USENIX Association

Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

Lihue, Hawaii, USA
May 18–21, 2003

Using Runtime Paths for Macroanalysis

Mike Chen, Emre Kıcıman, Anthony Accardi, Armando Fox, Eric Brewer
UC Berkeley, Stanford University, Tellme Networks
{mikechen, brewer}@cs.berkeley.edu, {emrek, fox}@cs.stanford.edu, anthony@tellme.com

Abstract

We introduce macroanalysis*, an approach used to infer the high-level properties of dynamic, distributed systems, and an indispensable tool when faced with tasks where local context and individual component details are insufficient. We present a new methodology,* runtime path analysis*, where paths are traced through software components and then aggregated to understand global system behavior via statistical inference. Our approach treats components as gray boxes and complements existing microanalysis tools, such as code-level debuggers. We use runtime paths to deduce application state, detect failures, and diagnose problems, all in an application-generic fashion. We have explored path-based macroanalysis both in a research setting and as part of a commercial infrastructure at Tellme Networks.*

1. Introduction

Divide and conquer, layering, and replication are fundamental design principles useful for building large, complex systems, such as Internet services, sensor networks, and peer-to-peer (P2P) systems. Such techniques make building large systems tractable, as they improve availability, increase code reuse, and simplify high-level application structure. Unfortunately, debugging ease and fault monitoring do not scale as well, since global context tends to be dispersed across many small components. Building large, complex systems that are reliable, yet maintainable and extensible, remains a challenge.

Existing debugging techniques make use of various microanalysis tools, such as code-level debuggers and application logs. Such tools tend to provide knowledge limited to component internals, or furnish a thread-level perspective, so that the execution context is lost at the thread boundaries. While they provide valuable, localized knowledge, many of these tools fail to capture aggregate component behavior and macro system properties. By way of analogy, microanalysis allows you to see the details of each honeybee, but macroanalysis is needed to understand how the bees interact to keep a beehive functioning.

Various systems have exposed and exploited non-local system context to address performance and resource allocation problems [1, 15, 17]. Macroanalysis makes use of non-local context to improve system management and reliability. This is especially important for large, dynamic systems, where execution context may be distributed across many components.

One key observation we make about dynamic, distributed systems is that most of them have a single system-wide execution path associated with each request that they service. Examples include Internet services that have request/response paths, and P2P systems and sensor networks that have one-way message paths. By tracing these runtime paths, we expose and connect various local contexts dispersed throughout the system. We then use statistics to analyze many of these paths and thereby better understand the system's behavior.

There are several open, challenging problems that can benefit from the high-level system perspective that macroanalysis provides:

Deducing system structure: Systems evolve through both changes to their components and changes in how these components interact. Understanding such inter-component relationships enables developers and operators to anticipate potential conflicts and debug problems. Unfortunately, current techniques for tracking changes in these relationships rely on error-prone, manual documentation, which is infeasible for rapidly changing systems. As systems grow and increase in complexity, we desire automated mechanisms for deducing system structure and tracking its evolution.

Detecting application-level failures: Despite our best efforts at unit testing and quality assurance, services still fail. Worse still, many application-level faults are only seen by end users after deployment, even though systems are constantly monitored for signs of failure. One large commercial service has found that such errors take considerably longer to detect than lower-level failures. The difficulty is that broken or misconfigured components or bad component interactions may only exhibit symptoms at the application level, and the global context required to programmatically diagnose such application-level failures is usually not available.

Diagnosing failures: Failures often manifest themselves far from their root cause. In the extreme case, faults are not detected within the system's boundaries at all and are only visible to external observers. Unfortunately, existing debugging and diagnosis tools have a limited, local view of the system, and thus work best when failures manifest themselves close to the cause. We desire tools that use global failure information to help operators and developers identify the root cause.

The contributions of this paper are:

1. Recognizing that macroanalysis is critical when developing, evolving, and maintaining reliable systems as they grow in size and complexity.
2. A path-based macroanalysis framework, where we first record the components and resources used to service each request, and then use statistical analysis techniques to deduce system structure, detect application-level failures, and diagnose problems.

