, and one that is more fault tolerant and scalable.

Other work related to ours is in the area of resource scheduling and load balancing tools for networked systems. There has been a lot of work in this area, most focusing on cluster and grid systems [7, 13, 11, 4, 16, 17, 19].

Condor [13] and the Now/GLUnix project [7] are two examples that are designed, in part, to run on general purpose networked systems like ours. NOW/GLUnix implements a cluster abstraction on top of a network of workstations that are simultaneously being used by individual users as a general purpose LAN. GLUnix stores the global state of the network on a single master node. This state is updated by daemon processes running on each node, which periodically send their local resource usage information to the master. The data are used to support resource allocation and parallel and sequential job placement by the master.

Condor implements a job submission and scheduling system for running parallel and sequential applications on LANs, clusters, and grids. When run on general purpose LANs, Condor discovers idle nodes on which to run jobs. When a node running a Condor job is no longer idle, Condor uses process migration to move Condor jobs to other idle nodes in the system. Condor uses a centralized manager and local daemons to collect system-wide load statistics and to perform process control.

GLUnix and Condor provide much higher-level services and abstractions than our work, but both collect system-wide resource usage data on the same types of target systems. PeerMon provides only the underlying system for data collection, but uses a P2P design instead of a centralized one. PeerMon could potentially be used to provide data to higher-level system services like Condor or GLUnix.

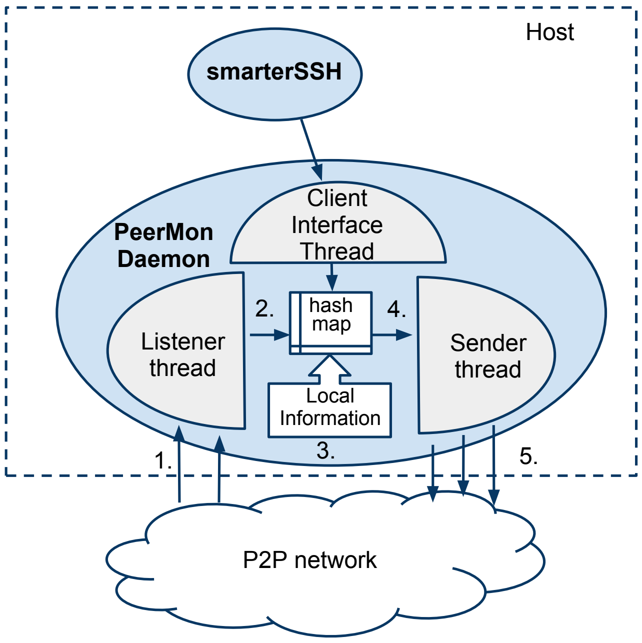


Figure 1: *PeerMon Architecture. Each host runs a PeerMon daemon . The Listener thread receives UDP messages from the P2P network (1) and updates the hashMap with the newest data (2). The Sender thread periodically wakes-up and updates the hashMap with local node resource data (3). It then selects three peers to send its hashMap data via UDP messages (4 and 5). Applications, like smarterSSH, interact with the PeerMon Client Interface thread via a TCP/IP to obtain PeerMon system-wide resource usage data.*

3 The PeerMon System

PeerMon is designed to run on a general purpose networked system where users can log into any node at any time and run any mix of parallel and sequential programs, and batch and interactive applications. The three main goals in designing PeerMon are: to efficiently provide, in real-time, system resource usage information; to scale to large-sized systems; and to be fault tolerant. The system also needs to be flexible enough to allow nodes to easily enter and leave the P2P network. Additionally, because each node continuously runs a PeerMon daemon, it is important that PeerMon uses minimal network and other system resources.

To meet these goals we chose a P2P design for PeerMon. Each node in the network runs a PeerMon daemon, which is an equal peer in the system; there is no central authority nor is there a hierarchical relationship among peers. Every node in the system provides system-wide resource usage data to its local users. Thus, users of PeerMon data need only contact their local daemon to get information about the entire system.

When a PeerMon node fails, the rest of the system continues to function; non-failed nodes continue to use PeerMon data collected from their local PeerMon daemons. Data from failed or unreachable nodes ages out of the system and will not be included as a ”best node” option by system services that use PeerMon.

Recovery from failure is easy. When a node comes up, it starts a PeerMon daemon that reconnects to the system by sending its information to three peers. Once other peers hear of the new peer, they will send it system-wide resource usage data in subsequent peer data exchanges. Our tests show that it takes on average eight rounds of message exchanges for a new peer to become fully connected into the system.

To reduce the amount of network traffic between peers, we use the observation that it is not necessary, nor is it possible, to have completely accurate system-wide resource usage information in general purpose networked systems. Even in a centralized resource monitoring system, the data do not represent an instantaneous snapshot of system-wide state [12]. PeerMon is designed so that each peer collects system-wide resource information, but individual PeerMon nodes may have slightly different information about the system. Distributed PeerMon data do not need to have the same type of consistency constraints as distributed file system and database data do. Thus, we do not need to implement expensive synchronization to support consistency of data across all peers. As long as each PeerMon peer has relatively recent resource-usage information about the system, its data is just as accurate and useful as data provided by a centralized system.

Higher-level services that use PeerMon data to implement load balancing or job-placement combine PeerMon data with accounting of their activities to make policy decisions. These higher-level services could be implemented as centralized, hierarchical or distributed independent clients of PeerMon. The constraints on higher-level service determine which PeerMon peers it would use to make policy decisions. This is no different than how such systems would use data from a centralized resource monitoring system. PeerMon, like other resource monitoring systems, does not need to account for how its data may be used by higher-level services.

3.1 System Architecture

Figure 1 shows the structure of the multi-threaded PeerMon daemon process. The Listener thread receives messages from other peers containing system-wide resource statistics. The Sender thread periodically wakes up, collects resource usage information about its local node and sends a copy of its system-wide state to three other PeerMon peers. The Client Interface thread exports the peer’s collected system-wide state to local applications that want to use PeerMon data.

IP	TS	TTL	Indegree	payload
130.52.62.123	5	7	4	(char *...)

Table 1: *Structure of a hashMap entry.*

Each PeerMon daemon stores its resource usage data in a data structure called the hashMap. The Listener and Sender threads update hashMap data in response to receiving or collecting newer resource information. The Sender and Listener threads communicate using UDP/IP sockets and the Client Interface thread communicates with applications using TCP/IP.

3.1.1 Resource Usage Data

Each PeerMon daemon stores system-wide resource usage information in a data structure called the hashMap. Table 1 shows the structure of a hashMap entry. Each entry in the hashMap contains information associated with a specific machine (node) in the network. The set of information stored includes the IP and port number of the node and PeerMon Listener thread, and the payload that contains the resource usage data from that node. Currently, PeerMon is implemented to collect and store resource usage information in the payload field, but the general structure is such that it could be modified to store other types of data.

The time to live (TTL) field approximates the age of the data. Its value is decremented each time stamp (i.e. each time the Sender thread sends information to three other peers). The Indegree field counts the number of messages that a node has received in the latest interval between two time steps. The time stamp (TS) field contains the last time stamp when the node was directly contacted from this PeerMon daemon. The TTL, Indegree, and TS are used by heuristics to select the three peers to send hashMap data to at each time step. The TS field is stored locally and is not sent to other nodes. All other hashMap data are shared with peers.

3.1.2 Sender and Listener Threads

The Sender thread periodically wakes up, collects resource statistics about its local node, adds them to its hashMap, and then selects three peers to send its hashMap data. The Listener thread is responsible for receiving hashMap entry data from other peers and updating its local hashMap with all or some of these data. The Sender thread decrements the TTL field of each entry each time it performs this operation. The TTL field approximates how old the resource data are for each node. The Listener thread compares TTL fields of entries in its hashMap and entries received from peers. If the peer data has a larger TTL value, it updates the hashMap with the peer data (i.e. this is more recent data about the node). If the current hashMap entry's TTL value is larger, it does not update its hashMap with the data from the peer (i.e. the current entry represents more recent data about the node than the data the peer sent). Currently, the TTL field's value starts at 10. Experiments run on networks of sizes 25-500 show that this value works well to ensure both recent data and high connectivity.

We chose to have the Sender and Listener threads use UDP/IP sockets to communicate to avoid TCP connection overheads each time peers wanted to communicate. As long as most UDP messages are delivered, an occasional dropped packet will not affect the quality of the data in the system. Because absolute correctness of the data cannot be guaranteed, losing an occasional packet will have little or no effect on the system. When the node receives other system-state messages, the window of time during which it missed an update about a node's state is small. If packet loss is common, then the added guarantees provided by TCP may be worth its higher communication overhead.

Because UDP is used to send hashMap data, care must be taken to ensure that loss of a single packet does not cause a Listening thread to block forever waiting for a lost message. To ensure this, the Sender thread sends several independent messages containing parts of its hashMap data to each node. Each independent message fits into a single packet so that if a packet is dropped, the Listener thread will never block trying to receive it; it just receives and handles the next independent message never knowing that it received one fewer message than a Sender thread sent.

3.1.3 Heuristics used to select Peers

To ensure that all nodes have recent usage information about all other nodes in the system, care must be taken in selecting which of three peers the Sender thread sends it hashMap data to. We developed three heuristics for selecting peers that, when used in combination, do a good job of distributing new data to all peers and of maintaining a high degree of connectivity in the P2P network. Each time the Sender thread wakes up, it applies one of the three heuristics. The heuristics are cycled through in round-robin order.

The first heuristic, named "Contact New Nodes", picks peers that are relatively new to the network. Since PeerMon nodes can enter or leave the P2P network at any time (e.g. due to node failure and restart) this heuristic ensures that new nodes in the system collect system-wide information soon after they connect the network. The heuristic picks peers with the smallest value of: $MAX\_TTL - TTL + Indegree$. The heuristics ensures that nodes with a high TTL (i.e. nodes whose information is new) and a low Indegree (nodes who have not been sent to recently) are selected. The heuristic results in new peers being quickly integrated into the system; however, its use alone can lead to P2P network partitioning.

The second heuristic, "Contact Forgotten Nodes", selects the three nodes with the lowest TTL (i.e. nodes that the present node has heard from least recently). The third heuristic, "Contact Old Friends", is designed to ensure that a node cannot become permanently isolated. It uses the TS field values to choose peers that it has not sent data to recently.

The combination of three heuristics works well to prevent network fragmentation and to allow for new nodes to quickly become fully incorporated into the system.

4 Current Implementation of PeerMon

PeerMon is implemented in C++. It runs on the Swarthmore Computer Science Department network of about 60 Linux 2.6/x86 machines. Our system has some heterogeneity in that machines have different numbers of cores (we have 2, 4 and 8 core machines), different amounts of RAM, and slightly different processors. All machines are connected by a switched 1Gbit Ethernet network.

PeerMon daemons collect local resource data for CPU load, amount of free RAM, and number of users through the Linux /proc interface on the node on which they run. PeerMon can be modified to collect and distribute other data. Currently, this would require changing PeerMon code. In the future we plan to add a programming interface that would allow users to more easily change the set of data PeerMon collects and change how it collects it.

4.1 Starting Up a PeerMon Daemon

The PeerMon executable takes several command line arguments that can be used to run and configure PeerMon in different ways. Figure 2 shows the command line options that include specifying the port number for the Listener thread, running the daemon in collector-only mode, starting with a user-defined configuration file, and specifying the number of seconds the Sender thread sleeps between collecting local information and sending its hashMap data to three peers.

When a PeerMon daemon starts-up it reads information from a config file that contains addresses of three PeerMon nodes. These are the first three nodes that the Sender thread contacts to start the exchange of system-wide resource data.

If the PeerMon daemon is started in collector-only mode it will receive resource usage information about other nodes, but sends "invalid" information about itself.

```
peermon -p portnum [-h] [-c] [-f configfile] [-n secs]
   -p  portnum:   use portnum as the listen port for peermon
   -c:            run this peermon daemon in collector-only mode
   -f conf_file:  run w/conf_file (default /etc/peermon/peermon.config)
   -n secs:       how often daemon sends its info to peers (default 20)
```

Figure 2: *Command line options to peermon daemon.*

Other nodes, upon receiving ”invalid” data, will not include the collector node’s information in data it exports to its local users. This allows a collector-only node to use PeerMon data but not make itself a candidate for other node’s use. We run PeerMon in collector-only mode on our DNS server so that other nodes will not choose it as a target for ssh or spawning application processes.

Each machine in our system is configured to start a PeerMon daemon when it starts-up. Each machine also periodically runs a cron job to detect if the PeerMon daemon is still running, and if not, re-starts it.

4.2 PeerMon Data Interface

Users (clients) of PeerMon data, such as smarterSSH, obtain PeerMon data by connecting to the Client Interface thread and sending it a message requesting its hashMap data. TCP sockets are used to connect to the Client Interface thread.

In our current implementation, the PeerMon daemon also exports its hashMap data by writing it to a file on the local file system. The PeerMon Sender thread replaces the old contents of the file with updated hashMap values each time it wakes up. Clients can access PeerMon data by opening and reading this file. There is a potential race condition between the reader and writer of this file. However, because we do not plan to support the file interface in the final version of the code, we ignore handling the unlikely event of a read/write conflict to this file (in practice we rarely see it). The file interface was our initial client interface to PeerMon before adding the Client Interface thread, and is currently used to help with debugging of our system.

Although the file interface is easier for clients to use than the TCP interface, it has two problems: the first is the potential read/write race condition to the shared file that could result in clients reading garbage or incomplete data; the second, and more serious, problem is that there is non-trivial overhead associated with re-writing the file contents each time data are collected. With the TCP interface the PeerMon daemon only exports its data when they are being used by a client.

In the future we plan to implement a higher-level programming interface for PeerMon clients that will hide the underlying TCP interface in an easier to use library.

5 Example Applications that make use of PeerMon data

The initial motivation for developing PeerMon was to design tools that could make better load balancing decisions in general purpose network systems by considering system-wide resource usage data. As a demonstration of how PeerMon data can be used for such purposes, we developed three tools: smarterSSH; autoMPIgen, and dynamic DNS binding based on resource load.

5.1 smarterSSH

smarterSSH is our tool for choosing the ”best” ssh target node based on PeerMon data. It is implemented in Python and has several command line options that allow a user to specify different criteria for ordering the ”best” node(s) and to select different runtime options.

The following are the command line options to smarterSSH:

```
-c: order nodes by CPU load
-m: order nodes by free memory
-n num: print out the best num nodes
        rather than ssh into the best
-i: verbose printing mode
```

By default, smarterSSH orders nodes based on a combination of their CPU load and amount of free RAM using the function: $\frac{freeMem}{1+CPUload}$ (1 is added to prevent division by 0).

When run with no command line options, smarterSSH connects to its local PeerMon daemon to obtain its hashMap data, sorts the data based on the combination of CPU load and free RAM, randomizes the order of equivalent nodes, and ssh’s into the top node from the sorted result. Running with command line options `-c` or `-m` sorts the data by CPU load only or free RAM only. The ordering functions use small delta values to place nodes into equivalence groups so that small differences in free RAM or CPU load are not deemed significant.

Running with command line options `[-n num]` causes smarterSSH to print out an ordered list of its top `num` nodes rather than ssh’ing into the ”best” node.

As an example, Figure 3 shows output from a run of smarterSSH with the command line options: `-c -i -n 10`. This run will order nodes by CPU load only, and will

```
host      CPU load   free RAM    cores
--------------------------------------
avocado   0.000      13068052      8
pimento   0.000      15828112      8
orange    0.000      2933896       4
cucumber  0.000      6291932       4
dill      0.000      5967724       4
ginger    0.000      3170436       4
marjoram  0.000      7049804       4
molasses  0.000      6881228       4
anise     0.030      14659024      8
perilla   0.020      5597020       4
```

Figure 3: *Example output from a run of* `smarterSSH -c -n 10 -i` *(print out the top 10 "best" nodes as ordered by CPU load). Eight of the nodes are equally good with a CPU load of 0.0. anise is ranked higher than perilla because it has 8 cores vs. 4.*

print out the top 10 nodes rather than ssh'ing into the top node. In this example there are eight "best" nodes, all with CPU load 0.0. Each time smarterSSH is invoked, it randomizes the PeerMon data so that the total ordering among equal nodes varies. This means that subsequent runs of the above command could result in a different ordering of the first eight nodes. Randomization is used so that multiple invocations of smarterSSH will distribute the load over the "best" nodes while these new ssh's have not yet had time to effect system load.

5.2 Automatic MPI host file generation

autoMPIgen is another tool that uses PeerMon data to perform load balancing in general purpose LANs. It automatically generates MPI host files by choosing the best nodes based on PeerMon's system-wide resource use data. It is written in Python and is very similar to smarterSSH. When run, autoMPIgen interacts with the local PeerMon daemons to obtain system-wide resource usage information. It has command line options to allow the user to specify how to order machines and how to configure the resulting OpenMPI [5] hostfile [1] containing the "best" machines.

The following are the command line options to autoMPIgen:

```
-n num: choose total num nodes
-f filename: specify the output file.
-c: order best nodes by CPU load only
-m: order best nodes by free RAM only
    (default is combination CPU and RAM)
-i: printout results to stdout
-p: include a node's number of CPUs
    in the generated hostfile
-cpus:  interpret the num value from
        (-n num) as number of cores
```

As an example, using the PeerMon data from Figure 3, autoMPIgen run with the command line options `-n 9 -c -p` generates the following hostfile (the 9 best hosts ordered by CPU load and including the core count in the hostfile ("slots=n")):

```
avocado slots=8
pimento slots=8
orange slots=4
cucumber slots=4
dill slots=4
ginger slots=4
marjoram slots=4
molasses slots=4
anise slots=8
```

A run adding the additional command line argument `-cpus` interprets the `-n 9` value to mean CPUs rather than nodes, and generates the following hostfile (best machines with at least a total of 9 cores):

```
avocado slots=8
pimento slots=8
```

5.3 Dynamic DNS

Our third example of using PeerMon data is to incorporate it into dynamic domain name server (DNS) binding. [1] This allows a virtual host name to be mapped to one of the set of "best" physical nodes where "best" nodes are selected based on system-wide load.

Using PeerMon data to select a set of "best" nodes has several benefits over BIND's support for load distribution that selects a host to bind to using either round-robin or random selection from a fixed set of possible hosts. Our solution allows for the "best" host to be selected based on current system resource load, thus adapting to dynamic changes in system resource usage and resulting in better load distribution. Our solution is also resilient to nodes being unreachable due to temporary network partitioning, node failure, or to deliberate shut-down of nodes in order to save on energy consumption during times of low use. In BIND, if the selected host is not reachable, then ssh hangs. Using our system, unreachable or failed nodes will not be included in the set of "best" targets. When a node is reachable again, PeerMon will discover it and the node may make its way back into the set of "best" targets.

Adding support for dynamic DNS binding using PeerMon data is fairly easy if you have control over your own domain name server. In our department we run our own DNS server and control both the name-to-address and the reverse address-to-name mappings for our sub-domain (`cs.swarthmore.edu`.) The following is a summary of the steps we took to add support for dynamic binding to nodes chosen using PeerMon data:

1. Run PeerMon on our domain name server in collector-only mode.

2. Periodically (currently once per minute) update the resource records for our sub-domain so that one hostname (`cslab.cs.swarthmore.edu`) has n address records associated with it (we have n set to 5). These 5 machines are selected using data from the local PeerMon daemon.

3. Use the round-robin feature of BIND 9 to rotate through the 5 addresses when queries for `cslab.cs.swarthmore.edu` are made

The first step requires that PeerMon is running on potential target machines in our system and on the DNS server. We run PeerMon daemons on most of our machines (we exclude most servers and a few other machines that are designated for special use). The DNS server runs the PeerMon daemon in collector-only mode, which will exclude it from being a target of smarterSSH, autoMPIgen, or any other tool using PeerMon.

The second and third step for adding PeerMon data into the DNS records require that we first enable the dynamic update feature of BIND 9 by adding an ”allow-update” sub-statement to our DNS zone configuration file:

```
zone "cs.swarthmore.edu" {
 type master;
 file "cs.db";
 allow-update {127.0.0.1;130.58.68.10;};
};
```

Next, a script to update DNS records based on PeerMon data is added as a cron job that runs once per minute. When run, the script first contacts its local PeerMon daemon to obtain system-wide resource usage data to determine the 5 ”best” machines. For example, suppose these are currently the five best machines based on PeerMon data:

```
130.58.68.41
130.58.68.70
130.58.68.162
130.58.68.74
130.58.68.148
```

The script next generates a file of commands for nsupdate (part of the BIND 9 software), deleting the old records first, and then adding new A records (an example is shown in part (a) of Figure 4.) As a last step, the script runs ”nsupdate” on the generated file to change the DNS records (the results on the example are shown in part (b) of Figure 4):

The round-robin feature of BIND will map `cslab.cs.swarthmore.edu` to one of these 5 ”best” nodes until the cron job runs again to change the mapping to a possibly new set of the 5 ”best” nodes.

Our implementation led to a couple difficulties that we had to solve. First, every PeerMon daemon must have the same ssh host key. Otherwise, when users repeatedly ssh to `cslab`, each time getting a different machine from the PeerMon list, ssh would warn them that the host identification has changed for `cslab.cs.swarthmore.edu`. We solve this problem by giving all machines running PeerMon the same ssh host key, and distributing an ssh_known_hosts2 file that reflects this fact.

The second difficulty had to do with editing DNS data files. Because we are using dynamic DNS, a program running on our DNS server updates our domain data files every few minutes. A serial number in the domain data file is used to signal the change in the zone’s data, which means that the serial number for the zone data is being changed with each dynamic update. This poses no problem until we need to manually edit the domain data file (e.g., to add a new name-to-address mapping). To solve this problem, our system administrators must first ”freeze” the zone, then make manual editing changes, and then ”unfreeze” the zone. BIND 9’s `rndc` command makes this fairly easy:

```
$ sudo rndc freeze
  (edit the data files here, being
   sure to update the serial number)
$ sudo rndc thaw
```

Once set up, students and faculty can ssh into `cslab.cs.swarthmore.edu` and be automatically logged into a machine with the lowest load in the system. Because we update the mappings every minute, and because remote ssh is not a frequent system activity, the result will be good distribution of remote ssh’s accross nodes in our system. Another benefit is that users do not need to remember specific machine names to log into our system; they simply ssh into `cslab.cs.swarthmore.edu` and are placed on a good machine.

By using PeerMon data, machines with high loads, machines that are unreachable, or machines that have been shutdown will be excluded from possible hosts. This not only means that there is better load balancing using PeerMon data, but that our approach to dynamic DNS binding is resiliant to network partitioning and node failures. No longer do users log in to machines that are already heavily loaded, or try to log into a machine, only to see their ssh process timeout. A benefit for our system administrators is less editing of the DNS data files. If a machine is taken out for service, it is automatically (within a minute or two) removed from the pool of best-available machines, requiring no manual editing of the DNS data files. When a machine is restarted, it will quickly be added back into the PeerMon network

```
(a) example generated file contents:
    --------------------------------
update delete cslab.cs.swarthmore.edu.
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.41
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.70
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.162
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.74
update add cslab.cs.swarthmore.edu. 30 IN A 130.58.68.148
<a blank line is necessary here>

(b) results after executing nsupdate:
    ---------------------------------
$ host cslab.cs.swarthmore.edu
cslab.cs.swarthmore.edu has address 130.58.68.70
cslab.cs.swarthmore.edu has address 130.58.68.74
cslab.cs.swarthmore.edu has address 130.58.68.148
cslab.cs.swarthmore.edu has address 130.58.68.162
cslab.cs.swarthmore.edu has address 130.58.68.41
```

Figure 4: *Dynamic DNS impementation details: (a) an example generated file containing update command for nsupdate; and (b) output from running* `host` *after the script runs* `nsupdate`*.*

and will automatically be a candidate target for dynamic DNS binding.

6 Performance Results

We present results mesuring the performance of PeerMon in terms of its overheads, the degree of P2P network connectivity, the age of system-wide resource data, and its scalability to larger networks. We also present results using the tools we developed that make use of PeerMon data to perform load balancing in our network. Our results show that these tools significantly improve the performance of application programs running on our system.

6.1 PeerMon P2P Network Conectivity and Age of Resouce Usage Data

To evaluate the connectivity of PeerMon peers, we simulated a network of 500 nodes by running 10 instances of a PeerMon daemon process on each of 50 machines in our lab. Each daemon was started with a time stamp value of 5 seconds [2] (how frequently the Sender thread wakes-up and collects and distributes usage data).

P2P network connectivity is computed as the average number of nodes contained in each daemon's hashMap divided by the total number of nodes in the network. A connectivity of 1 means that every node in the network has current information about every other node in the network. For networks up to size 500, we consistently saw connectivity values above 0.99.

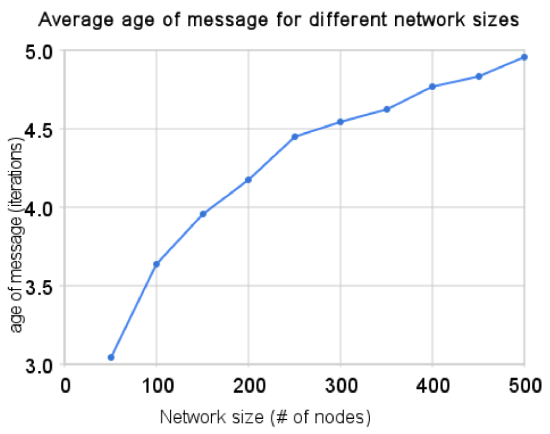
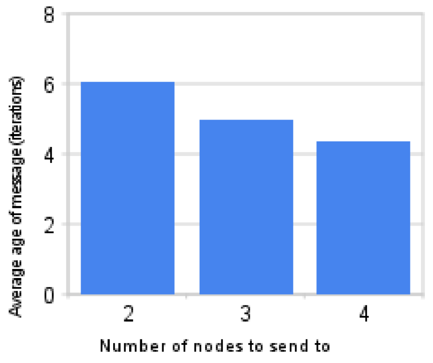


Figure 5: *Average message age across all nodes in the network for various network sizes.*

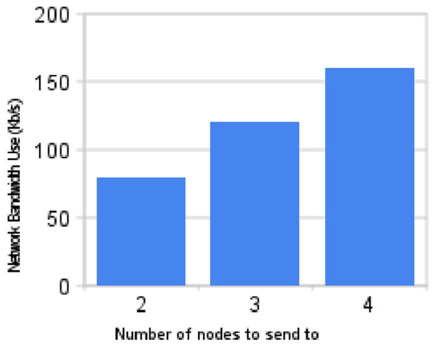
In addition to connectivity, we computed the average age of hashMap data for different sizes of networks. Figure 5 shows the results. For a network of size 50, the average age of data is about 3 iterations (roughly 15 seconds old). The average message age increases with the size of the network. For a network size of 500, the average age of a message is about 5 iterations (roughly 25 seconds old).

Additionally, we ran experiements to help us determine a good value for the number of peers that the Sender thread should send to each time it wakes-up. We ran experiements of different numbers of send-to peers on a network of 500 nodes. The results, in Figure 6(a), show that that as the number of peers increases (x-axis) the average message age decreases (y-axis). However, Figure 6(b) shows that as the number of peers increase, the PeerMon network load increases linearly.

Picking a good number of peers to send to each time step involves acheiving a good balance between maintaining newer data and good P2P network connectivity and maintaining low messaging overheads. Our results indicate that a send-to value of 2 is too small, resulting in older resouce usage data and potentially less than full connectivity. A send-to value of 4 results in an average data age of about 22 seconds with 100% connectivity; however, nearly 150 Kb/s of data are sent from each node in the network. Based on our results, we chose 3 peers as producing a good balance between achieving low data age (about 25 seconds on average), high connectivity (around 99.5%), and moderate network bandwidth usage (about 120 Kb/s).



(a) Average Message Age



(b) Network Bandwidth Use

Figure 6: *Average message age (a) and bandwidth used (in Kb/s) (b) on each node for different send-to values on a network of 500 nodes.*

6.2 PeerMon Scalability

To evaluate how well PeerMon scales to larger-sized systems, we ran multiple instances of PeerMon on each of 55 machines in our system to simulate systems of larger

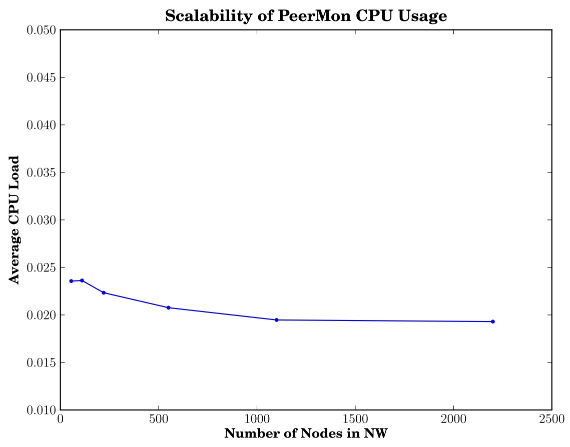


Figure 7: *The average additional CPU load per PeerMon host for different sized networks (Numbers of Nodes). The results show a basically fixed-size per-node CPU load as the PeerMon network increases.*

sizes. We ran experiements with 1, 2, 4, 10, 20, and 40 PeerMon daemons per real machine to simulate LANs of size 55 to 2,200 nodes. For these experiments we used the default 20 second rate at which the Sender thread sends to three of its peers. We ran a script on our monitoring server to periodically get MRTG [14] data to obtain a trace of five minute averages of network, memory and CPU load for every machine in our network. In order to ensure that our results for different sized systems were comparable, we ran experiments over a weekend when our system was mostly idle so that PeerMon was the primary load on the system. The data collected from each of the physical 55 machines in our network were divided by the number of PeerMon daemons running per host to obtain the per-node values for the larger systems that we simulated.

Figure 7 shows CPU load per PeerMon node and Figure 8 shows the amount of free RAM per PeerMon node for different sized networks. Both per-node CPU load and per-node RAM use stay relatively fixed as the network size increases. As the system-size grows, each PeerMon node has a larger hashMap data structure. However, the amount of memory storage and CPU processing that this data structure requires is so small that the overheads for a network of 2,200 nodes are equivalent to overheads for a network of 55 nodes. These results show that neither RAM nor CPU use will limit PeerMon's scalability to larger sized LANs.

Figure 9 shows the number of bytes sent and received per second per PeerMon node for different sized networks (from 55 to 2,200 nodes). The amount of network traffic remains very small as the size of the network grows. On the 2,200 node PeerMon network each

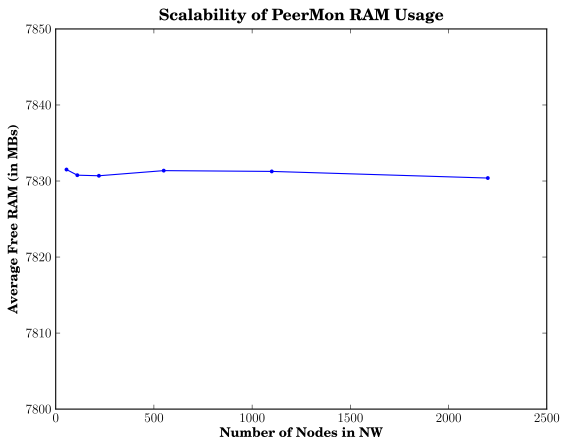


Figure 8: *The Amount of Free RAM (in MB) per PeerMon host for different sized networks. These data show that PeerMon uses little RAM space, and that the amount it uses per node stays fixed as the size of the network grows.*

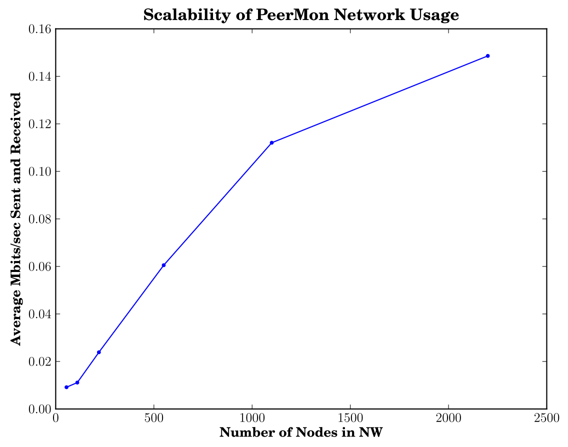


Figure 9: *The average Network load per PeerMon host for different sized networks. The data are the average Mbits/second sent and recieved per node. The data show that although there is a slight increase in network bandwidth used per node as the PeerMon network size increases, the amount used per node is still a small fraction of the total bandwith available to the node.*

PeerMon daemon uses less than 0.16 Mbits/second on its 1 Gbit connection. However, there is an increase in the amount of data each PeerMon daemon sends to its three peers as the network grows (the number of peers sent to by each PeerMon daemon is constant, but the size of each message grows with the number of nodes). On a 55 node network, each PeerMon deamon's hashMap has at most 55 entries. On a 2,200 node network, each hashMap can contain up to 2,200 entries. Each time the Sender thread wakes up and sends its hashMap contents to three peers, the total number of bytes sent to each peer grows with the size of the network.

Even for a network with 2,200 nodes, our results show that PeerMon adds very little network overhead and that its network use scales well to the types of systems for which it was designed. However, the data show some added network costs as the size of the network grows. The decision for each PeerMon daemon to send its full HashMap contents works well for the systems we are targeting, but it could become a bottleneck if PeerMon were to be deployed on a system with tens or hundreds of thousands of nodes. In this case, its design may need to be changed so that each peer exchanges only partial hashMap contents.

Our results show that PeerMon scales well to large-sized systems of the type we are targeting. It adds only negligable amounts of network, RAM, and CPU load to the system.

6.3 Results Using smarterSSH and autoMPIgen on Application Workloads

The initial motivation for developing PeerMon was to implement tools that could distribute user and program load in general purpose networked systems. Therefore, as a way to evaluate this use of PeerMon data, we ran experiments using smarterSSH and autoMPIgen to select the best nodes on which to run sequential and parallel MPI applications.

The experiments were run during a time when our system was heavily used so that there was variation in system-wide resource usage. For some experiments we additionally ran artificial workloads on some nodes to ensure more variation in resource usage across nodes. For these experiments, we needed to ensure some variation in resource usage, because if all nodes are basically equal, a randomly chosen node will be just as good as one chosen based on PeerMon's system-wide resouce usage data.

We evaluated the results of running different applications in the network using smarterSSH and autoMPIgen to select the nodes on which to run the applicaiton. We found significant improvements in application runtime when using our tools. The results were consistent across a broad range of tests.

For each experiment we compared runs of a benchmark program using smarterSSH or autoMPIgen to pick the ”best” node(s) to runs of the benchmark on randomly selected nodes (representing no use of PeerMon data). For the smarterSSH runs, we tested all three node ordering criteria (CPU only, RAM load only, and both). We ran each benchmark 200 times, doing 50 tests of random selection and 50 tests of each of the three smarterSSH ordering criteria. We interleaved the runs of each test so that the results would be equally affected by changes in system load over the course of the experiement.

Our first benchmark is a memory intensive sequential program that allocates a 2.8 GB array of integers and then iterates over the array elements ten times, modifying each array element as it goes. By reading and writing array values with each iteration, we ensure that if the entire array does not fit in available RAM, the application will trigger swapping on the node, which will significantly increase its total execution time.

The ”Memory” column in Table 2 lists speedup values of using smarterSSH over randomly chosen nodes. The results show that the run time using smarterSSH with CPU load ordering is not significantly different from random (speedup value of 0.87) [3] However, the two smarterSSH runs that use RAM load to select the ”best” node perform significantly better than randomly selected nodes (speedup values of 4.62). The speedup value of 0.87 for CPU, although not significantly different than random, does show that picking nodes based on CPU load alone for this benchmark will not necessarily result in good choices. Since this is a memory intensive benchmark, it makes sense to choose nodes based on their RAM load.

Our second experiment uses a primarily CPU intensive benchmark consisting of an openMP implementation of Conway’s Game of Life (GOL) [6]. The benchmark program runs on a single machine. It consists of a two threaded process that computes the Game of Life on a 512x512 grid for 1000 iterations. The column labeled ”OpenMP GOL” in Table 2 presents the speedup values obtained using smarterSSH vs randomly selecting a node. Our results show that speedup is significant for all three smarterSSH runs, with the combination ordering criterion performing slightly better than the others (speedup of 2.29).

The final benchmark program is an OpenMPI implementation of the Game of Life [4]. We ran the benchmark on a 10000x10000 grid for 30 iterations. The program consists of 8 MPI processes that are distributed across 8 different nodes in our system. The implementation proceeds in rounds where processes must synchronize with the others before starting the next round. As a result, the runtime is determined by the slowest process. autoMPIgen was used to automatically generate the MPI hostfiles for the runs using PeerMon data.

Node	**Benchmark**		
Ordering	Memory	OpenMP GOL	MPI GOL
CPU	*0.87*	1.63	1.27
Memory	**4.62**	2.19	1.78
Both	**4.62**	**2.29**	**1.83**

Table 2: *Speedup over random selection of machines using each heuristic on all three of the benchmarks. Cursive entries are not significantly different from random selection.*

For these experiments we ran a CPU intensive program on 9 of the 50 nodes to create imbalances in CPU load across machines in our system (18% of the machines in our network have a high CPU load). Using randomly selected nodes, there is a 85.7% chance that each trial would include one of the nine machines running our CPU intensive program. For the autoMPIgen runs, these 9 nodes should not be selected.

The speedup values are shown in the ”MPI GOL” column in Table 2. The results show autoMPIgen runs performing significantly better than random node selection. Ordering nodes based on both CPU load and RAM load results in the best performance (speedup of 1.83).

Our benchmark tests show that using PeerMon data to select good nodes based on CPU load and RAM load results in applications performing significantly better than when run on randomly selected nodes. In the worst case, ordering nodes by CPU load does not perform significantly worse than random. A knowledgeable user should be able to predict which ordering criterion is most useful for her program based on whether the program is more CPU-intensive or more memory-intensive. However, our results demonstrate that for all the benchmarks ordering nodes using the combination of CPU load and RAM load works best. This is likely due to the fact that all programs require a certain amount of both CPU time and RAM space to execute efficiently. Based on these performance results, we use a combination of CPU and RAM load as the default ordering criteria in smarterSSH and autoMPIgen.

7 Conclusions

Our results show that PeerMon is a low overhead system that quickly provides accurate system-wide resource usage data on general purpose LAN systems. Its peer-to-peer design scales well to large sized systems and is fault tolerant. Our example applications that use PeerMon data (smarterSSH, autoMPIgen, and dynamic DNS binding based on system load) demonstrate that PeerMon data can be very useful for implementing load balancing applications for systems that do not have centralized control of resource scheduling. Our benchmark studies show significant improvement in application performance using PeerMon data to make good choices about process placement in the system. PeerMon provides a system-wide data collection framework that can be used by higher-level tools that implement management, scheduling or other monitoring activities.