We stress that macroanalysis complements, and does not replace, traditional component-oriented systems approaches. We often use such tools to flesh out issues identified via macroanalysis. For example, our failure diagnosis can determine the specific requests and component(s) involved in a failure, but identifying the actual cause may require looking at source code or component logs.

The paper is organized as follows: Section 2 develops the runtime path model. Section 3 describes our analysis framework and current status. Section 4 discusses our results addressing the challenging problems above, both in a research setting and as part of a commercial infrastructure at Tellme Networks. We outline future research directions in Section 5 and discuss related work in Section 6.

2. Runtime Paths

We extend the dataflow paths in Scout [15] and Ninja [18] [1] to incorporate runtime properties. A runtime path is the control flow, resources, and performance characteristics associated with servicing a request. Paths can be recorded during runtime by tracing each request through a live system, spanning the system's layers to access direct component and resource dependencies. Each path then provides a vertical slice of the system from a request's perspective.

We use the term "request" in a broad sense to mean a unit of work. This includes both requests that require responses (e.g., HTTP) and those that don't (e.g., one-way messages).

There are two main requirements for a system to support runtime paths. First, it must be possible to associate a unique path with each distinct request. For example, if the same request is handled by different components in different, possibly distributed processes, we must be able to make this connection, perhaps by using a unique ID that travels with the request. Second, it must be possible to report *observations* and associate them with the components that made them. For a pipelined system, a logging mechanism together with knowledge of component request entry and exit would be sufficient.

[1] Scout: "a logical channel through a multi-layered system over which I/O data flows within a single host". Ninja: "a flow of typed data through multiple services across the wide area".

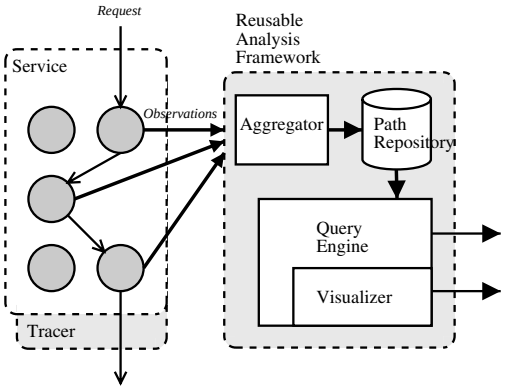

Figure 1. Analysis Framework.

The components make local observations, from which global context is obtained by stringing them together along a runtime path. Each observation contains information about some active component, such as its name, location, timestamp, latency, and arguments. The tracing should be extensible to allow for integration with microanalysis techniques. For example, identifiers could be included in each observation to provide a link with standard application logs.

3. Analysis Framework

The analysis framework consists of five major modules, as illustrated in Figure 1. The *Tracer* tracks requests through the target system, reporting any observations made. Although Tracer is platform-specific, it can be application-generic for platforms that host application components by monitoring requests that enter and exit the components.

The *Aggregator* receives these observations and reconstructs the runtime paths, which the *Path Repository* stores. The *Statistical Declarative Query Engine* handles the data management complexity. It enables us to transparently optimize data storage, and evaluate and update different analysis algorithms. Monitoring and debugging tools should be built on top of this engine. The *Visualization* module helps users understand system behavior. Paths have a natural graph representation: nodes are observations and edges indicate request propagation.

We implemented an extensible Aggregator, a Path Repository, and a Query Engine, with plugins for data clustering and structural anomaly detection. We are currently experimenting with different analysis algorithms.

We built a Tracer for a web server, Jetty, and a clustered J2EE application server, JBoss [10]. The web server inserts a unique request ID into the HTTP request header.

This ID is placed in thread local storage for intra-thread component calls, and passed via a modified RMI library for inter-thread calls. The total code impact was 428 lines in 10 files.

We have been running PetStore, an e-commerce application, and ECperf [20], an industry-standard benchmark for J2EE application servers. Both have a 3-tier architecture consisting of a web server, application components, and a database. The tracing instrumentation is application-generic, so no application changes were necessary.

To measure worst-case performance overhead, we recorded the observations synchronously, using the Java Messaging Service and Java's default serialization methods. We computed a throughput overhead of 16% and an average observation size of 200 bytes – 16 bytes after gzip compression. The compression ratio is high because of redundant text strings, including JVM version identifiers and host names. Using better Java object transport and serialization routines [23] should improve performance.