Future directions for our work include: investigating, collecting, and using other system-wide statistics in PeerMon; investigating scalability and security issues associated with supporting PeerMon running on multiple LANs; and further investigating ways in which PeerMon data can be used to improve individual application performance in general purpose LANs. Additionally, we plan to implement an interface to PeerMon clients that is easier to program than the current TCP interface. Our current plan is to implement a library interface that would hide the low-level TCP socket interface. We also plan to implement better support for extensibility by adding an interface to allow users to more easily change the set of system resources that are monitored by PeerMon.

Acknowledgments

We thank Tomasz Gebarowski for his OpenMPI implementation of the Game of Life that we used to benchmark our system.

References

[1] ALBITZ, P., AND LIU, C. DNS and BIND, 4th Edition. O'REILLY, 2001.

[2] ANDROUTSELLIS-THEOTOKIS, S., AND SPINELLIS, D. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv. 36*, 4 (2004), 335–371.

[3] ARPACI, R. H., DUSSEAU, A. C., VAHDAT, A. M., LIU, L. T., ANDERSON, T. E., AND PATTERSON, D. A. The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proc.ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (1995), pp. 267–278.

[4] CLUSTER RESOURCES. MOAB Workload Manager. www.clusterresources.com/products/moab-cluster-suite.php.

[5] GABRIEL, E., FAGG, G. E., BOSILCA, G., ANGSKUN, T., DONGARRA, J. J., SQUYRES, J. M., SAHAY, V., KAMBADUR, P., BARRETT, B., LUMSDAINE, A., CASTAIN, R. H., DANIEL, D. J., GRAHAM, R. L., AND WOODALL, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[6] GARDNER, M. Mathematical games: The fantastic combinations of John Conway's new solitaire game life. *Scientific American 223* (1970), 120–123.

[7] GHORMLEY, D. P., PETROU, D., RODRIGUES, S. H., VAHDAT, A. M., AND ANDERSON, T. E. Glunix: a global layer unix for a network of workstations. *Software Practice and Experience 28* (1998), 929–962.

[8] GOLDSZMIDT, AND YEMINI. Distributed management by delegation. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems* (Washington, DC, USA, 1995), IEEE Computer Society, p. 333.

[9] HARRINGTON, D., PRESUHN, R., AND WIJNEN, B. An architecture for describing simple network management protocol (SNMP) management frameworks. RFC Editor, 2002.

[10] JETTE, M., AND GRONDONA, M. SLURM: Simple linux utility for resource management. In *ClusterWorld Conference and Expo* (2003).

[11] KHAN, M. S. A survey of open source cluster management systems. Linux.com (www.linux.com/archive/feed/57073), 2006.

[12] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[13] LIVNY, M., BASNEY, J., RAMAN, R., AND TANNENBAUM, T. Mechanisms for high throughput computing. In *Proceedings of the 21st SPEEDUP Workshop: Distributed Computing* (1997), vol. 11(1).

[14] OETIKER, T. MRTG Multi Router Traffic Grapher. http://oss.oetiker.ch/mrtg/.

[15] PANISSON, A., DA ROSA, D. M., MELCHIORS, C., GRANVILLE, L. Z., ALMEIDA, M. J. B., AND TAROUCO, L. M. R. Designing the architecture of p2p-based network management systems. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 69–75.

[16] PBS WORKS. OpenPBS. http://www.pbsworks.com/.

[17] PLATFORM COMPUTING INC. Platform LSF. www.platform.com/workload-management/high-performance-computing.

[18] SANTANA, R. L., GOMES, D. G., DE SOUZA, J. N., ANDRADE, R. M. C., DUARTE, JR., E. P., GRANVILLE, L. Z., AND PIRMEZ, L. Improving network management with mobile agents in peer-to-peer networks. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing* (New York, NY, USA, 2008), ACM, pp. 1874–1875.

[19] SKOVIRA, S. K. M. R. P. M. D. B. J. F. Workload management with loadleveler. IBM RedBooks (www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf), 2001.

[20] SUBRAMANYAN, R., MIGUEL-ALONSO, J., AND FORTES, J. A. B. A scalable SNMP-based distibuted monitoring system for heterogeneous network computing. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)* (Washington, DC, USA, 2000), IEEE Computer Society, p. 14.

Notes

[1] autoMPIgen currently generates OpenMPI hostfiles, but could be easily changed to output hostfile formats for any MPI implementation.

[2] We think 5 seconds is too frequent for normal deployment, but it allowed us to run our experiments more quickly

[3] Significance testing was done using the Mann-Whitney U test.

[4] http://myitcorner.com/?page_id=2, by Tomasz Gebarowski

Keeping Track of 70,000+ Servers: The Akamai Query System

Jeff Cohen
Akamai Technologies
Cambridge, MA 02142
jecohen@akamai.com

Thomas Repantis
Akamai Technologies
Cambridge, MA 02142
trepanti@akamai.com

Sean McDermott
Akamai Technologies
Cambridge, MA 02142
sean@akamai.com

Scott Smith
Formerly of Akamai Technologies
Cambridge, MA 02142
scott@clustrix.com

Joel Wein
Akamai Technologies
Cambridge, MA 02142
jwein@akamai.com

Abstract

The Akamai platform is a network of over 73,000 servers supporting numerous web infrastructure services including the distribution of static and dynamic HTTP content, delivery of live and on-demand streaming media, high-availability storage, accelerated web applications, and intelligent routing. The maintenance of such a network requires significant monitoring infrastructure to enable detailed understanding of its state at all times. For that purpose, Akamai has developed and uses *Query*, a distributed monitoring system in which all Akamai machines participate. Query collects data at the edges of the Internet and aggregates it at several hundred places to be used to answer SQL queries about the state of the Akamai network. We explain the design of Query, outline some of its critical features, discuss who some of its users are and what Query allows them to do, and explain how Query scales to meet demand as the Akamai network grows.

1 Introduction

Akamai's edge network is a distributed computing platform with over 73,000 servers in 70 countries in about 1,000 autonomous systems, which on any given day may handle upwards of 20% of Internet traffic. Akamai provides multiple services including the delivery of static and dynamic HTTP content and live and on-demand media streams, reliable storage, Web and IP application acceleration, and DNS services; see [15] for a recent overview. Each Akamai server runs multiple applications, constructed out of multiple components, and potentially participates in providing more than one of these services. Thus, Akamai's edge platform consists of over 1 million distributed software components.

The Akamai network supports customer businesses that run twenty-four hours a day, seven days a week. In many cases outages of even a short period of time can cause substantial business impact. The need for reliable real-time monitoring of the state of our network, therefore, is critical.

Query is a near real-time monitoring system, developed in-house, that monitors the Akamai network to provide up-to-date information about its state. It is used by automated applications to detect problems and measure performance over time, by software engineers to ensure their systems are behaving properly in the field, by operations staff to troubleshoot problems and ensure that the network is properly configured, and by services that provide data to customers. Information from Query is provided through a SQL interface, allowing users a familiar, precise way of specifying the information they need.

The Akamai network is divided into several thousand *clusters* all over the world at the edges of the Internet. It is in those clusters that Query begins collecting data. Every Akamai machine runs Query, and any software component on any machine can send data to the local Query instance to be published into database tables. Some subset of the machines in each cluster are designated as *Cluster Proxies* who also have the job of collecting all the data from their respective clusters. Each Cluster Proxy takes all the tables it receives from machines in its cluster and combines them into larger tables.

Query is partly distributed and partly centralized. The collection of data in thousands of clusters all over the world is fully distributed, but that data need to be aggregated to allow the issuing of SQL queries about the entire Akamai network. A set of a few hundred machines, called *Top-Level Aggregators (TLAs)* collects data from the cluster proxies and combines data from all the clusters into larger tables. Because it takes all the resources available to most TLAs just to talk to all those clusters and combine their data, TLAs don't have enough processing time left to also answer queries. Therefore they send their aggregated tables to *SQL parsers* that actually receive queries and compute their answers.

Several types of users make requests to Query. Human users, including software engineers and operations staff, issue queries to understand the state of the Akamai network. This is particularly important for detecting and responding to problems quickly. This monitoring and diagnosing is facilitated by the fact that Query provides aggregated data in the form of tables that can be accessed using a familiar SQL interface. This interface enables users to easily combine data from multiple real-time data sources, as well as statically generated configuration data, without the need to log in to individual machines. For example, by issuing a query such as the one below, a user can see processes on machines with role “dns” that are using more than 75% of system memory for their RSS:

```
SELECT sys.ip ip, procname, rss, pid
FROM   sys, processes
WHERE  sys.ip = processes.ip
  AND  (rss*100)/sys.memtotal > 75
  AND  sys.ip in
       (SELECT ip
        FROM machinerole
        WHERE role='dns');
```

Numerous automated applications issue queries as well. For example, Akamai’s alert system is an important tool for detecting problems and fixing them before they affect customers. It issues queries to detect each of several thousand conditions that indicate problems, then alerts staff in the Network Operations Control Center whenever those conditions are present.

A third group of users is customer-facing applications. For example, EdgeControl [3], the Akamai customer portal, provides graphs of usage to each customer. The data presented fall under two categories. The most reliable usage data are collected from detailed logs on the machines and displayed precisely. Query, however, can report results faster than the logs can be processed, but with less than perfect reliability. We display to customers the most recent data based on results from Query, and the most accurate data based on log analysis. Similarly, graphs such as the ones that are available to the general public on the Akamai website [22] depend on data collected from Query. We will discuss how each of the groups mentioned above uses Query and the benefits each gains from it.

The rest of this paper is structured as follows: Section 2 talks about the goals of Query’s design. Section 3 talks about the architecture of Query that achieves these goals. Section 4 explains several of Query’s features that are most important to users, and Section 5 details who some of those users are and how they use Query. In Section 6, we present techniques that have allowed Query to scale as the company has grown far beyond its size when Query was first written. Along with those techniques we use a number of other techniques to manage Query and make sure that it is provisioned and configured as needed, detailed in Section 7. In any large deployed network, failures are bound to occur, so we explain how Query handles them in Section 8. Finally, we compare query against related systems in Section 9, before concluding in Section 10.

2 Design Goals

Query is designed with a number of goals in mind. Occasionally, those goals conflict, providing us with difficult tradeoffs. We describe those goals and some resulting tradeoffs.

2.1 Goals

- **Reliability:** Query should always be available to answer requests.
- **Scalability:** Query should continue to stand as the load doubles several times over.
- **Data latency:** When data are published at the edge, they should appear in the answers to queries promptly.
- **Query latency:** When a user issues a query, an answer should come back quickly.
- **Completeness:** All published data should be available. Query results should be based only on complete tables.
- **Consistency:** When data are published, they should eventually be available everywhere. Requests served by distinct machines should have similar answers.
- **Synchronization:** All data available on a machine should be up-to-date as of about the same time, so that all tables from that machine reflect the state at one moment as closely as possible.
- **Fault tolerance:** When a machine fails or a connection goes down, the system should still be available to serve requests.
- **Fault quarantining:** A fault in one place should stay in that place instead of spreading.

2.2 Tradeoffs

Some of the aforementioned goals sometimes conflict. Here we describe some of the more interesting tradeoffs we face in the design of Query.

2.2.1 Data Latency and Completeness

To have complete data, Query must wait for every machine to send its contributions to every table before putting each table together. To have low data latency, Query must put its tables together quickly, waiting for as few things as possible. We achieve a balance between the two by providing a relaxed notion of best effort completeness, which will be discussed in Section 4.2.

2.2.2 Fault Tolerance and Completeness

Fault tolerance requires Query to move on and work around machines that fail. Completeness requires it to find a way to obtain their data. We strike a balance between the two with the same relaxed notion of completeness we describe in Section 4.2.

2.2.3 Fault Tolerance and Quarantining

A desire for fault tolerance suggests that when a machine fails, we should move requests to it to another machine. A desire for limiting the scope of faults suggests that, because a request could consume a large number of resources and take down a machine, we should not move requests that fail to another machine. We achieve a balance by having sets of a few equivalent machines called *aggregator sets* among which requests can move. A bad request may take down two or three machines in one aggregator set, but it will not take down the hundreds of aggregators system-wide or any machines that serve customer data. We are also very careful with aggregator sets used for critical data so that they do not get requests that consume more resources than they can afford. We describe aggregator sets in more detail in Section 3.5.

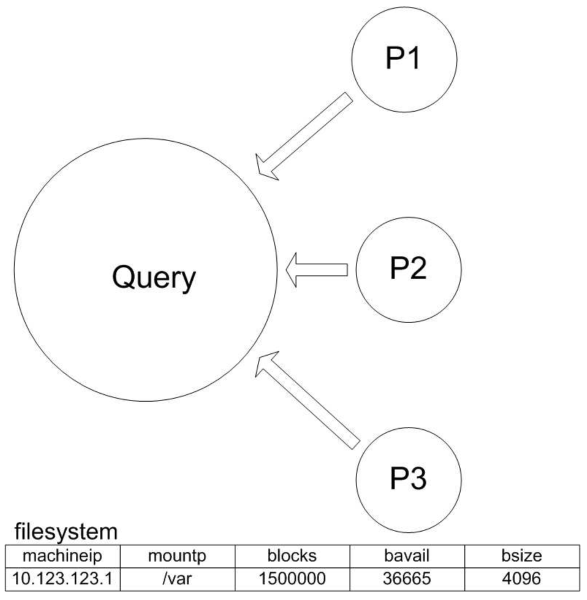
3 Architecture

We explain the architecture of Query by tracing the path data take from the time they are published to the time users see them affect the answers to queries.

3.1 Query at the Edge

Every machine on the Akamai platform runs an instance of Query. That instance listens for communications from processes on the same machine. Any process may open a connection to Query, after which point it is required to send Query a list of tables it wants to publish. Query does not start collecting these tables immediately, however. Some of them are very rarely used, and collecting them all preemptively would be a waste of resources. Instead, each Query instance maintains a list of tables that have been requested from it and requests from each process on the machine only those tables it needs.

Every once in a while (once a minute or two, depending on the machine configuration), every process is obligated to send Query a copy of all tables that process has claimed to be publishing that Query has requested. At the same frequency, but offset by several seconds, Query combines the tables being published by all processes on that machine. We call the set of tables a machine combines together a *generation*. The reason for the offset is so that the other processes have time to publish the data before Query consumes them. When Query prepares its generation, all the data were collected within a relatively short time span (several seconds), so the data provided by any individual edge machine come close to reflecting its state at one moment.

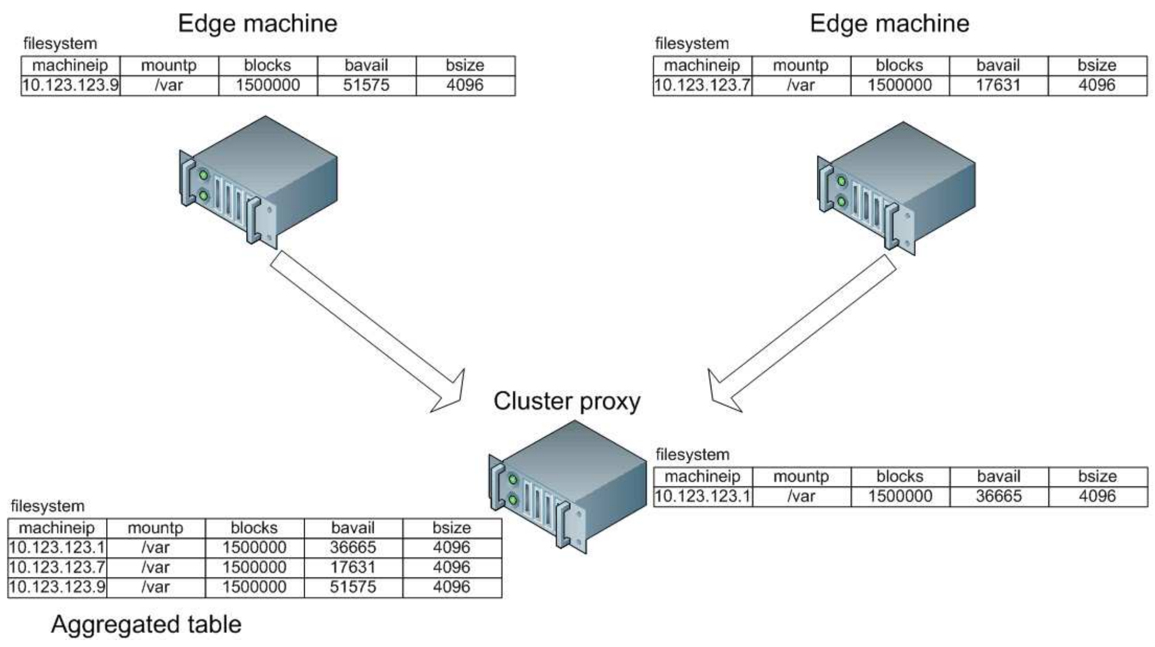


machineip	mountp	blocks	bavail	bsize
10.123.123.1	/var	1500000	36665	4096

Figure 1: Query on an edge machine. Three processes, $P1$, $P2$, and $P3$, are shown publishing into Query, as is one example table.

There are several interfaces to Query used to publish on the edge machines. The most basic is a programmatic C interface that handles all the communication with Query. Wrappers around that interface exist in several other languages. Users also have the option of writing a file containing the values in their table in a text-based format. A daemon on the machines reads those files periodically and publishes their contents into Query. Finally, a separate software component enables Query to collect data published by SNMP-enabled devices, such as routers or filers.

A picture of an edge machine is shown in Figure 1. A single Query process and three publishing processes are shown, as is one row of an example table. That row describes information about one mount point on the ma-



filesystem (Edge machine)

machineip	mountp	blocks	bavail	bsize
10.123.123.9	/var	1500000	51575	4096

filesystem (Edge machine)

machineip	mountp	blocks	bavail	bsize
10.123.123.7	/var	1500000	17631	4096

filesystem (Cluster proxy)

machineip	mountp	blocks	bavail	bsize
10.123.123.1	/var	1500000	36665	4096

filesystem (Aggregated table)

machineip	mountp	blocks	bavail	bsize
10.123.123.1	/var	1500000	36665	4096
10.123.123.7	/var	1500000	17631	4096
10.123.123.9	/var	1500000	51575	4096

Figure 2: Query in a cluster. Two edge machines and a Cluster Proxy are publishing tables, which the Cluster Proxy aggregates. One example table is shown.

chine. In reality, that table has multiple rows per edge machine. Also, there are really several times as many publishing processes and hundreds of different tables available on each machine.

3.2 Cluster Proxies

The collection of data by Query is hierarchical. The Akamai network is divided into clusters all over the world, each located within a single data center. Within each cluster, some number of machines are designated *Cluster Proxies* and have the job of collecting data from all machines in the cluster. Each cluster is small, having at most a few dozen machines, so data collection does not incur a high overhead.

Each Cluster Proxy collects requests from the next level down the hierarchy and requests tables from each machine in its cluster. Every time any Query process collects a generation, it sends each Cluster Proxy a copy of all tables the Cluster Proxy is requesting. Any time Query sends a generation of tables from one machine to another, it sends it in an efficient encoded format to save bandwidth. Offset by several seconds from that process, the Cluster Proxies collect their own generations containing all the data from their entire respective Clusters.

The Cluster Proxies also serve as edge machines, so they also publish their own data, which they combine with the data from other machines in the cluster when making their generations.

A picture of Query in a cluster is shown in Figure 2. Only two edge machines, a cluster proxy, and one table are shown. In practice, a cluster would have up to dozens of machines, several cluster proxies, and hundreds of tables. The rows from all the edge machines are combined at the Cluster Proxy.

3.3 Aggregators

The next level in the hierarchy is the *Top-Level Aggregators* (TLAs). Each TLA has a complete view of the network, because it talks to a Cluster Proxy in each cluster. The job of a TLA is to collect generations from the Cluster Proxies, aggregate together global generations of all the tables from everywhere, and provide those global generations to other machines that will use them to answer SQL queries. We don't have the TLAs answer queries because it takes all the resources they have just to aggregate the generations.