	Database Tables				
Request Type	Product	Signon	Account	Banner	Inventory
verifysignin	R	R	R		
cart	R			R	R/W
commitorder	R				W
category	R				
search	R			R	
productdetails	R				R/W
newaccount		R	R		
checkout					W

Table 1. An automatically generated partial state dependency table for PetStore. To determine which request types share state, group the rows by common entry under the desired column. For example, the checkout request only writes to the Inventory table, and shares state with three other requests: cart, commitorder, and productdetails.

4. Applying Macroanalysis

4.1. Deducing System Structure

Understanding a service's structure, including the relationship between external requests and the service's internal components and state, enables developers and operators to anticipate potential problems before they upgrade the system. Knowing how components and shared state are used is critical when debugging failed requests. Dependency models have been proposed to improve system reliability and availability [8], but there are few techniques that generate such models automatically.

Key idea: Paths directly capture application structure.

Runtime paths record how a system services real requests, which compares favorably with error-prone, human-generated models and static analysis that predict how the system *might* service such requests. Automatically generated models help developers and operators understand the *actual* behavior of systems under investigation, and can be used as input to recovery mechanisms, such as recursive restarts [4], to reduce mean time to repair.

The Magpie project is using macroanalysis techniques to generate detailed and accurate models of distributed system workloads [14]. While currently used for capacity planning, these models may also be applied to performance debugging, system tuning, and fault diagnosis.

Key idea: Paths associate requests with internal state.

Internet services typically store persistent state in a database to allow for easier front-end scaling. Different requests often share persistent state, but the components handling each request may be unaware of any such sharing. For example, although checkout and login requests may seem like independent HTTP requests, they may share a user profile. A bug during checkout could corrupt this shared state and cause subsequent login failures. Such bugs are difficult to diagnose without an understanding of how various dispersed local contexts depend on each other.

By tracing runtime paths from the web servers, through the application components, and to the databases, we can easily determine how state is shared across requests.

Table 1 shows the mapping between request types and their reads and writes to database tables in PetStore, our desired level of state granularity.

4.2. Detecting Failures using Anomalies

Key idea: Paths often behave differently in failure modes. Hence we can detect failures via changes in path behavior.

Application-level failure detection remains a major challenge today. In practice, quality assurance testing mainly catches simple bugs. Many complex bugs exist in deployed software because of difficulties accurately simulating the workload of a production environment, difficulties modeling the production environment itself, incomplete test coverage, and economic factors. Detecting these bugs in a live system can be difficult, since many bug symptoms are only evident to an end-user, such as incorrect text on a web page.

If we analyze a live system using macroanalysis techniques, however, we can often see secondary effects of failures. Many errors cause runtime paths to end prematurely, while others send paths to less-often used error handlers. Still others, such as fail-stutter faults, simply cause deviations in the latencies of particular path components.

One macroanalysis technique for detecting these changes in path behavior is to search for asymmetries in the interactions among replicated, load-balanced components. Consider an Internet service where all but one of the middle tier nodes are sending queries to a database. Since the middle tier nodes are replicas of one another, it is likely that the

one node's database inactivity is a symptom of some problem. We can extend this idea to treat excessively heavy or light component interaction volume as a sign of failure.

Another technique is to group paths by request type and search for significant deviations in latency or structure.

4.3. Diagnosing Failures through Correlations

Key idea: Runtime paths make the root cause's interaction with a failed request apparent, so that we can quickly explain a failure by a bad component-level or systemic behavior, and can also quickly assess the user visible impact (and hence the priority) of the problem.

When something fails, we want to know why. The challenge here is in tracing externally observed (application-level) failures back to a system fault or root cause.

In practice, we start with sets of suspected failed requests (e.g., reported by users or discovered via anomaly detection) and successful requests, and face the task of identifying the runtime path features underlying any real failures.

The diagnosis task can be cast as a data mining problem, by using data clustering to group the components associated with each failed request's path. Using a previous prototype, Pinpoint [5], we showed that for single component failures the data clustering approach provides a trade-off between accuracy, at 70-90%, and false positives, at 20-40%. This compares favorably with direct fault detection and other automatic analysis methods, which either offer 40% accuracy or many (almost 90%) false positives. Also, we found that paths are vital when dealing with multi-component faults.