TLAs collect generations of data from all Cluster Proxies in much the same fashion that Cluster Proxies do from machines in their clusters. TLAs collect their generations once every one or two minutes. Because we are interested in data about all Akamai machines, including TLAs, each TLA also publishes into Query. It collects its own information and sends it to all other interested TLAs. Each generation a TLA makes can include, in addition to data from the Cluster Proxies, data from itself and other TLAs.

3.4 SQL Parsers

A *SQL parser* is a machine that receives generations of tables from a TLA, receives the text of SQL queries from clients, computes the answers to the queries, and sends back the results. If a SQL parser has all the tables it needs to answer a query, it does so immediately. Otherwise, it sends a request to the TLA and waits for the TLA to send back a generation that contains those tables. To provide results with data collected at about the same time, all the tables used to answer a query are required to be from a single generation.

Just as Cluster Proxies and TLAs publish into Query, so do SQL parsers. When TLAs collect their generations, they can also include data from SQL parsers.

A picture of the Query system is shown in Figure 3. The cloud represents all the thousands of clusters talking to the TLA. The TLA shown is currently providing tables to two SQL parsers. There is a user at a terminal issuing a query against the table published in Figure 1 and Figure 2 to figure out which machines on the network have less than 3% of space available on some mount point. There are actually hundreds of TLAs and SQL parsers, but only one TLA and two SQL parsers are shown.

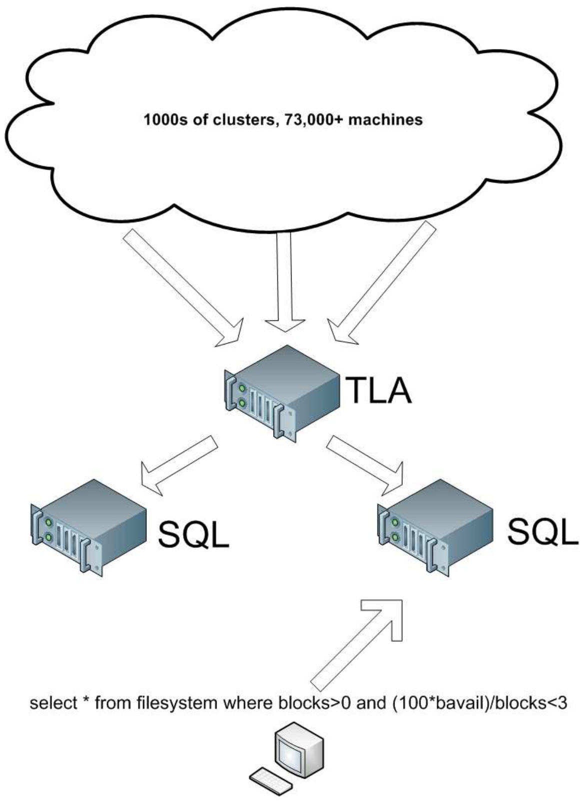

1000s of clusters, 73,000+ machines
TLA
SQL
SQL
select * from filesystem where blocks>0 and (100*bavail)/blocks<3

Figure 3: The Query system. The cloud is the whole Akamai network. Also shown are a TLA, the two SQL parsers getting tables from it, and a user at a terminal issuing a query.

3.5 Aggregator Sets

Not all queries are interested in data from the whole network, so not all TLAs talk to the whole network. For example, some queries' sole purpose is to monitor the health of the TLAs and SQL parsers. Those queries can get sent to machines that contain only data from the machines they are interested in. Each TLA can be configured to talk to only a subset of the network, and each SQL parser can be configured to talk to only a certain set of TLAs. We call the subset a TLA talks to its *span*. Because a SQL parser can get exactly the same data its TLAs can get, the span of a SQL parser is the same as the span of its TLAs.

Different users have different needs. For some users, latency is critical, and they need to issue queries to machines that are lightly loaded so that they never have to wait for a machine to collect a large generation before it can compute the answers. Other users need to join so much data that the issuing of their queries alone will make a machine heavily loaded.

We handle these disparate needs by dividing clients who issue queries into groups and giving each group some set of aggregators. We call the group to which a TLA or SQL parser is assigned that machine's *domain*.

Any time a set of TLAs or SQL parsers share a span and domain, the machines of each type in that set are performing the same job and are interchangeable. We call such a set of TLAs and SQL parsers sharing a span and domain an *aggregator set*.

3.6 Combined TLA-SQLs

Some aggregator sets are under light enough load that one machine can actually do all the work of a TLA and all the work of a SQL parser for that set. We configure those sets to do just that, to reduce our machine count and our costs. We call a machine doing the work of a TLA and a SQL parser a *TLA/SQL*.

3.7 Overall Network Distribution

Currently, Akamai has several hundred TLAs, SQL parsers, and TLA/SQLs divided into several dozen aggregator sets. Each aggregator set has at least three tuples of TLAs and SQLs for fault tolerance, and often many more, depending on its load.

4 Features

In this section we elucidate several of Query's features and explain how they empower its users.

4.1 Near Real-Time View

Query makes a new generation at each machine every minute or two, so the data at the edges of the network are at most two minutes old. Seconds go by between the collection of those generations and the aggregation of generations at the Cluster Proxies. Seconds more pass before aggregation begins at a TLA, a process which takes tens of seconds on the TLAs with the heaviest load. Encoding a generation to send to a SQL parser and decoding it at the SQL parser each take tens of seconds. Consequently, data at the SQL parsers can be a few minutes old.

Compared to the amount of time it may take for a human to diagnose and respond to a problem, a few minutes is not much. However, because Query's work is the first step in detecting and understanding a problem, a minute of time spent before data get into the results of queries represents a minute delay in the rest of the response process. Therefore, even though the latency of data is fairly low, continuing to lower it remains a priority.

Query's reliability is not perfect (see Section 8). Consequently, data in Query cannot be relied upon for certain things. Nevertheless, it is one of our fastest means of getting information, and sometimes we need fresh data, even if they are imperfect. In such situations, we use Query.

4.2 Synchronization

All data collected from a machine are collected within the span of several seconds. Although a TLA may have data collected potentially minutes apart from two different edge machines, its data from any one machine were published at about the same time. This condition is weak enough that we can achieve it without much overhead, but strong enough to provide some valuable abilities to the company.

The low variance in age of data from a given machine means we don't miss multiple related conditions, or the correlations among them. For example, suppose some rare event lead to the consumption of a large amount of memory. When one datum is present, the other will be as well, allowing us to detect such correlations.

4.3 Historical View

Query is used not just to understand the state of the network now, but also how it has changed. Query can be used to record prior data to get a view of past partial states of the Akamai network.

We have two means of doing this. The first is Akamai's historical reporting system, which will be described in Section 5. This system stores the results of queries for a long time and displays them in graphs, giving us a visual representation of how data in Query have changed. The second is Query History, a feature whereby aggregators can be configured to store old generations' copies of certain tables, load them, and answer queries based on them.

The ability to get a historical view has tremendous power. It allows us to see how usage patterns have changed over time, predicting future growth in usage based on past trends. It allows us to correlate changes in multiple parameters, so that we can know how much CPU is consumed by additional end user requests, how much memory, how much bandwidth, etc. If we detect a problem after it has existed for a while (say due to a software bug causing occasional spikes in the usage of some resource), we can figure out when that condition started to exist, helping us narrow down the cause.

4.4 Static Tables

Some tables don't change often and have contents that should be dictated by the structure of the network or some sort of unchanging information. There is no reason to spend resources to aggregate such tables through the normal Query system at the cluster level. Instead, we store tables in text files on the disks of TLAs, and we store index files describing where those files reside. Each TLA reads its static data off of its disk, adds it to the data it has from the Cluster Proxies, and re-reads the data any time they change.

Below is an example of a query that joins normally published data with static data. It looks at three tables: (1) load_info, which has information about all requests Akamai is currently handling; (2) region_data, which describes data about the geographical regions our machines are in; and (3) continent_data, which describes information about the seven continents. The query computes how many hits we're serving on each continent per second.

```
SELECT   c.continent_name,
         SUM(l.hits) hits
FROM     load_info l,
         region_data r,
         continent_data c
WHERE    l.georegion=r.id AND
         r.continent=c.continent
GROUP BY c.continent_name
ORDER BY hits DESC;
```

```
c.continent_name       hits
----------------  ---------
   North America  4,620,551
          Europe  3,392,102
   South America    655,175
            Asia    552,258
          Africa    106,781
         Oceania     39,905
      Antarctica        135
```

A query similar to this one is used to generate one of the graphs Akamai displays on its web site [4]. The numbers of hits and even the ordering of continents change throughout a typical day. That data, for example, were collected at about 3:15 PM Eastern Standard Time, when one would expect most of the Americas and Europe to be awake, but most of Asia and Australia to be asleep.

5 Applications

We now explain several of the key uses of Query and how they empower operations staff at Akamai.

5.1 Alert System

Akamai's alert system is the primary tool for detecting problematic conditions on the Akamai network. Engineers and operations staff can easily develop and activate alerts by writing SQL statements which are submitted to the Query system at regular intervals. For example, consider this simplified SQL statement to detect disks with less than 3% of their disk space left free:

```
SELECT
    machineip     ip_key,
    mountp       mnt_key,
    bavail*bsize free_space,
    (100*bavail)/blocks pct
FROM
    filesystem a
WHERE
    blocks > 0 and
    (100*bavail)/blocks < 3;
```

```
   ip_key    mnt_key  free_space  pct
------------  ----  -----------  --
10.123.123.1  /var  150,179,840   2
10.123.123.7  /var   72,216,576   1
```

The SQL statement along with many other configurable settings form an *alert definition*. Each row returned by the SQL statement constitutes a problematic condition, or an *alert instance*. Each time the alert query is run, the result is compared to the previous result. Any new rows are considered new instances of the alert. As soon as an alert instance is detected, the alert is said to *fire*. If any rows from the previous iteration are no longer present, the alert is said to *clear*.

Akamai has found it important to tune when alerts fire and clear. For example, when writing a "High CPU usage" alert for a critical server, we may want to fire an alert when CPU is over 98% usage. A single spike to 98% isn't interesting but if we check every 2 minutes and the CPU is still greater than 98% after 15 iterations, then there is clearly a more chronic condition worthy of investigation. On the other hand, when writing a "Disk showing SCSI errors" alert, we would want to ensure the alert stays active even if the underlying disk errors do not repeat. This gives time for the operations staff to react to the alert and investigate the condition further.

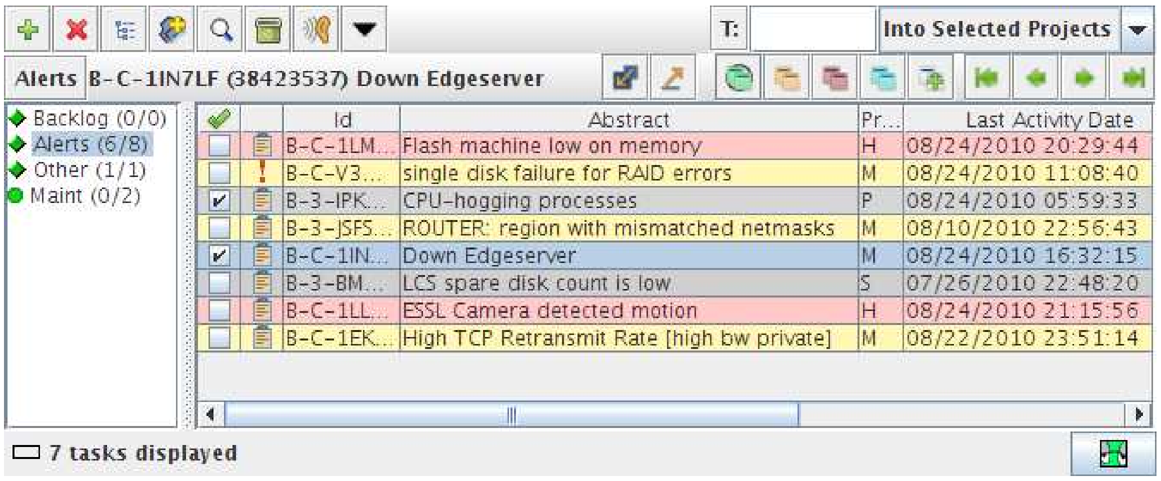
As a result, three commonly used alert definition settings deal with these timing parameters:

- Frequency of SQL execution (typically one minute).
- Number of iterations the data are present before an alert fires.
- Amount of time the data must be absent before an alert clears.

When an alert fires, the alert system can be configured to do one of two things. It can alert staff in the 24/7 Network Operations Control Center (NOCC), which is done for urgent matters, or it can send an e-mail to engineering or operations staff for later follow-up. In the former case, the NOCC staff can take appropriate action using a custom user interface shown in Figure 4. The user interface combines the alert details with corresponding procedure, network access and ticketing. In many cases, alert procedure steps include analysis using further Query data. The NOCC can routinely handle over 10,000 new alert instances in a single day with this approach, coming from over 73,000 machines. (That figure includes problems on partner networks, and problems that the Akamai mapping system can automatically route around.)

At present, there are several thousand queries that run to detect alert conditions, with multiple thousands running every minute. The alert SQL queries are typically much longer than the example queries above, sometimes with pages of complicated SQL logic. Using techniques such as the ones we describe in Section 6, we have allowed a few tens of TLAs and SQL parsers to handle all of this load.

The alert system and Query provide several advantages for incident detection and response. If an operations staff member begins to suspect a problem and wants to create a query to detect it, that person can create a query in a matter of minutes, start testing it immediately to make sure it produces the desired results and doesn't


T: Into Selected Projects

Alerts B-C-1IN7LF (38423537) Down Edgeserver

Backlog (0/0)
Alerts (6/8)
Other (1/1)
Maint (0/2)

✓		Id	Abstract	Pr...	Last Activity Date
		B-C-1LM...	Flash machine low on memory	H	08/24/2010 20:29:44
	!	B-C-V3...	single disk failure for RAID errors	M	08/24/2010 11:08:40
✓		B-3-IPK...	CPU-hogging processes	P	08/24/2010 05:59:33
		B-3-JSFS...	ROUTER: region with mismatched netmasks	M	08/10/2010 22:56:43
✓		B-C-1IN...	Down Edgeserver	M	08/24/2010 16:32:15
		B-3-BM...	LCS spare disk count is low	S	07/26/2010 22:48:20
		B-C-1LL...	ESSL Camera detected motion	H	08/24/2010 21:15:56
		B-C-1EK...	High TCP Retransmit Rate [high bw private]	M	08/22/2010 23:51:14

7 tasks displayed

Figure 4: Alert-handling interface.

consume too many resources, and begin collecting results very quickly. Query also allows us to detect a problematic condition, then examine a large amount of information about it, all using one tool. Even if we did not anticipate needing some specific information before a problem is detected, we can write new queries and issue them at any time to help diagnose a problem.

The need to run the alert system using Query imposes several key constraints on Query's design that relate to the tradeoffs described in Section 2. Reliability, completeness, and low data latency are critical for the alert system. When the alert system issues a query, the answer needs to come back reliably, quickly enough, and be computed with data that are fresh and comprehensive enough to detect the problem promptly and respond to it before it affects our service to customers. The alert system also needs Query to be scalable. The ability to issue alerts to detect a wide variety of problems is quite useful. When the network grows in size and is handling more traffic, we need to continue to be able to answer all of the existing alert queries, as well as new ones that become necessary.

5.2 Historical Reporting System

Another tool for analyzing and diagnosing the Akamai network is the historical reporting system, which collects and stores data from Query over time and graphs the results. The reporting system is Akamai's primary tool for observing how the network has changed over time. While we use the alert system to detect issues that need immediate attention, we use the reporting system to proactively analyze network behavior with the intention of preventing issues before they occur.

Much like the alert system, the reporting system stores several thousand queries written by developers and operations staff. Each query is issued every few minutes and the results are shown on graphs. The system provides various ways of displaying data to assist in visualizing and understanding the parameters of the network.

The resolution of the reporting system, which issues each query once every several minutes, is insufficient to detect problems and respond to them in real-time. It is sufficient, however, to help understand problematic conditions over the span of several hours. For example, a bug in Query itself once caused it to consume too much CPU. Due to the difference in scale between the Akamai network and the test network, this bug was not realizable in the test environment. After deploying the new software with the bug to a small number of machines, the alert system detected the increase in CPU load on some machines. Before deploying the software to more machines, we investigated the problem. The reporting system showed spikes in the CPU utilization of Query on certain machines, and seeing the frequency of CPU spikes helped in diagnosing the bug.

5.3 Customer Access

Several Akamai services that provide data to customers use Query to collect that data. Most customer interaction with those systems is through a web-based interface, EdgeControl [3], which is the Akamai customer portal.

The alert system can issue alert queries on customers' behalf, notifying a customer if one of that customer's alerts fires. That notification may be done via e-mail, a web service call, or through an SNMP MIB that runs on the customer's site (which allows customer alerts to be integrated with local monitoring clients like Openview [10], and Tivoli [11]).

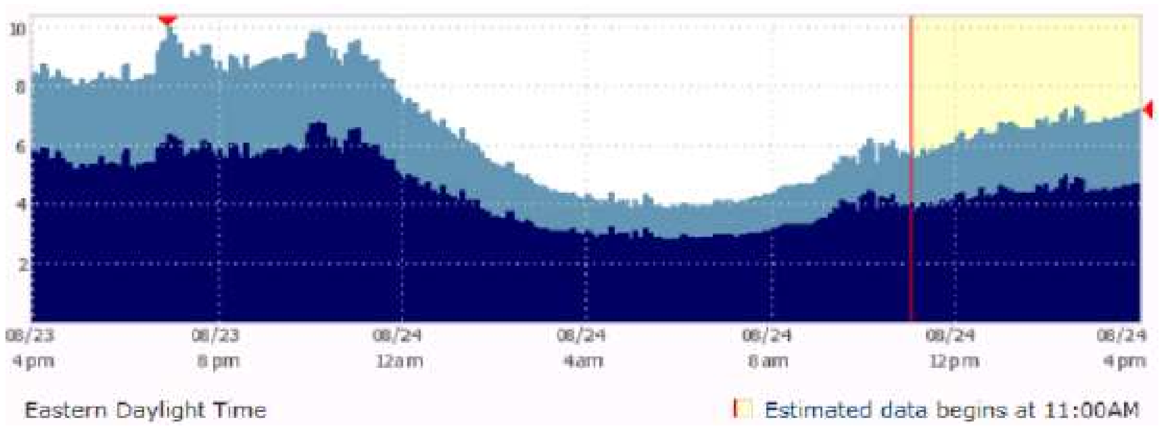


Figure 5: Customer access to traffic data via EdgeControl. The highlighted estimated data come from Query.

In addition to providing certain alerts to customers, we provide each customer with graphs of various data, such as how much traffic we have served for that customer over time. Figure 5 shows an example of such a graph. Query does not achieve perfect completeness. The first several hours of data are based on processing logs of all traffic we have served. However, we want to display usage graphs to our customers more quickly than we can process all the logs. Therefore we show customers log data until the latest time they are available, then show what we call estimated data for the most recent time period. That estimated data come from Query.

Several requirements arise from the fact that the data are customer-visible. The machines collecting the data have to be reliable, to have uninterrupted data display. In order for our displayed estimates to be as accurate as possible, the data need to be as complete as possible. Data must also be consistent across Query to avoid graph discrepancies. This is because a query may be issued to one SQL parser at a particular time and then issued to another SQL parser several minutes later. Additionally, data latency and query latency must be low, because we want to display near real-time data to customers quickly, providing them with current estimates. Finally, several of the queries whose results are displayed to customers are expensive and grow rapidly, joining multiple tables. Several of these tables grow as the number of machines in the Akamai network and the number of Akamai customers grow. Thus, Query must be scalable, to continue to handle the load from collecting the data for those graphs. Failure to provide features such as the ones outlined above would be unacceptable to customers.