Alternatively, the task can be cast as a classification problem in the machine learning domain. Here, the root causes would be the strongest classification rules.

An important benefit of our failure diagnosis approach is that the runtime path data links the logically separate tasks of failure detection and diagnosis, which enables an understanding of mechanisms where the connection between problem causes and symptoms is not otherwise apparent.

4.4. A Commercial Example

Tellme Networks has developed a path-based macro-analysis infrastructure, its *Observation Logs*, to help ensure the high reliability of Tellme's network. Tellme runs voice applications; for the purpose of our current discussion, the system is a telephony network with an Internet back-end, and serviced requests include an audio response to a telephone caller's voice query.

After explaining how runtime paths behave in this example, we will discuss a failure and describe how we used paths to first diagnose the problem, then deduce runtime system structure, and ultimately craft a detection algorithm.

An actual, though simplified, voice response path is illustrated in Figure 2. There are 31 observations in this runtime path, plotted by the relative time at which each observation was made. We call attention to 4 of these, indicating

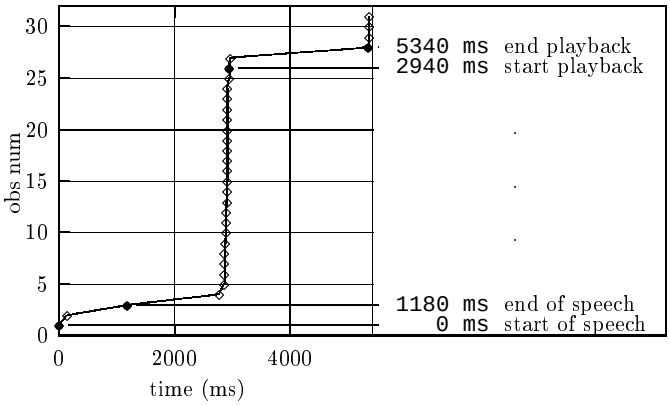

Figure 2. A typical runtime path in Tellme's network.

when the user begins and finishes speaking an utterance and when we start and stop sending audio in response.

The latency profile of the runtime path shown in Figure 2 is close to ideal for this system; most of the time is spent listening to or talking to the user, and the logic connecting the request with the response is rapidly executed. An important metric is the latency between end of speech and start of playback. This 1760 ms delay is perceived by the user, and indicates how quickly the system appears to respond.

Now consider a real network failure. In this case, an isolated process supplying the audio to the machine on the telephony network experienced an internal fault, so that it produced shorter waveforms than the desired ones. This failure is not catastrophic at the system level; a rare, short waveform typically goes unnoticed by the user. The problem is therefore difficult to detect via low-level system monitoring and requires significant application knowledge to handle effectively.

Despite this challenge, we quickly diagnosed this problem using global context in the Observation Logs. An engineer familiar with a particular application noticed a short audio playback and provided details of a phone call that enabled us to quickly locate the relevant runtime path. Once visualized as in Figure 2, a short playback time suggested a truncated waveform. The preceding observations confirmed that a remote process thought it had successfully serviced the audio request, when in fact a rare error had occurred. We identified the particular remote process from the path information, and text logs on that machine subsequently revealed the root cause.

Once we understood the runtime path characteristics for this failure, we were able to query the Observation Logs to detect any similar occurrences throughout Tellme's network. We deduced enough system state to know which components affected which applications, so we could isolate the failing component and assess application impact.

With this new knowledge, we crafted a monitor to use these sub-path latency deviations to detect any future failures in both our production and testing environments.

5. Future Directions

In addition to refining our framework and exploring different algorithms, we are applying our path-based macroanalysis methodology to sensor networks and P2P systems.

Key idea: Violations of macro invariants are signs of system intrusion or buggy implementations. Macroanalysis can help discover invariants, detect violations, and pinpoint the offending components.

Because of the highly distributed and dynamic nature of these systems, many domain-specific macro invariants are difficult to validate using microanalysis or static analysis [7]. Consider an upper bound on the number of hops made during message delivery in a P2P system as a macro invariant. By applying root cause analysis, we can identify peers that incorrectly route messages. In this example, although each node may detect an invariant violation, a diagnosis is difficult without the context contained in a runtime path.