5.4 Incident Response

An *incident* is an urgent occurrence that adversely affects customers, or may adversely affect them if left unchecked. Query is a vital tool for incident response at Akamai. As previously explained, it underlies the alert system and the reporting system, two important tools for incident response. Often, incidents begin when the alert system detects a high severity problem. If the problem is related to any of the thousands of graphs collected in the reporting system, that is another tool for understanding the problem.

In addition to being used by these tools that help with incident response, responders often issue SQL statements directly to Query. Much of the time, some information published into Query can help illuminate the problem and possible solutions. This use of Query, again, emphasizes certain goals for Query's design. Data latency and query latency are both vitally important: in an incident, we need to end problems before they impact customers, or minimize their impact, and every minute counts. Scalability is also important, because we don't know what tables will be needed until the incident takes place. The system may need to get any data from any or all of Akamai's machines, and the amount of data collected is far larger than any one machine can hold. The ability to divide data up among many machines, providing scalability, is vital to handling incidents. Our techniques for achieving such scalability are addressed in Section 6.

6 Scalability

In this section, we discuss the reasons Query has high needs for scalability and how we deal with those needs. We will address three ways of achieving scalability:

caching at each machine only the tables that machine needs; partitioning the network so that each machine needs only a subset of the data; and adding SQL parsers.

6.1 Causes of Growth

The volume of data and queries a TLA or SQL parser must be able to handle depends on several factors: the number of machines on the network publishing into Query, the number of customers about whom data are published, the volume of traffic on the network to be monitored, the number of services on each machine publishing data, and the number of ideas for things to monitor that people have come up with, among others. All of these factors grow monotonically with time.

As time goes by, Akamai signs more customers. As the Internet grows, our customers have more customers, so Akamai must serve more end users, leading to more traffic. To handle this additional load, we must deploy more servers. Managing rapid growth is one of the major challenges in the design and operation of Query.

6.2 Caching Policies

We try to cache tables around the SQL parsers and TLAs such that (1) each machine always has many of the tables it will need soon, and (2) machines have few tables they won't need soon. This is necessary because the volume of data available in Query is far larger than any single Query machine can hold: tens of gigabytes, a generation of which would take minutes to decode. That's why SQL parsers request tables from TLAs, which in turn request them from Cluster Proxies, which request them from the machines in their clusters.

Each machine *prewarms* tables. This means that it fetches those tables whether it needs them or not. That dramatically reduces query latency, because if the tables a query needs are already resident on the SQL parser, it doesn't need to spend minutes fetching them from the clusters through the TLAs. We can configure each aggregator set to prewarm its own distinct set of tables. That set can be thought of as our guess for which tables the aggregator will need. For example, we know what queries the alert system needs to issue for alerts. That means aggregators devoted to the alert system will need a specific known set of tables, so we prewarm that set.

Of course, tables exhibit temporal locality of reference: if a user issues a query using a table, that table is likely to be used again in the near future. If a table that isn't prewarmed is used, it continues to be requested in all generations for some period of time afterwards, and that timer is reset every time the table is used again.

A second type of caching is views. We cache the results of every view we compute for use in future queries, invalidating that cache every time we switch to using a new generation of tables. There are about 1000 view queries defined, many of which describe common subqueries. Storing their results reduces query latency and improves scaling in the number of queries by avoiding repeated computations. It also reduces the memory load on the SQL parsers, because the intermediate state for computing the answers to queries can be reduced by computing them fewer times.

Another technique for improving scalability is *diff updates*. Instead of sending a full copy of each encoded table to each TLA, the Cluster Proxies send only a diff – that is, a description of how the tables in that cluster have changed. The first time a Cluster Proxy sends data to a particular TLA, it sends a full generation, but subsequently, it sends only the diff. This makes TLAs decode tables more quickly, and saves about half the bandwidth Query would otherwise need to consume.

6.3 Partitioning

Partitioning the Query system can provide scalability. We have three ways of doing this: we can partition the network, we can partition the users, and we can partition the tables each individual user needs.

6.3.1 Network Partitioning

Talking to 73,000 machines takes a lot of resources from each TLA, but not all issuers of queries are interested in data from the whole network. Therefore we designate certain subsets of the Akamai network to be the span of each machine, as described in Section 3.5. For example, aggregators whose purpose is to monitor the Query system itself need span only the few hundred aggregators, not all 73,000 or more Akamai machines. Aggregators with small spans suffer far fewer demands on their memory, bandwidth, and CPU than machines that span all of Akamai.

6.3.2 User Partitioning

We don't want users going to randomly chosen aggregators to issue queries. Some applications, like the alert system, are critical, and must send queries to machines we know will have the resources to handle them. Some users run test queries to see how they perform, and while they are being written, they may mistakenly use excessive amounts of machine resources. This leads to assigning each aggregator set to a user or set of users, as described in 3.5.

After assigning a span and domain for each aggregator set, we can figure out what tables it is likely to need and make sure each machine in the set can decode, aggregate, and store all the tables it needs.

Because each machine in an aggregator set prewarms the same tables, no aggregator set can have more tables than one machine can handle. If a user is so demanding as to need more tables than a machine can handle, that user needs multiple aggregator sets. Users can issue queries to whichever of multiple aggregator sets they need, but operations staff responsible for Query need the ability to change where queries are sent without having to change the software of the components issuing the queries. We place each aggregator set behind a hostname and have users issue their queries to an arbitrary machine that hostname resolves to. Operations staff for Query can then change the machines a hostname resolves to, to add or remove aggregators from a set.

6.3.3 Table Partitioning

Partitioning tables among aggregators also helps with scalability. Suppose an aggregator set has four machines, A, B, C, and D, and the tables it prewarms grow too large for one machine to handle. We can partition that aggregator set into two subsets, say A, B and C, D. On each subset, we prewarm half the tables the original set had. We point the same hostname at all four machines, but if A gets a query for which it doesn't have the tables and C, D do have them, A can send, in place of an answer, a message redirecting the query to C, D. The programmatic interface to Query then automatically goes to C or D to get its answer. In practice, the partitioning can't be perfectly even and some overlap between the tables on A, B and the tables on C, D must exist to still answer every query users want to issue. To date, in all cases where we have tried to partition an aggregator set in this fashion, no machine has needed more than 55% of the data of the original set.

6.3.4 Aggregator Sets and Fate-Sharing

Different users have different needs, but sometimes their needs are similar enough that they can be grouped together using a single aggregator set. There are a number of benefits, risks, and costs to doing so.

The main benefit is saving machines. Instead of many aggregator sets, we must deploy only one. The main risk is that two users on the same aggregator set share fate. If one user causes a failure, all of them will feel it.

The lesson here is that the most critical users should be isolated, and other users should be placed in groups with shared expectations about reliability and failure. If two non-critical applications that may potentially bring down an aggregator set have to share it, no problems will occur: both applications are non-critical and are designed with aggregator failures in mind.

6.4 Adding SQL Parsers

Akamai provides global traffic management and enhanced DNS services [2], mapping a hostname to several IP addresses and balancing the load among them. Because Query's users issue their requests to hostnames, rather than specific IP addresses, we can allocate the queries approximately evenly among all the IP addresses sharing a hostname. We create a hostname for each aggregator set's SQL parsers (or combined TLA/SQLs) and have our users issue queries to those. If we need to add more machines to handle more queries, we can do so transparently to the user. Twice as many machines can handle twice as many queries.

There are two caveats. First, we cannot simply add SQL parsers, because each TLA won't have the resources to send hundreds of megabytes of encoded tables (gigabytes of decoded tables) to that many other machines during an one or two-minute interval between generations. If we add too many SQL parsers, we must also add TLAs. Second, we must combine this approach with partitioning and wise caching policies. Each SQL parser must decode a generation of tables every time one arrives. Without partitioning and intelligent caching, as the network grows, eventually the SQL parsers will spend most or all of their time decoding generations.

7 Management Lessons

Managing a complex system like Query has taught us several lessons that may be of use to administrators of other systems. We need to be able to fix a variety of problems in Query that arise during operation. Some of these problems are due a user needing more tables than one aggregator set can handle. Some are due to a user needing to issue more expensive queries than their aggregators can handle, due to their requirements in either CPU or memory. Some are from the need for new features. Some are from software bugs. We now explain some of the lessons we have learned about these issues.

7.1 Management options

When a difficult use case arises, either due to new needs or due to organic growth, we have several options:

- Find a less expensive means of achieving the same goal.
- Reconfigure the network.
- Deploy additional hardware.
- Perform additional operational work to handle the use case.

- Develop the Query software to be more efficient about that use case.

- Go without, telling the user that the difficult use case cannot be accommodated.

The first option, finding a less expensive solution, is always the first thing we try, as it clearly saves the company the most money. Unfortunately, it isn't always possible, and the interesting tradeoffs are among the remaining options.

7.2 Configuration Options

For urgent problems, reconfiguring the network is usually the best option if it is a possibility. We can deploy configuration files to the entire network quickly to change what the machines are doing. For example, if a set of queries are taking up a lot of CPU on some set of SQL parsers, but they contain a common subquery, we can push a configuration file that creates a new view to reduce the number of times we need to compute it.

This example shows an important lesson: make rapid changes in behavior easy any time it is safe to do so. Some aspects of a machine's behavior, such as the software version, are difficult to change safely without restarting. Others, such as the views, are easy to change safely without restarting. In early versions of Query, all of these changes required a software install. Now, many just require our configuration management system [19] to copy new configuration files to the machines, which makes us much more reactive.

7.3 Adding Hardware

We can deploy new hardware to fix some problems. If a SQL statement is too expensive for the machines trying to run it, we can always put up additional SQL parsers to send it to. Deploying new machines takes less work than developing new software and can be done much more quickly. If we can't accommodate a request by configuration options or finding a more efficient way to achieve the user's goal, this is by far our most common solution.

7.4 Operational Intervention

Sometimes, a problem can be fixed by operations personnel manually. For example, we found a slow memory leak in Query that affected one set of aggregators such that their resources were essentially all consumed after about a month of continuous operation. We came up with a temporary solution to use until the next regularly scheduled release: manually restart the machines in the set every few weeks.

The lesson we've learned from trying this solution is that it's good for the short term only. It's expensive, because it requires a human in the loop. It's time consuming and stressful for operations personnel. It doesn't actually fix the problem; it's just a way of living with it. We try to use this approach as rarely as possible and to depend upon it only for short periods of time.

7.5 Software Development

Software development is a longer-term activity than pushing a configuration file or deploying new machines. To deploy new software, we must develop it, run it through Quality Assurance, and install it in several phases, allowing time between phases to make sure the part of the network that was installed initially is working properly.

The advantage to developing software to fix a problem or add a feature is that once it's done, the problem is fixed or the feature is available everywhere forever. No one has to do any work to maintain it, and there is no additional hardware cost.

One lesson we have learned about when to develop new software to solve a problem is that it's best to use it after the other solutions, because it's slower, cheaper, and more permanent. A few years ago, there was a bug that caused SQL parsers to be unable to get new data and to continue answering queries with old data. We initially solved this problem with operational work, adding an alert to the alert system to detect the condition and asking the Network Operations Control Center to restart machines when the alert fired. That was a temporary solution that lasted for a few months until we could fix the bug.

8 Handling Faults

With over 73,000 machines publishing into Query and several hundred running infrastructure for Query (TLAs, SQL parsers, and TLA/SQLs), some number of them are down at any time. Sometimes pairs of them can't reach one another. Sometimes TCP sockets between machines fail due to congestion. Sometimes machines have too many resources consumed and can't keep up with all the communication they're supposed to do. This section is about how we handle faults.

There are several goals regarding handling faults, including:

- **Easy detection:** Problems should be found quickly and easily.

- **Fault tolerance:** When a fault occurs, Query should work around it.

- **Quarantining faults:** The scope of a fault should be kept narrow, limiting the number of machines that go down.

8.1 Error Detection

Query is unusual among Akamai systems in that a lot of the other systems can count on Query to detect their faults. If something goes wrong with Query itself, we have an obvious bootstrapping problem.

If something goes wrong with a subset of the Query processes on the network, we can detect it because multiple aggregator sets span all the TLAs, SQL parsers, and TLA/SQLs. If any of those aggregator sets are functioning properly, we can detect problems. Additionally, we can detect problems that cause queries to fail rather quickly, because the absence of an answer coming back registers as an error in, for example, the alert system.

There remain two cases: incorrect results coming back that cause false positives for alerts, and incorrect results that cause false negatives. False positives are easy to deal with: when an error has been detected but the people looking into it can't figure out the root cause, they know to also bring in experts on Query to debug simultaneously. This shows another lesson we have learned: don't forget that your monitoring tools may be the problem when you've detected an error.

False negatives are trickier. Occasionally, an alert will not fire. Usually, we find this is due to a bug in the alert SQL, not a bug in Query. The only way to deal with that is for alert writers to test carefully before and after their alerts are deployed. If there were a mass-scale incident of Query failing to publish data, we would also detect that case, because of a number of alerts that check for the presence of data, not their absence. For example, if data from half the network were to disappear, the alert for Query having data from too few machines would fire almost immediately.

8.2 Fault Tolerance and Quarantining

When deciding how to achieve fault tolerance and quarantine faults, we must keep in mind the tradeoffs of Section 2.2. There is a tension between the two goals. Tolerating faults requires moving load away from a machine that fails, so that its outstanding requests may still be serviced. Quarantining faults requires that load *not* be moved away from a machine that fails, because the load may have caused the failure.

Aggregator sets help us limit the scope of failures while achieving fault tolerance. The SQL parsers of each set have a single hostname pointing to all of them. If a SQL parser fails, the programmatic interface to Query automatically redirects the query to another machine in the set. A Query that consumes enough resources to take down a machine could thus take down the whole set. This can happen occasionally due to an ad hoc query being written by a human, but only on non-critical aggregator sets used for development. If an aggregator set is used by humans writing ad hoc queries, we only send queries to it from applications that are allowed to fail to get answers sometimes.

If a TLA goes down, any SQL parsers talking to it continue providing answers to queries with old data until they can get tables from another TLA in the same aggregator set. This typically takes a few minutes (not much more than a normal interval between generations). This allows the same balance between tolerating faults and quarantining them as for SQL parsers failing.

If a TLA loses its connection to a cluster, similarly, SQL parsers switch away from it. Each TLA advertises how many clusters it can see and SQL parsers take that information into account when selecting TLAs. Initially, each SQL parser chooses a TLA arbitrarily. Suppose SQL parser S chose TLA $T1$. If $T1$ loses visibility to some clusters, some other TLA, $T2$, may gain the ability to see some percentage more clusters than $T1$ can. S will then switch to using $T2$ instead of $T1$. If there are multiple such $T2$, S will switch to an arbitrary one.

Usually, if there are connectivity problems, one TLA will fail to see some set of clusters, but the other TLAs will be able to see it. In other words, there will be multiple possible choices for a $T2$ to switch to. That prevents the switching algorithm from placing extreme load on any one TLA.

We want SQL parsers to prefer TLAs that are geographically close to them. Using configuration files we can tell each SQL parser to give a bonus to some set of TLAs when deciding which one to use. That helps make the mapping from SQL parsers to TLAs more static and prefer close by machines, while also helping each SQL parser have as complete a view of the network as possible.

TLAs are normally required to have a full view of the network before they can collect a generation. Each cluster must have reported tables within a certain time interval. If a cluster has failed to do so, the TLA drops the cluster's tables and advertises one fewer cluster, so that SQL parsers can switch away from it as needed.

9 Related Work

In this section we review related work in the area of large-scale network monitoring that has appeared throughout Query's lifetime of approximately 12 years.

Several system administration tools such as Nagios [14], Microsoft SCOM [21], Hewlett-Packard OpenView [10], IBM Tivoli [11], and Sun Management

Center [20] exist for monitoring network services and machine resources, often using SNMP. Akamai accomplishes network monitoring by feeding data collected and aggregated by Query into applications such as the ones discussed in Section 5. Query allows users to specify complex monitoring tasks using a SQL-like interface. In addition to providing a familiar interface, Query's focus is on scaling its monitoring capabilities to tens of thousands of machines, while still providing near real-time updates. Via a software component that acts as an SNMP gateway, Query is able to collect data published by SNMP-enabled devices, as was described in Section 3.1. Similarly, Query is also able to export data as an SNMP MIB, as was described in Section 5.3.

One common approach to network management for security purposes is Security Event Managers (SEMs). An SEM logs all events it expects will be interesting to system administrators. When a problem is detected, the SEM provides a means of reading the logs from each machine and presenting them to the administrators. By publishing such data into Query, Akamai has all of that information in one place that is easy to monitor constantly by human users and automated applications. Additionally, queries can be issued right away, minimizing the setup time for detecting conditions of interest.

Processing large volumes of continuously updated data in real-time has also been the focus of several academic and industrial research projects in the area of stream processing systems. Telegraph [6], STREAM [13], and Aurora/Medusa [7] were the first generation of such systems, with a focus on providing a SQL-like interface to query continuously updated data. The next generation of such systems focused on distributed implementations, to increase both the scalability and the fault-tolerance of low-latency, high-throughput stream processing applications. Borealis [1] has focused on challenges related to implementing a stream processing system in a distributed fashion, with particular emphasis in load shedding and fault-tolerance. Synergy [18] has focused on composing distributed stream processing applications, while paying attention to their end-to-end Quality of Service requirements. Among industrial research efforts in the area of distributed stream processing, IBM's System S [23] has focused on a variety of stream processing applications with highly variable rates, utilizing a large number of stream processing nodes. Additionally, AT&T's Gigascope [8] has focused on monitoring network traffic at extremely high-volumes. Similar to the systems above, one of Query's main challenges comes from the large data volumes that need to be processed near real-time. Query addresses this challenge via the clustered architecture outlined in Section 3 and the techniques described in Section 6.

Research efforts have also focused on the challenges faced by large-scale network monitoring systems, both due to data volume and network size, as well as due to network and machine failures. SDIMS [24] has attacked the scalability challenges by using Distributed Hash Tables to create scalable aggregation trees. It has also utilized lazy and on-demand reaggregation to adjust to network and node reconfigurations. PRISM [12] has proposed imprecision to provide consistency guarantees with reduced monitoring overhead and despite failures. Specifically, arithmetic imprecision was proposed to bound numeric inaccuracy, temporal imprecision to bound update delays, and network imprecision to bound uncertainty due to network and node failures. Query faces similar tradeoffs, as was described in sections 2 and 8.

Distributed event services can also be used for network monitoring. Research projects in this area, such as Siena [5] and ECho [9], have focused on maximizing the performance of event notification, while providing data models that are generic enough to express a variety of events. CORBA also provides support for event [16] and notification [17] services. Akamai uses Query to collect data from many different software components, implemented in a variety of programming languages. To achieve that, the publishers utilize various native language interfaces that Query provides, as was described in Section 3.1.

10 Conclusion

We have explained the goals and design of Query, Akamai's near real-time monitoring system. We have presented a number of the issues we face developing, managing, and operating it. We have stated some of the lessons we have learned from our experiences. Management of Query has been made much easier by the availability of rapid changes in configuration; isolating critical users and putting others into groups of similar reliability expectations; having multiple ways of addressing a problem in both the short term and the long term, and being explicit about which ones are good for each time scale; and having a strategy for debugging our monitoring tools. All of those strategies have allowed Query to scale with the Akamai network and handle the growth of load that it has been experiencing for more than a decade.

11 Acknowledgements

The authors would like to thank everyone who has developed and supported Query throughout the years, including Andy Berkheimer, Josh Buresh-Oppenheim, Timo Burkard, James Chalfant, Ron Chaney, Greg Fletcher, Stephen Gildea, Dan Katz, Sef Kloninger, Alan Krantz, Phil Lisiecki, Andrew Logan, Brian Mancuso, Barry Margolin, Erik Nygren, Tim Olsen, James Park, Jan-Michael Santos, and Brad Wasson.

We would also like to thank Andy Ellis, our shepherd Chad Verbowski, and the anonymous reviewers, for valuable advice on the writing of this paper.

References

[1] ABADI, D., AHMAD, Y., BALAZINSKA, M., CETINTEMEL, U., CHERNIACK, M., HWANG, J., LINDNER, W., MASKEY, A., RASIN, A., RYVKINA, E., TATBUL, N., XING, Y., AND ZDONIK, S. The design of the Borealis stream processing engine. In *Proceedings of the Second Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2005).

[2] AKAMAI EDGE PLATFORM PRODUCTS. `http://www.akamai.com/html/technology/products/index.html`, 2010.

[3] AKAMAI EDGECONTROL. `https://control.akamai.com`, 2010.

[4] AKAMAI EDGEPLATFORM. `http://www.akamai.com/html/technology/dataviz3.html`, 2010.

[5] CARZANIGA, A., ROSENBLUM, D., AND WOLF, A. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC), Portland, OR, USA* (July 2000).

[6] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., ANDJ.M. HELLERSTEIN, M. F., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[7] CHERNIACK, M., BALAKRISHNAN, H., BALAZINSKA, M., CARNEY, D., CETINTEMEL, U., XING, Y., AND ZDONIK, S. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[8] CRANOR, C., JOHNSON, T., SPATASCHEK, O., AND SHKAPENYUK, V. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, CA, USA* (June 2003).

[9] EISENHAUER, G., BUSTAMANTE, F., AND SCHWAN, K. Event services for high performance computing. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC), Pittsburgh, PA, USA* (August 2000).

[10] ENTERPRISE MANAGEMENT SOFTWARE: HP OPENVIEW. `http://www.managementsoftware.hp.com/`, 2010.

[11] IBM TIVOLI SOFTWARE. `http://www.ibm.com/software/tivoli/`, 2010.

[12] JAIN, N., MAHAJAN, P., KIT, D., YALAGANDULA, P., DAHLIN, M., AND ZHANG, Y. Network imprecision: A new consistency metric for scalable monitoring. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, CA, USA* (December 2008).

[13] MOTWANI, R., WIDOM, J., ARASU, A., BABCOCK, B., BABU, S., DATAR, M., MANKU, G., OLSTON, C., ROSENSTEIN, J., AND VARMA, R. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research, CIDR, Asilomar, CA, USA* (January 2003).

[14] NAGIOS. `http://www.nagios.org/`, 2010.

[15] NYGREN, E., SITARAMAN, R., AND SUN, J. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (July 2010).

[16] OMG. Event service specification v.1.2, 2004-10-02, 2004.

[17] OMG. Notification service specification v.1.1, 2004-10-11, 2004.

[18] REPANTIS, T., GU, X., AND KALOGERAKI, V. QoS-aware shared component composition for distributed stream processing systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS) 20*, 7 (July 2009), 968–982.

[19] SHERMAN, A., LISIECKI, P., BERKHEIMER, A., AND WEIN, J. ACMS: The Akamai configuration management system. In *Proceedings of the 2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI), Boston, MA, USA* (May 2005).

[20] SUN MANAGEMENT CENTER. `http://www.sun.com/software/products/sunmanagementcenter/`, 2010.

[21] SYSTEM CENTER OPERATIONS MANAGER (SCOM). `http://www.microsoft.com/systemcenter/en/us/operations-manager.aspx`, 2010.

[22] VISUALIZING GLOBAL WEB PERFORMANCE WITH AKAMAI. `http://www.akamai.com/html/technology/visualizing_akamai.html`, 2010.

[23] WU, K., YU, P., GEDIK, B., HILDRUM, K., AGGARWAL, C., BOUILLET, E., FAN, W., GEORGE, D., GU, X., LUO, G., AND WANG, H. Challenges and experience in prototyping a multimodal stream analytic and monitoring application on System S. In *Proceedings of the 33rd Very Large Databases Conference (VLDB), Vienna, Austria* (September 2007).

[24] YALAGANDULA, P., AND DAHLIN, M. A scalable distributed information management system. In *Proceedings of the 2004 ACM SIGCOMM International Conference on Data Communication, Portland, OR, USA* (August 2004).

Troubleshooting with human-readable automated reasoning

Alva L. Couch
Tufts University

Mark Burgess
Oslo University College
and Cfengine AS

Abstract

In troubleshooting a complex system, hidden dependencies manifest in unexpected ways. We present a methodology for uncovering dependencies between behavior and configuration by exploiting what we call "weak transitive relationships" in the architecture of a system. The user specifies known architectural relationships between components, plus a set of inference rules for discovering new ones. A software system uses these to infer new relationships and suggest culprits that might cause a specific behavior. This serves both as a memory aid and to quickly enumerate potential causes of symptoms. Architectural descriptions, including selected data from Configuration Management Databases (CMDB) contain most of the information needed to perform this analysis. Thus the user can obtain valuable information from such a database with little effort.

1 Introduction

Troubleshooting is about linking symptoms with causes. The speed of troubleshooting depends upon how quickly one can do that, as well as how complete the list of potential causes can be made. It can further be enhanced so that more frequent causes are checked first. In a very complex system, it can be laborious to make a list all of the potential causes of a behavior.

In this paper, we present a method for deriving a description of causal relationships from a description of system knowledge. This method maps symptoms to possible causes via a methodology that we call "weak transitivity". Architectural facts and logical inference rules describe relationships between architecture and causation in a knowledge network. So, while architecture might vary, inference rules, delimiting meanings of relationships, are invariant and reusable. One can use these rules to efficiently reason about potential causes and to eliminate options incrementally as troubleshooting progresses. The system's logic and reasoning are straightforward, simple to understand, and scalable to arbitrarily large networks.

The key contributions of this work include:

1. An exterior ("black box") model of the meaning of relationships between architectural components, that permits logical inference based on incomplete or partial information.

2. The ability to exploit existing knowledge – e.g in Configuration Management Databases – to aid in the troubleshooting process.

3. The ability to generate a human-readable explanation of the possibly subtle relationships between components.

4. A set of useful, reusable classes and relationships along with rules that define their meanings.

2 Background

Our work arose from ideas for and against the use of logical reasoning in system administration[4, 9], but we approach the problem of applying logic to system administration from a new angle based on knowledge representation, specifically Topic Maps[23, 24]. In using topic maps to index documentation, we found that a particular way of thinking about the map led to more efficient use of documentation. If we view the map as a set of *links* between topics, it is easy to get lost in the map, while if we view a map as a set of chains of *reasoning*, the relationships become clearer and the map becomes more useful[7]. The same kind of reasoning that can be used to understand documentation can be utilized to understand complex *systems*. This paper applies our approach to the specific task of troubleshooting, which is – at its core – a problem of understanding and coping with what is known and unknown.

There are plenty of other approaches to troubleshooting[25]. Snitch[17] applies a maximum-entropy approach to creating dynamic decision trees for troubleshooting support, using a probabilistic model inferred from practice. Snitch is related to "revealed causal modeling"[18, 19], which also attempts to measure causality as a set of probabilities of relationships. Troubleshooting has an intimate relationship with cost of operations[12], which justifies use of decision trees and other probabilistic tools to minimize cost and maximize value. The Maelstrom approach[8] exploits self-organization in troubleshooting to re-organize the process based upon hidden precedences. STRIDER[26] employs knowledge of behavior of similar hosts and Windows registries to infer possible trouble points. Outside the system administration domain, SASCO[15] guides troubleshooting by heuristics, using what it calls a "greedy approach" to pick most likely paths to a solution.

There are several differences between our work and these prior approaches to troubleshooting. We base our troubleshooting upon an incomplete description of the *architecture* of the system under test, rather than statistical information about likelihood. We use architectural reasoning to infer the nature of dependencies in the system, and use those inferences to guide troubleshooting. This leads to a synergy between the accuracy of the description and effectiveness of troubleshooting, which leads in turn to increasing accuracy of the architectural information as it is revised to reflect observations. The net result is that we show how to apply something we already need to have – a global map of the architecture – to the troubleshooting process.

2.1 Formal reasoning

While it is certainly a kind of formal reasoning, this work is difficult to place in the context of other approaches to formal reasoning. It is a form of logical abduction[13, 16, 20] that explains connections between entities. Very complex systems have been built to reason using abduction, but none of these is guaranteed to output an easily understandable sequence of logical dependencies. Our method has its roots in using logic programming for configuration management[9], but also takes inspiration from methods used to manipulate topic maps in library science[23, 24], and is closely related to ontological reasoning in the semantic web. Unlike ontological reasoning, which attempts to match concepts based upon their interaction with others, we concentrate on inferring relationships between individual entities, based upon facts and rules that describe an architecture.

Our methods are somewhat removed from traditional approaches to logical inference and computer logic. First, we sidestep the difficult problem of reasoning with modal logic, by encoding modality into our relationships. A "modal logic" includes the ability to distinguish modal facts in English, e.g., "X might affect Y" from non-modal facts, e.g., "X affects Y." Instead of modeling modality, we incorporate all modality into our relationships, which makes our rules for relationships somewhat more complex, but also reusable and perhaps easier to compute.

2.2 Information modeling

Our work includes a limited form of information modeling as proposed by Parsons[21]. However, our notation escapes what Parsons calls the "tyranny of classification"[22] in which an instance *must* be a member of some class. We escape that tyranny by only partially defining such classifications, and leaving what is unknown out of the data specification. Likewise, our data are much simpler in structure from that in the Shared Information and Data model(SID)[14], mostly due to lack of structure (or even the need for structure) in our approach.

2.3 Knowledge management

Our problem is a sub-problem of the larger issue of knowledge management for complex networks. Knowledge management is a key challenge of the coming decade. The technologies and tools for system administration and configuration management have all progressed to the point where the main difficulty lies in the knowledge required to integrate them to produce a seamless IT infrastructure. With many of the basic problems of system administration essentially solved, a major system administration concern of the next decade will be loss of business continuity, due to inability to maintain and utilize appropriate systems knowledge. For example, when system administrators are fired or leave, the business can suffer from lack of knowledge of what they did, resulting in increased downtime, risk, and cost. It costs real money for a new system administrator to learn what his or her predecessors did. Knowledge and understanding of system complexity are also major limitations to system growth (scalability).

Cfengine was recently redesigned with knowledge in mind, using a model of "promises"[1, 4, 5, 6] that separates the intentions of system components from the mechanism by which they achieve those intentions. Promises combine clearly defined goals with self-documenting statements that have an associative structure. From there, it is a small step to create an associative meta-model (semantic web) of promises, which can be integrated with any other kind of semantically annotated documentation. Such a knowledge model can be used not only for searching for relevant information, but also for reasoning and for encoding expertise. Expert systems have been discussed many times before, but they are usually data-intensive and expensive to maintain. Here, we present a mechanism that is both cheaper and is designed to work for humans rather than to replace them. Most important, it arises naturally from the act of managing systems with Cfengine and requires no separate data collection.

2.4 Configuration Management Databases (CMDB)

Configuration Management Databases (CMDB), as defined by the IT Infrastructure Library (ITIL), gather system data, usually in a brute-force taxonomic form. Common data models in use include the Common Information Model (CIM) and the Shared Information and Data Model (SID). These concentrate on configuration *data* of specific hosts, while their meanings and inter-relationships are assumed to be entirely implicit in the taxonomy. The problem with this (and all hierarchical classifications) is that new information can only be introduced by expanding the model itself.

Our technology was developed specifically for Cfengine and its `cf-know` utility (where the required architectural model is available), though we describe the techniques we use more generally here. The lesson from Cfengine is that meta-models with weak constraints avoid many of the pitfalls of 'Object Oriented' hierarchical classification. The techniques can be used with any kind of configuration management database, provided that one can mine appropriate kinds of relationships from it.

3 A motivating example

Using architectural knowledge for troubleshooting might be a counter-intuitive idea, so here is a simple example. Suppose we have a very simple network with a fileserver '`host01`', a DNS server '`host02`', and a client workstation '`host03`'. We might code the relationships between these hosts as a set of abstract "sentences", like:

```
host01 | is an instance of | file server
file server | provides | file service
host02 | is an instance of | dns server
dns server | provides | dns service
host03 | is an instance of | client workstation
client workstation | requires | file service
client workstation | requires | dns service
```

Each sentence has a subject, a verb, and an object separated by vertical bars (|). Sentences are preparsed into subject, verb, and object *by the user*; no natural language parsing is employed. We call each such sentence a *fact*[1].

From the base facts above, we can intuit several other facts, including:

```
host01 | provides | file service
host02 | provides | dns service
host03 | requires | file service
host03 | requires | dns service
host03 | might depend upon | host01
host03 | might depend upon | host02
```

The last two are subtle: the fact that a host provides something does not mean that it provides it to everyone who requires it.

Suppose that something goes wrong with this network, e.g., `host03` stops responding. The main problem in troubleshooting is to enumerate the entities that can cause the symptoms, rule out causes, and thus determine *where to look* for problems. Obviously, one symptom is that `host03` is broken, which according to the above can be due to a problem with `host03`, a problem with *host01*, a problem with `host02`, or a problem with the network connecting the hosts. If we know more, e.g., that the network is functional but that `host03` DNS service is broken, then this rules out `host01` and points to either `host02` or `host03` as potentially problematic. If we know as well that DNS is functional and `host03`'s configuration for DNS is correct, this points toward `host02`. In other words, the more we know, the more we can eliminate and the narrower the sieve of options becomes.

What our system does is to suggest possibilities in order of approximate likelihood, based upon a description of architecture. For example, in the above it would first report the dependencies upon `host01` and `host02`, which are the "closest" possible causes according to a notion of distance based upon the number of logical inferences required to connect two entities. Then, for each possibility, it can "explain" the relationship between a probable cause and the symptom, all from a description of architecture.

In this trivial case, one can easily do this by hand. With systems of thousands of components, however, the problem becomes more complex. In this paper, we describe a mechanism whereby one can reason about very complex architectures and obtain explanations of complex dependencies between subsystems. We verify our thinking via a simple prototype that serves as a proof of concept. In describing our ideas, we utilize the notation of the prototype, to encourage system administrators to try it out with their data and see what it can do for them.

[1]Functionally, these are just like facts in the logic programming language Prolog, where our fact '`client workstation | requires | file service`' becomes the Prolog fact `requires(client_workstation, file_service)`.

4 Entities and relationships

The key to our solution is a description (cached as a database) describing the *architecture* of the underlying system. The role of this description is to serve as a model of locations and interactions. For this, we appeal to a very old idea: entity-relationship modeling[2]. We describe the network as a graph of named entities and relationships, either manually or by mining the configuration.

Entities in the network are named by strings and can be named at any level, including subnet, host, component, or even software package. Kinds of entities include:

1. physical machines, e.g., '`host01`'.
2. software, e.g., '`RHEL5`', '`Linux`'.
3. services, e.g., '`LDAP`', '`SMTP`', '`HTTP`'.
4. classes of physical items, e.g., '`webserver`', '`mailserver`'.

An entity is a noun whose meaning does not change over time. Nouns can represent classes of things, e.g. '`client workstation`'.

Relationships can be anything, including:

1. Dependencies, including '`requires`' and '`provides`'.
2. Containment, including '`is a part of`', '`is an instance of`'.

[2]We refer specifically to the ER-diagrams utilized in Software Engineering, as opposed to those utilized in database theory. The former describe interactions, while the latter describe functional dependencies.

3. Causality, including '`determines`', '`influences`'.

4. Connectivity, including '`connected to`'.

5. Intent, including '`promises`', '`uses`'.

While entities are nouns, relationships are (usually) verbs. Any invariant relationship can be documented. Verbs can also represent classes of relationships, e.g., '`determines`', which allows many different *kinds* of determination.

Most relationships are directional, i.e., if '`A | is a part of | B`' one cannot conclude that '`B | is a part of | A`', any more than "A is a part of B" would imply that "B is a part of A" in English. However, every relationship corresponds to a unique *inverse relationship* that is simply another predefined formal symbol. If '`A | is a part of | B`', then '`B | has part | A`'. The formal symbol '`has part`' is *defined* as the "inverse" of the formal symbol '`is a part of`'.

5 Relationship to topic maps

One can also think of entities as "topics" and relationships as "associations" between topics in a topic map[24, 23]. This is a kind of generalized ER-model utilized usually in library science[3]. Unlike our simplified ER-model, a topic map describes relationships between three kinds of entities[23]:

1. *Topics* (entities) are analogous to entries in an index of a book.

2. *Associations* (relationships) are analogous to "See also" in a book index.

3. *Occurrences* are are analogous to page numbers in an index, and specify "where" a topic is mentioned.

[3]In this paper, we will concentrate on a simple application of the idea, and not a broader view. While what we do here *can* be utilized with a variety of kinds of data, we concentrate specifically on troubleshooting data and avoid more general problem statements for clarity.

While this work was inspired by initial work in topic maps, the results presented here are more broadly applicable to any ER-model.

The most important thing we draw from topic maps is the *semantics* of our representations. Our ER-diagrams, like topic maps, are intended to *define* entities through their relationships with other entities. Throughout this paper, we will make design decisions that preserve "definition-like" qualities for both entities and relationships. Notably:

1. Entities are static and do not change over time (from the point of view of the reasoning system, inside the formal model).

2. Relationships are static and do not change over time.

3. Definitions are additive and define *facets* of a thing. The total definition of a thing is the union of partial definitions (just as in a dictionary).

Our definition of inverses as verb phrases is consistent with the Cfengine-3 notion of inverse relationships, but differs from the more refined notion of inverses in topic maps. In a topic map, a relationship is a *noun phrase*, and the meanings of sides of the relationship are clarified via what are called "roles". For example, our statement '`cat food`' '`is manufactured by`' '`pet food companies`' would be written in a topic map as "cat food" in role of "product" has relationship "manufacture" to "pet food companies" in role of "manufacturer". We do not need this extra complexity, so we sidestep it. What we lose from this is that our prototype is only compatible with "subject-verb-object" (SVO) natural languages (e.g., like English, French, etc.), as opposed to "subject-object-verb" (SOV) languages (e.g., Arabic, Japanese, etc.). The topic map mechanism handles both SVO and SOV languages, by translating relationships into foreign languages *after* processing.

6 Facts

The first step in utilizing our system is to create (or transform) an appropriate database of suitable facts.

Each fact is a subject-verb-object triple, where subject and object are system entities (or classes), and the verb indicates some kind of relationship between the two entities. Common kinds of facts include class membership, e.g.,

```
couch1 | is an instance of | client workstation
```

class descriptions, e.g.,

```
client workstation | requires | file service
```

and ownership, e.g.,

```
couch1 | is owned by | Alva L. Couch
```

There is no checking as to whether facts make sense in English. The system trusts the user to use relationships that are transitive verbs, and subject and object that are nouns. There is no natural language processing at all in the system. Subject, verb, and object in the fact are syntactic tokens, and nothing more.

6.1 Coding and avoiding hierarchy

Note that the way we specify facts looks very similar to object-oriented modeling, but there is an important difference. Our encoding method is non-hierarchical, in the sense that there is no need to place each host into a hierarchy of relationships. One can do this when convenient, but it is not necessary to the reasoning method. Thus one need not become subject to the "tyranny of classification", in which hierarchy impedes information encoding[22]. Instead, one can freely classify objects into *several* convenient hierarchies, without contradiction. A machine can be a kind of server, a member of an ownership hierarchy, and a kind of client, with no confusion.