6. Related Work

We consider both microanalysis and macroanalysis work, as well as hybrid approaches.

Macroanalysis Magpie [14] profiles web sites to observe the processing state machine for each HTTP request and to measure request resource consumption (CPU, disk, and network usage) at each stage. The focus is on building probabilistic models of the workload suitable for performance prediction, tuning, and diagnosis.

Microanalysis: Anomaly detection has been used to identify software bugs [7, 22] and to detect intrusions [12], using events based on resource usage [6], system calls [9], and network packets [16]. Paths provide nonlocal context and may make the detection of a new class of intrusions possible.

Hybrid: There are several recent commercial request tracing systems of note. PerformaSure [19] and AppAssure [3] focus on performance diagnosis. IntegriTea [21] focuses on capturing and replaying failure conditions. These systems work with isolated requests, while we aggregate multiple paths and use statistical techniques to infer collective system behavior.

Dynamic program slicing [2] and Whole Program Paths [11] capture dynamic control flow and have been applied to single-process systems analysis.

Some distributed and parallel debuggers support stepping through remote function calls [13]. These tools typically work with individual requests and homogeneous components, and are designed to aid in low-level debugging.

7. Conclusion

Macroanalysis satisfies a need when monitoring and debugging large, complex systems where local context is of insufficient use. To this end, we have presented a runtime path-based approach along with a family of macroanalysis tools that are proving effective in addressing several challenging and important problems. Our method involves dynamically tracing runtime paths through live systems, and recording local observations about performance, interacting components, and resource usage along the way. We subsequently apply data mining techniques to statistically infer aggregate system behavior. This approach is applicable to a large variety of systems, and complements existing microanalysis tools that provide additional insight into individual components.

Our results with distributed Internet systems demonstrate promising progress in 1) deducing system structure and dependencies, 2) detecting failures via path anomalies, and 3) diagnosing problems. We plan to extend our methodology to peer-to-peer systems and sensor networks by validating macro invariants.

References

[1] M. B. Abbott and L. L. Peterson. Increasing Network Throughput by Integrating Protocol Layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, 1993.

[2] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.

[3] Alignment Software. AppAssure, 2002. `http://www.alignmentsoftware.com/`.

[4] G. Candea and A. Fox. Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel. In *HotOS VIII*, 2001.

[5] M. Chen, E. Kıcıman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Symposium on Dependable Networks and Systems (IPDS Track)*, 2002.

[6] G. W. Dunlap, S. T. King, S. Cinar, M. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *OSDI*, 2002.

[7] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *OSDI*, 2000.

[8] B. Gruschke. A New Approach for Event Correlation based on Dependency Graphs. In *5th Workshop of the OpenView University Association*, 1998.

[9] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[10] JBoss.org. JBoss J2EE Application Server, 2001. `http://www.jboss.org`.

[11] J. R. Larus. Whole Program Paths. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, Atlanta, GA, May 1999.

[12] W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In *USENIX Security Symposium*, San Antonio, TX, 1998.

[13] M. S. Meier, K. L. Miller, D. P. Pazel, J. R. Rao, and J. R. Russell. Experiences with Building Distributed Debuggers. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1996.

[14] Microsoft Research. Magpie, 2003. http://research.microsoft.com/projects/magpie/.

[15] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *OSDI*, 1996.

[16] V. Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(23–24):2435–2463, 1999.

[17] J. Reumann, A. Mehra, K. G. Shin, and D. Kandlur. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference*, pages 117–130, June 2000.

[18] S. D. Gribble et al. The Ninja Architecture for Robust Internet-scale Systems and Services. *Computer Networks*, 35(4):473–497, 2001.

[19] Sitraka. PerformaSure, 2002. http://www.sitraka.com/software/performasure/.

[20] Sun Microsystems. ECperf J2EE benchmark, 2001. http://java.sun.com/j2ee/ecperf/.

[21] TeaLeaf Technology. IntegriTea, 2002. http://www.tealeaf.com/.

[22] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[23] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, 2000.