Hierarchy is not absent from our system; it is simply *not essential.* Complex entities with many parts are easily modeled via part and subclass relationships, e.g.,

```
dns server | has part | dns local zone information
dns server | has part | dns configuration file
dns server | is an instance of | server
```

with the obvious meanings. Users and privilege can be modeled straightforwardly by thinking of the user as a primary key:

```
Alva | refers to person | Alva L. Couch
Alva | uses shell | /bin/bash
Alva | administers | couch1
Alva | administers | couch2
```

to describe an entity '`Alva`' who administers two machines '`couch1`' and '`couch2`'.

As in the preceding example, one describes multiple relationships by listing instances:

```
Mark | administers | couch1
Alva | administers | couch1
```

means that *both* administer '`couch1`'. Sets of facts are treated as if all are true, i.e., listing two facts implicitly connects them with logical 'and'.

There is no equivalent to logical 'or' in the calculus, nor is there any equivalent to negation. To express that something is one thing or another, one can construct a (synthetic) class '`admin1`' with more than one instance:

```
Mark | is an instance of | admin1
Alva | is an instance of | admin1
admin1 | administers | couch1
```

to denote that *some instance* of the class '`admin1`' administers '`couch1`'.

Note that when a class is used in a fact, an instance is *implied*; no class can "administer" anything. However, this form of disjunction is not exclusive and thus does not preclude that both '`Mark`' and '`Alva`' administer '`couch1`'.

6.2 Modal facts

In our reasoning system, there are very precise meanings of modal expressions in English such as '`X | can serve | Y`' or '`X | might serve | Y`'. The qualifier 'can' implies capability but not intent: '`X | can serve | Y`' means that `X` is capable of serving `Y` but not that `X` is actually serving `Y`. The qualifier 'might' means that there is some (as yet unknown) possibility of a thing. If we say '`X | might serve | Y`', this means that in some *worlds*, `X` serves `Y` and in others, `X` is not known to serve `Y`. These are strength indicators for one's confidence that something is true: 'might' is stronger than 'can'. Neither of these is a temporal distinction; if something might serve something else, it still does or does not serve it, i.e., either '`X |`

`serves | Y`' is a fact, or not. The modal fact encodes the possibility that the non-modal fact is present. Later we will see that modal constructions have a complex interaction with class membership ('`is an instance of`') and structural ('`is a part of`') relationships.

6.3 Pitfalls in declaring facts

The main trap in representing a fact is to represent "too much", so that the implications of a fact far exceed what is intended. Representing "too much" costs the administrator time in sifting through impossible alternatives, while representing "too little" does not depict valid alternatives. Thus, the best practical advice is "when in doubt, specify too little.".

Another way to say this is that one should adopt a "maximum entropy principle" that what is not known for sure is not considered to be known at all.

For example, suppose you do not really know that a client workstation utilizes a specific file server. It would be bad to declare that it uses something that it might not, but fairly harmless to declare that it uses *some* file server, identity unknown. The former will misdirect the reasoning system, while the latter will point out to the reasoning system that this particular facet of configuration is unknown, leading to possibility rather than hard fact. This is what happened in the inferences in the first example, where the relationship '`might depend upon`' indicates that uncertainty.

Another pitfall of encoding facts might best be called the "tyranny of naming". A system entity is often best described by its attributes rather than its name. The name of an object is – at best – nothing more than a (hopefully) unique key. Obviously, it is very bad to use the same name for two distinct things. It might be best, therefore, to use automatically generated unique names for entities, e.g., '`id29394510`', and let attributes of the objects define their physical identities, e.g.,

```
id29394510 | has hostname | couch1
id29394510 | has manufacturer | dell
id29394510 | has serial | 000-123-4567
id29394510 | is owned by | Alva
```

The unique key '`id29394510`' need not be central to a query; one can ask the system what entities influence the (human) '`Alva`', and it can respond with, e.g., hostnames.

A third pitfall of encoding facts is that – because of the simplicity of our representation – relationships often imply the types of their arguments. For example, if one has the fact:

```
host01 | provides | dns service
```

then it is implicit in the relationship '`provides`' that '`host01`' is either a machine or a class of machines. Saying, e.g., that:

```
Alva L. Couch | provides | dns service
```

is thus made somewhat nonsensical – a person can't be a machine or instance of a machine.

In topic maps, this ambiguity is resolved via the concept of *roles*, which determine the types of the subject and object of a relationship. Thus, notating roles as subscripts, one might write:

$$\texttt{host01} \mid {}_{\texttt{machine}}\texttt{provides}_{\texttt{service}} \mid \texttt{dns service}$$

to encode the fact that '`host01`' is an instance of the generic class '`machine`' and '`dns service`' is an instance of the generic class '`service`'. In this case, the relationship between '`host01`' and '`dns service`' is the ternary symbol ${}_{\texttt{machine}}\texttt{provides}_{\texttt{service}}$, where roles are listed on the side to which they apply. In the interest of simplicity, for this paper, roles will remain implicit, but in general, roles can be useful to disambiguate between relationships that are, in fact, different: e.g., ${}_{\texttt{machine}}\texttt{provides}_{\texttt{service}}$ versus ${}_{\texttt{person}}\texttt{provides}_{\texttt{service}}$.

7 Rules

We reason about troubleshooting using a simple calculus of facts and rules that is inspired by – but somewhat different from – ontological reasoning in the semantic web. During ontological reasoning, one connects two entities by looking at how they interact with other entities. Two entities are "similar" if they interact with nearly the same other entities. By contrast, our rules do not concern similarity between entities, but instead derive relationships from relationships, without considering how entities are similar or dissimilar. Rules suggest new facts in several ways, including canonicalization, inverse relationships, transitive relationships, and implications.

7.1 Canonicalization

The purpose of canonicalization is to both save typing and ensure that representations of facts are sufficiently precise to be useful. The relationship '`is a`' is ambiguous; `X | is a | Y` could mean that X is an instance of Y, or that X is a kind of Y. The canonicalization:

```
is a => is an instance of
```

disambiguates between these two alternatives (and more). Canonicalizations are always denoted by "=>", and allow one to utilize a shorthand when writing rules that is expanded later. In the prototype implementation, we employ the following canonicalizations:

```
is a superclass of => has subclass
has superclass => is a subclass of
```

to ensure that we only talk about subclass relationships rather than the equivalent superclass relationships. This is so all class relationships will be comparable.

7.2 Inverses

Inverses allow one to reverse a relationship so that the object switches positions with the subject. The *inverse rule*

```
is an instance of <> has instance
```

means that for every X and Y, if '`X | is an instance of | Y`', then '`Y | has instance | X`' (and vice-versa). The inverse for a relationship is the English phrase that – in English – represents the reversed relationship. Inverses are syntactic, and not semantic. They are always defined, and never inferred.

A few relationships are self-inverse, i.e., '`is a peer of <> is a peer of`', because '`A | is a peer of | B`' exactly when '`B | is a peer of | A`'. Most inverses are simply other ways of stating the same relationship, such as '`is an instance of <> has instance`', which means that '`A | is an instance of | B`' exactly when '`B | has instance | A`'.

The meaning of an inverse in English is incidental to its use. E.g., if you define '`foo <> bar`', then these relationships are inverses, regardless of what they might mean in English; this rule means that if '`Alva | foo | Mark`', then '`Mark | bar | Alva`'.

7.3 Weak transitive rules

Weak transitive rules make connections between previously unconnected objects.

In mathematics, a *transitive relation* is a set of ordered pairs R where for any A, B, and C, if $(A,B) \in R$ and $(B,C) \in R$, then $(A,C) \in R$. In our context, a transitive relation is represented by a verb phrase `R` where for any nouns `A`, `B`, `C`, if `A | R| B` and `B | R | C`, then `A | R | C`. For example, '`is a part of`' is transitive: if A is a part of B, and B is a part of C, then A is always a part of C. Examples of some transitive relations are shown in Table 1.

Each of these relations corresponds to a transitive *rule* in our reasoning system. The relations in the table correspond to the rules

```
is larger than ^ is larger than ^ is larger than
is caused by ^ is caused by ^ is caused by
is the same as ^ is the same as ^ is the same as
depends upon  ^ depends upon  ^ depends upon
is a part of  ^ is a part of  ^ is a part of
is the same as ^ is the same as ^ is the same as
```

where "^" delimits relationships.

However, in our system, there are rules that look somewhat like the former, but whose antecedent and consequent relationships differ from one another. We call these *weak transitive rules*, because they look somewhat like transitive rules but are not. For example, if A is an instance of B and B provides C, then A provides C, meaning that if something is a member of a class that does something, the instance does it too. Some examples of weak transitive rules are listed in Table 2. We notate weak transitive rules in the same way as transitive rules; the rules in the table are notated as

Fact 1	Fact 2	Implies...
A is larger than B	B is larger than C	A is larger than C
A is caused by B	B is caused by C	A is caused by C
A is the same as B	B is the same as C	A is the same as C
A depends upon B	B depends upon C	A depends upon C
A is a part of B	B is a part of C	A is a part of C
A is the same as B	B is the same as C	A is the same as C

Table 1: Transitive relationships correspond to transitive rules.

Fact 1	Fact 2	Implies...
A is an instance of B	B provides C	A provides C
A is an instance of B	B requires C	A requires C
A is larger than B	B might be larger than C	A might be larger than C
A might be larger than B	B is larger than C	A might be larger than C
A depends upon B	B might be influenced by C	A might be influenced by C

Table 2: Weak transitive rules look like transitive rules except that antecedents and consequent differ in some way.

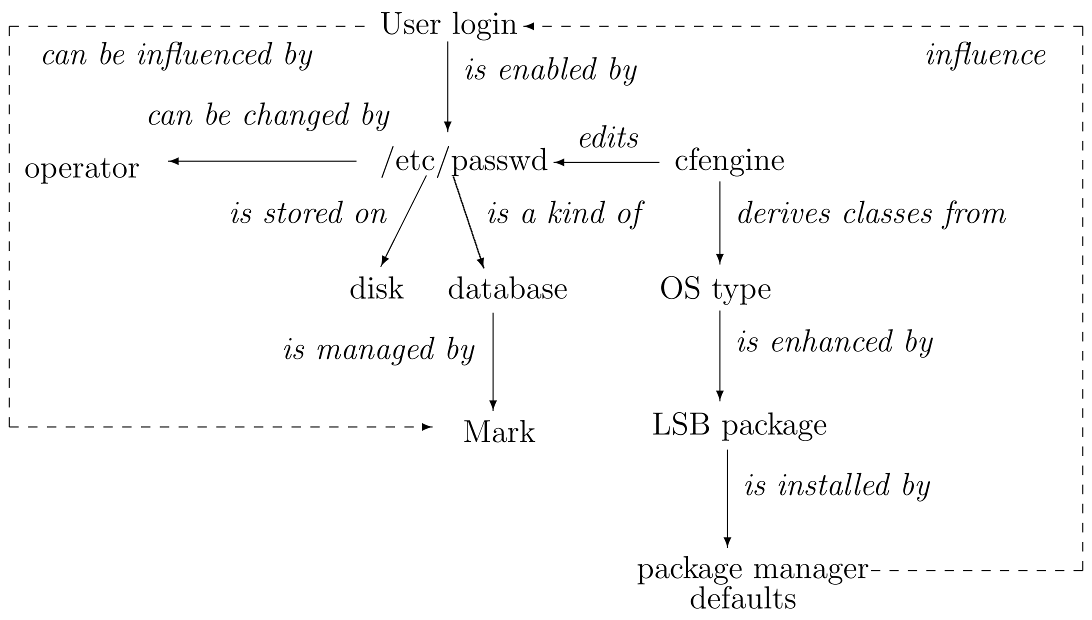


Figure 1: One useful depiction of architecture is a graph in which nodes are entities and arrows represent relationships. Base facts are depicted as solid lines, while two inferred facts are depicted as dashed lines.

```
is an instance of ^ provides ^ provides
is an instance of ^ requires ^ requires
is larger than ^ might be larger than ^ might be
    larger than
might be larger than ^ is larger than ^ might be
    larger than
depends upon ^ might be influenced by ^ might be
    influenced by
```

The point of weak transitive rules is to allow us to codify all ways in which two entities can be connected to one another. Each rule provides one form of connection, and these are the only rules in our system that make new connections.

While transitive rules often result in strong connections (e.g., '`determines`'), weak transitive rules often result in weaker connections (e.g., '`might influence`') that say less about the relationship between the two entities. The point of weak connections is that, even when strong connections do not exist, weaker relationships can guide humans in finding problems. Weak transitivity, as we define it here, offers a simple and measured approach for enumerating possibilities.

7.4 Implications

Implication rules allow one to change the level of abstraction at which reasoning occurs. The *implication rule*

```
provides -> determines
```

means that for every pair of entities '`X`' and '`Y`', if '`X | provides | Y`' then '`X | determines | Y`'. The purpose of implication in our system is to allow one to raise the level of abstraction to a level at which reasoning can occur. If '`Z -> W`', then Z is more specific than W, and W is more abstract (generic) than Z. Specific facts may have no obvious inter-relationship, while their generic equivalents may be obviously related.

For example, consider the facts:

```
host01 | is a file server for | host02
host02 | provides | print service
```

On the surface, these do not have any relationship to each other. But if we translate to a higher level of abstraction via the implications:

```
is a file server for -> influences
provides -> influences
```

then we get the higher-level facts

```
host01 | influences | host02
host02 | influences | print service
```

Then, by the transitive rule:

```
influences ^ influences ^ influences
```

we obtain the new fact

```
host01 | influences | print service
```

which might be quite important to know. In this example – and many others – raising the level of abstraction exposes relationships that are not apparent at lower levels.

7.5 An example of reasoning

Consider the example in Figure 1. A user is unable to log on to a given host, so a diagnostician points the prototype at the entity '`User login`'. The prototype invokes our algorithm to enumerate possibilities. These possibilities are relationships between entities, and not obviously anything that can be logically connected with faults. The human user must evaluate the possibilities.

For instance, if '`User login`' is enabled by the file '`/etc/passwd`', then it '`is influenced by`' it. If '`/etc/passwd`' can be changed by an operator, then it '`can be influenced by`' the operator. If '`/etc/passwd`' is stored on the '`disk`', then it '`is influenced by`' the disk. If '`/etc/passwd`' is a kind of '`database`' and databases are managed by '`Mark`', then '`/etc/passwd | can be influenced by | Mark`'.

But often, more subtle and hidden connections are the real cause of the problem. Here is a problem we have experienced in real practice. A possible but less than obvious cause of a missing user entry in '`/etc/passwd`' is that the file is being managed by an agent (like Cfengine), whose policy applies only to a certain operating system type. That operating system type is only detected in the prescribed manner if the package '`Linux Standard Base`' (LSB) is installed. This in turn depends on the default settings for the package manager in use. In other words, the default settings of the package manager *influence* user login.

What we see in this example is the power of lateral thinking. The system generates alternatives and the administrator rules out each one in turn. The system does not perform logical elimination to find the cause of a fault, but rather the opposite: it enumerates possibilities the administrator may not have considered.

7.6 Rules as shorthands

One purpose of rules in our system is to shorten notation and to allow automatic inference of related facts. We could– in principle – simply enumerate all facts, but this would be a laborious process. One rule suffices as a substitute for writing down many facts.

For example, suppose that there are entities '`LDAP`', '`login privileges`', and '`shell access`', where

```
LDAP ^ can determine ^ login privileges
login privileges ^ can determine ^ shell access
```

Note that these relationships describe potential for interaction, rather than assurance of interaction.

Implications allow us to avoid writing down obvious outcomes. The rather obvious rule

```
can determine -> might determine
```

denotes that the *capability* to do a thing is necessary in order for the *possibility* to do a thing. This rule means that we do not have to write down the facts:

```
LDAP | might determine | login privileges
login privileges | might determine | shell access
```

because these are implied by the facts above and the implication.

Likewise, if there is a transitive rule

```
can determine ^ can determine ^ can determine
```

then we do not have to write down the fact

```
LDAP | can determine | shell access
```

because the latter is a result of that rule and the base facts above.

Rules can also *interact* with each other to produce new rules. The implication

```
can determine -> might determine
```

and the transitive rule

```
can determine ^ can determine ^ can determine
```

together imply the rule

```
can determine ^ can determine ^ might determine
```

because *possibility is weaker than capability.* The rule still applies if the consequent of the rule is weakened.

Moreover, if we have the obvious implication and transitive laws

```
determines -> can determine
can determine ^ can determine ^ can determine
```

then we also can infer the *rules*

```
determines ^ determines ^ can determine
can determine ^ determines ^ can determine
determines ^ can determine ^ can determine
```

because the rule still applies if either of the antecedents are strengthened, and '`determines`' is stronger than '`can determine`'. Also, from

```
determines -> can determine
determines ^ determines ^ determines
```

we can infer that

```
determines ^ determines ^ can determine
```

In general, any rule remains valid if we *strengthen the antecedents* and/or *weaken the consequent.* This is how the prototype actually works internally, and is part of the reason it is efficient.

7.7 Rules as meaning

Another unique aspect of our system is how meaning is imparted to symbols. In most logical systems there is some external model that defines what symbols mean. In our system, *the meaning is the rules.* The interactions between the relationship '`influences`' and other relationships *comprise* its meaning, and two different tokens (e.g., '`influences`' and '`coerces`') are identical whenever their interactions with the other tokens are the same. In other words, ontological equivalence between relationships implies that the relationships have the exact same meaning (with respect to all other relationships considered in the rules).

To understand this (rather subtle) idea, consider the rules

```
determines -> influences
determines -> can determine
can determine -> might determine
influences -> can influence
can influence -> might influence
influences ^ is a part of ^ influences
is a part of ^ influences ^ influences
determines ^ is a part of ^ influences
is a part of ^ determines ^ determines
influences ^ is an instance of ^ might influence
is an instance of ^ influences ^ influences
determines ^ is an instance of ^ might determine
is an instance of ^ determines ^ determines
```

These rules – in a nutshell – encode the principal semantic differences between '`influences`' and '`determines`' with respect to '`is a part of`' and '`is an instance of`'. Note that if one influences an instance of a thing, then one *might influence* all instances (the containing class), or not. If one determines a thing, then one determines its part, but if one determines a thing that is a part of another, one only influences the larger thing. We consider this interaction to be part of the *definition* of the relationships '`determines`' and '`influences`'.

7.8 Classes and structures

The rules for classes and structures deserve special comment. As in object-oriented modeling, a *class* of things shares some common attributes and has instances that have those attributes. Likewise, a *structure* has parts.

For classes, note that

```
is an instance of ^ has attribute ^ has attribute
```

is almost the *definition* of a class. But, perhaps counter-intuitively

```
has attribute ^ is an instance of ^ might have
    attribute
```

because the existence of an attribute in an instance does not mean that it is present in all instances (and thus the class). An instance might be also an instance of a subclass.

Causal relationships have some subtleties. Straightforwardly,

```
is an instance of ^ is determined by ^ is
    determined by
```

because determining all of a class determines its instances. But

```
is determined by ^ is an instance of ^ might be
    determined by
```

because the fact that an instance determines something does not mean that *every* instance determines it.

For structures, note that

```
determines ^ has part ^ determines
```

because if one determines a thing, one determines all parts. But rather obviously,

```
has part ^ determines ^ influences
```

because determining a part does not implicitly determine the whole thing. Again, some subtleties arise:

```
influences ^ has part ^ might influence
has part ^ influences ^ might influence
```

because if something is a part of something larger, and we influence the whole thing, we might or might not touch a specific part. These rules might be considered the definition of '`has part`'.

8 Philosophical concerns

In using our system, several strongly held philosophical decisions become immediately obvious. We designed the system around an open model of knowledge, in the sense that no model is considered to be complete. We also designed the inference system so that knowledge is convergent, in the sense that multiple rounds of inference converge to a fixed-point knowledge base in a finite number of iterations. These decisions give the reasoning system both speed and scalability, but also match the fundamental philosophy of Cfengine upon which the system is based.

8.1 Open knowledge

There are two ways of conceptualizing a knowledge model. A *closed-world model* attempts to describe everything about a system, so that *facts that are absent are assumed to be false.* In an *open-world model*[10], facts describe only what is known, and leave other facets to be described later. When a fact is absent, this does not mean that it is false, but simply that *it is not known to be true* (yet). It might become known to be true in the future, or not. In other words, open world models are ambiguous about whether the lack of a fact implies that it is false.

Like Cfengine, we adhere to an open-world philosophy. We never assume that our knowledge model is complete (or 'closed'), and err on the side of trying not to claim anything that is false. This makes it extremely easy to add information later, once it is known, while leveraging what is known in the meantime. Incompleteness of the architectural model does not hamper its use if we remember that it is incomplete.

8.2 Convergent inference

Another concept we borrow from Cfengine is the notion of convergence[2, 11, 9]. We think of the inference system as creating new facts from old facts, and new rules from old rules. An inference system is convergent if – by some finite number of applications of rules – it achieves a *fixed point state* in which no further operations add new facts or rules[3, 6].

The reason for this philosophical stance is computational. This will allow us (in the future) to code the inferences on a cloud at massive scale, because we can compute resultant facts in advance and then use Map/Reduce to find them[7]. This allows us to turn a logic problem into a database search problem, greatly simplifying implementation.

9 Queries

In the process of troubleshooting, the reasoning system provides guidance as to possibilities by answering several kinds of questions. These questions include what entities are potentially related to a subsystem, and precisely how two given entities are related to one another.

9.1 What are nearby entities?

In a complex system, on average, the most closely related entities to a symptom are most likely to contain the problem. Given an entity or set of entities with symptoms, the system can list those entities with some connection to the set, either via facts or rules. The 'closest' entities are those with some direct connection via a fact or implied fact, while more 'distant' entities are connected via weak transitive laws. The distance between two entities (with respect to some target relationship) is the number of weak transitive laws applied to connect them, plus 1. Entities directly connected by a fact are distance 1 apart, and every application of a weak transitive law adds 1.

Our concept of distance depends upon adopting some target relationship as a goal. Typically, the relationship of interest is '`might influence`', for some very subtle reasons. First, the reasoning system becomes more powerful as the level of abstraction increases. The relationship '`might influence`' is the most abstract relationship that is useful in troubleshooting. While we might actually be more interested in '`determines`', few strong lines of determinism arise in a realistic set of facts. The relationship '`might influence`' has several more concrete versions, specified via the implications

```
determines -> influences
determines -> can determine
can determine -> might determine
influences -> can influence
can influence -> might influence
```

where 'can' implies 'might' because *capability precedes possibility.* Thus '`might influence`' is a "more abstract" relationship than any of '`determines`', '`can determine`', '`might determine`', '`influences`', and '`can influence`', simply because it is more general and applies to more pairs of entities.

Implications are not counted as distance, because all they do is to restate a fact in a *different and less specific language*, and do not change the nature of the fact. By contrast, weak transitive rules add new facts and connections that were never explicit before.

9.2 What is the connection?

The second kind of query explains the connection between two entities. This gives guidance to the troubleshooter trying to debug that connection.

A *story* is a human-readable explanation of why some relationship exists. One can think of it as a "mathematical proof" of the soundness of reasoning.

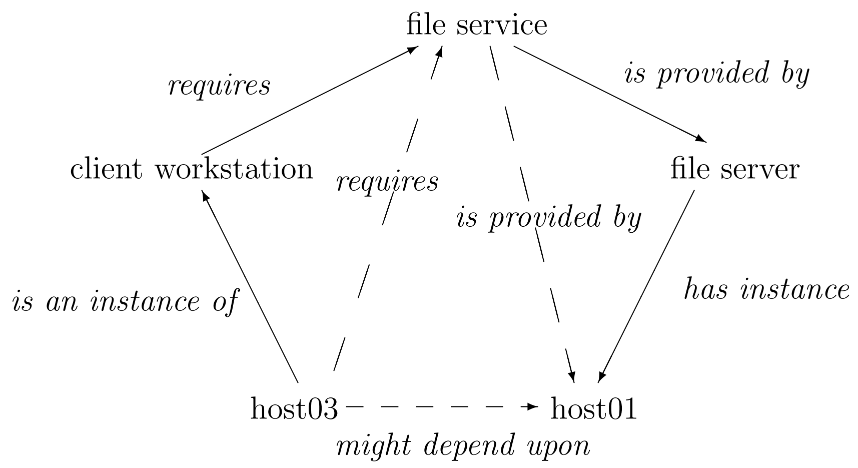


Figure 2: A chord diagram depicts entities in a story in a circle, while relationships are depicted as chord lines of the circle. Solid arrows represent facts, while inferred relationships are represented by dashed arrows. The story explains the dashed horizontal line at the bottom.

One key attribute of our system is its ability to generate easily readable stories.

As a really simple example, suppose we want to know the relationship between '`host01`' and '`host03`' in the initial example. The system utilizes the facts:

```
host01 | is an instance of | file server
file server | provides | file service
host03 | is an instance of | client workstation
client workstation | requires | file service
```

and the weak transitive rules:

```
is an instance of ^ requires ^ requires
is provided by ^ has instance ^ is provided by
requires ^ is provided by ^ might depend upon
```

to infer that:

```
host03 | might depend upon | host01
```

The difference between our system and other forms of logical reasoning is that we have crafted the system so that this inference, once discovered, can be *explained*. An explanation of a relationship is a linear chain of entities and relationships whose combination via rules results in the relationship in question, e.g.,

```
host03 | is an instance of
| client workstation | requires
| file service | provided by
| file server | has instance
| host01
```

We call such an explanation a *story* of the relationship between '`host03`' and '`host01`'. Due to the nature of our rules, every high-level inference corresponds to at least one story (with perhaps many alternatives).

In the previous example, we have avoided depicting one thing, which is the specific set of rule applications that led to the story. In the example, one cannot simply apply rules from top to bottom. The series of rule applications can be depicted in a *chord diagram* (Figure 2), in which the entities are depicted in a circle and the base facts (before reasoning) are depicted as solid lines. The dashed lines (which are all chords of the circle) indicate inferred relationships.

9.3 Lifting and grounding

The preceding example was one of the simplest forms of reasoning of which the system is capable. Often, more trouble must be taken to make reasoning possible and understandable. Architectural descriptions are often incomplete and specified at different levels of abstraction. To cope with this, our system utilizes implication to "lift" facts to a *common level of abstraction or generality* at which reasoning can occur, and then "grounds" that reasoning by expressing the high-level abstract facts in terms of the low-level facts that were their basis.

Consider, e.g., the following quandary:

```
host02 | is an instance of | file server
host03 | is an instance of | client workstation
client workstation | requires | file server
```

What is the real relationship or dependency between 'host02' and 'host03'?

To answer this question, we must proceed to a higher level of abstraction:

```
requires -> is influenced by
```

after which the facts available also include:

```
client workstation | is influenced by | file server
```

and, using the rules

```
is an instance of ^ is influenced by ^ is
    influenced by
is influenced by ^ has instance ^ might be
    influenced by
```

we infer that

```
host03 | might be influenced by | host02
```

from which we infer the story that:

```
host02 | is an instance of
| client workstation | is influenced by
| file server | has instance
| host03
```

but *this is not good enough.* To complete the picture, we "ground" the lifted relationships by replacing them with the concrete relationships that are their subclasses:

```
host02 | is an instance of
| client workstation | requires
| file server | has instance
| host03
```

which "explains" the abstract reasoning in more concrete terms.

10 A prototype

We implemented a prototype reasoning system as a web-based troubleshooting aid. In a troubleshooting situation, a user inputs locations at which symptoms have occurred, and the reasoning system responds with a likely list of other locations that might be the source of the problem. Options are listed in order of inference distance within the reasoning system, i.e., how many transitive rules had to be applied; we have found that this *roughly* corresponds to the strength of coupling between entities. Clicking upon a candidate source "explains" its relationship with the symptoms as a linear chain of dependences. The prototype is written in Perl, and the facts and rules are specified in a text file, using the notation in our examples. The current prototype does everything online. No pre-computed state is kept between queries. Thus the prototype is limited to relatively small examples, e.g., at most a few hundred entities. By contrast, the algorithm itself can be run on clouds, and can scale to arbitrary input sizes.

There are several ways this technology can be used to solve common troubleshooting problems. It can be used to remember details that might be otherwise forgotten, to learn about a new system with which one is unfamiliar, or even to debug one's architectural description of a system. The system does not replace human thought, but rather, assures that known facts are not forgotten.

10.1 Remembering details

First, the system aids a troubleshooter in remembering details or dependencies that might be missed. If one selects a trouble source, the system can respond with those hosts, services, or other entities that might be interfering with that source. For example, inputting '`DNS`' to the system (with relationship '`can influence`') gives a list of things that might affect DNS, in order of distance from DNS.

10.2 Exploring legacy systems

Another typical use case is to learn about legacy systems. System administrators change jobs more frequently than we would like to admit. If a prior administrator has documented the architecture, the new administrator faced with a new system can utilize the data to learn what dependencies are, and to get a feel for how things are connected. For example, one can input two hosts and look for the dependencies between them, or one host and look for the hosts upon which it depends.

10.3 Debugging architectural descriptions

A final and not-so-typical use of the system is to debug architectural descriptions by examining the consequences of those descriptions. This occurs naturally as a result of using the system. When a relationship is explained, the chain of reasoning is presented in terms of the input facts. If an inference is incorrect, the cause must be an invalid input fact, and these are shown for every inference.

11 Critique

This method is not a panacea. It requires careful coding of relationships in order to avoid erroneous conclusions and wasted time. The "inference distance" metric used to determine "most likely" causes could use some refinement. Clearly, there are many shades of '`influences`', from '`greatly influences`' to '`slightly influences`'. The current calculus does not account for shades of meaning.

11.1 Sensitivity to definitions

On a related note, the core causal relationships must be rather rigorously defined in order for the system to work well. Our system "defines" relationships via their interactions with others. Our rules in some sense embody the definitions of our relationships. One must understand the core calculus of meaning in detail in order to properly write new rules.

This means – in turn – that the topics one utilizes to describe the network must be sufficiently understood by the describer to avoid confusion.

11.2 Lack of contradictions

One specific limitation – due to the need to scale to large data sizes – is that contradictions cannot be expressed in the logical system. There is no provision for any equivalent to the statement that "X is not like Y". One can assert similarity, but not difference. Since the associations are purely syntactic, there is no reason – within the system – that data cannot become contradictory by, e.g., asserting two mutually exclusive relationships for an entity.

11.3 Opportunities for further work

Several key questions remain:

1. Is inference distance the best metric of how related two entities are? Are there other better metrics? Is there a concept of relationship that could aid in measuring distance.

2. How should we handle ternary and n-ary relationships?

3. How can we automatically translate common CMDBs (other than Cfknowledge) into a useful form?

4. How can we relate this work to probabilistic methods for discovering connections?

The search will continue for answers to these questions.

12 Lessons learned

Perhaps the most important lesson learned in this work is that naive approaches to making connections between components do not work properly. This paper describes the 14th prototype. Prototypes 1-13 suffered from a variety of serious problems.

First, tracing connections without considering their meaning gives many *false positives* where the discovered connection is not useful or relevant. For example, one might naively infer from

```
ubuntu | has part | kernel
ubuntu | has part | contributed software
```

that somehow the kernel is related to the contributed software, but that is not particularly useful in troubleshooting.

Second, anything short of real computer logic results in *false negatives*. We tried, e.g., to build connections from known connected components to new ones, breadth-first. This resulted in lost relationships, because some causal relationships are inferred from non-causal ones. For example, consider

```
client workstation | contains | compiler
compiler | has instance | gcc compiler
gcc compiler | requires | linker
```

Because containment is not guaranteed to be causal, starting a walk at '`client workstation`' and looking for causal relationships will never get to '`linker`', even though the inference is that

```
client workstation | can require | linker
```

just because of the choice of starting point for the walk and the fact that there are two non-causal links in the sequence. If we start at '`linker`' instead, then the link will be made, but then other connections may be lost. We were unable to "simplify" the logic without losing connections in this manner.

Third, it is extremely important to keep that logic as simple as possible, so that a human can understand it. The simplest representation seems to be a linear chain of components, with their low-level relationships, where the logic is *not* represented in the chain. In the uses we have developed so far, it is the connections themselves – and not the logic by which they are proven to be connected – that is useful to the end-user.

Fourth, the least specific and most abstract forms of causation are the most useful to reason about. The reason for this is somewhat subtle. In the prototype, one specifies a "pivot" relationship, e.g., '`determines`' or '`can determine`' or '`might determine`', and requests the identities of all components having that relationship to the components that exhibit symptoms. This reasoning works best when that pivot is least specific (e.g., '`might determine`'), because our prototypical architecture specification is always incomplete (just like real architectural specifications).

Our prototype and strategy are not "the solution" to troubleshooting, but rather, utilizes a part of available information that was previously ignored. It is not a replacement for discovering causal links or remembering relationships, but makes relationships more difficult to *forget*.

It is our hope that this demonstration of the utility of this kind of information will encourage people to collect more of it, and in turn encourage all system administrators to utilize configuration management systems (either Cfengine or any other) to define configuration in terms of similar high-level architectural models. The ability of system administrators to think in terms of architectural models – and not this work in particular – is what will actually advance the state of the art.

13 Availability

The prototype is freely available from `http://www.cs.tufts.edu/~couch/topics` . We encourage you to try it with your configuration data and share your experiences with us. Your feedback is important and will help to shape the next generation of these tools and approaches.

14 Author Biographies

Alva Couch is an Associate Professor of Computer Science at Tufts University. He is an author of numerous papers on the theory and practice of system administration, and currently serves as Secretary to the USENIX Board of Directors and chair of the LISA steering committee. He can be reached by electronic mail as couch@cs.tufts.edu.

Mark Burgess is a Professor of Network and System Administration at Oslo University College, Norway. He is the author of Cfengine and several books and papers on system administration, as well as chief technical officer of Cfengine AS. He can be reached by electronic mail as Mark.Burgess@iu.hio.no.

15 Acknowledgments

We would like to thank Oslo University College for generously funding Prof. Couch's extended residence at the University, during which time this work was

done. Shepherd Carolyn Rowland provided invaluable feedback.

References

[1] Bergstra, J., and Burgess, M. A static theory of promises. Tech. rep., arXiv:0810.3294v1, 2008.

[2] Burgess, M. A site configuration engine. *Computing systems (MIT Press: Cambridge MA) 8* (1995), 309.

[3] Burgess, M. On the theory of system administration. *Science of Computer Programming 49* (2003), 1.

[4] Burgess, M. An approach to understanding policy based on autonomy and voluntary cooperation. In *IFIP/IEEE 16th international workshop on distributed systems operations and management (DSOM), in LNCS 3775* (2005), pp. 97–108.

[5] Burgess, M. Knowledge management and promises. *Lecture Notes on Computer Science 5637* (2009), 95–107.

[6] Burgess, M., and Couch, A. Autonomic computing approximated by fixed point promises. *Proceedings of the 1st IEEE International Workshop on Modelling Autonomic Communications Environments (MACE); Multicon verlag 2006. ISBN 3-930736-05-5* (2006), 197–222.

[7] Couch, A., and Burgess, M. Human-understandable inference of causal relationships. In *Proceedings of the First International Workshop on Knowledge Management for Future Services and Networks (KMFSAN10)* (2010), Springer.

[8] Couch, A., and Daniels, N. The maelstrom: Network service debugging via "ineffective procedures". *Proceedings of the Fifteenth Systems Administration Conference (LISA XV) (USENIX Association: Berkeley, CA)* (2001), 63.

[9] Couch, A., and Gilfix, M. It's elementary, dear watson: Applying logic programming to convergent system management processes. *Proceedings of the Thirteenth Systems Administration Conference (LISA XIII) (USENIX Association: Berkeley, CA)* (1999), 123.

[10] Couch, A., Hart, J., Idhaw, E., and Kallas, D. Seeking closure in an open world: A behavioural agent approach to configuration management. *Proceedings of the Seventeenth Systems Administration Conference (LISA XVII) (USENIX Association: Berkeley, CA)* (2003), 129.

[11] Couch, A., and Sun, Y. On the algebraic structure of convergence. *LNCS, Proc. 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Heidelberg, Germany* (2003), 28–40.

[12] Couch, A., Wu, N., and Susanto, H. Towards a cost model for system administration. *Proceedings of the Nineteenth Systems Administration Conference (LISA XIX) (USENIX Association: Berkeley, CA)* (2005), 125–141.

[13] Eiter, T., and Gottlob, G. The complexity of logic-based abduction. *J. ACM 42*, 1 (1995), 3–42.

[14] Forum, T. Information framework (sid). website.

[15] Jensen, F. V., Kjæ rulff, U., Kristiansen, B., Langseth, H., Skaanning, C., Vomlel, J., and Vomlelová, M. The sacso methodology for troubleshooting complex systems. *Artif. Intell. Eng. Des. Anal. Manuf. 15*, 4 (2001), 321–333.

[16] Liberatore, P., and Schaerf, M. Compilability of propositional abduction. *ACM Trans. Comput. Logic 8*, 1 (2007), 2.

[17] Mickens, J., Szummer, M., and Narayanan, D. Snitch: interactive decision trees for troubleshooting misconfigurations. In *SYSML'07: Proceedings of the 2nd USENIX workshop on Tackling computer systems problems with machine learning techniques* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–6.

[18] Nelson, K. M., Nadkarni, S., Narayanan, V. K., and Ghods, M. Understanding software operations support expertise: a revealed causal mapping approach. *MIS Q. 24*, 3 (2000), 475–507.

[19] Nelson, K. M., Nelson, H. J., and Armstrong, D. Revealed causal mapping as an evocative method for information systems research. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 7* (Washington, DC, USA, 2000), IEEE Computer Society, p. 7046.

[20] Nordh, G., and Zanuttini, B. What makes propositional abduction tractable. *Artif. Intell. 172*, 10 (2008), 1245–1284.

[21] Parsons, J. An Information Model Based on Classification Theory. *MANAGEMENT SCIENCE 42*, 10 (1996), 1437–1453.

[22] Parsons, J., and Wand, Y. Emancipating instances from the tyranny of classes in information modeling. *ACM Trans. Database Syst. 25*, 2 (2000), 228–268.

[23] Pepper, S. The tao of topic maps. In *Proceedings of XML Europe Conference* (2000).

[24] Pepper, S. *Encyclopedia of Library and Information Sciences.* CRC Press, ISBN 9780849397127, 2009, ch. Topic Maps.

[25] Steinder, M., and Sethi, A. A survey of fault localization techniques in computer networks. *Science of Computer Programming 53* (2003), 165.

[26] Wang, Y.-M., Verbowski, C., Dunagan, J., Chen, Y., Wang, H. J., Yuan, C., and Zhang, Z. Strider: A black-box, state-based approach to change and configuration management and support. In *LISA '03: Proceedings of the 17th USENIX conference on System administration* (Berkeley, CA, USA, 2003), USENIX Association, pp. 159–172.