A graphical representation for identifier structure in logs

Ariel Rabkin, Wei Xu*, Avani Wildani†, Armando Fox*, David Patterson* and Randy Katz**

*UC Berkeley** *UC Santa Cruz †*

Abstract

Application console logs are a ubiquitous tool for diagnosing system failures and anomalies. While several techniques exist to interpret logs, describing and assessing log quality remains relatively unexplored. In this paper, we describe an abstract graphical representation of console logs called the *identifier graph* and a visualization based on this representation. Our representation breaks logs into *message types* and *identifier fields* and shows the interrelation between the two.

We describe two applications of this visualization. We apply it to Hadoop logs from two different deployments, showing that we capture important properties of Hadoop's logging as well as relevant differences between the two sites. We also apply our technique to logs from two other systems under development. We show that our representation helps highlight flaws in the underlying application logging.

Index terms: Logging, log analysis, assessment, software development, characterization.

1 Introduction

Logs are an important tool for monitoring and troubleshooting computer system behavior [23, 13]. As a result, there has been substantial work on automated log analysis. Techniques have been proposed for highlighting anomalous messages [1, 18, 20] or patterns of messages [23, 10]. Generally, these techniques are evaluated on proprietary data sets described in fairly general terms. Many recent papers describe the logs in question with a handful of excerpted lines plus a few aggregate statistics [1, 13, 20]. This is unfortunate, because it makes it hard to reason about the relationship between log structure and analysis quality. Many research results are based on measurements taken at particular sites with highly customized software environments; there is often a justifiable reluctance to reveal operational details. There is, however, still a need for better ways to characterize particular logs.

Currently, there are no standard techniques to compare different logs, including those from different sites or different components of the same system, or different levels of logging from the same system. Nor are there good ways to understand the overall structure of an application's logs in ways that help developers spot deficiencies. This paper outlines a new approach to characterizing and visualizing logs designed to help with these sorts of tasks. We target two distinct groups of users: developers seeking to improve their logs and members of the log analysis community seeking more expressive ways to describe the content of particular logs.

Our analysis of logs centers on *identifiers*: variable strings that refer to some particular object or component in the system such as transaction or task IDs. We describe a graph representation of the relationships between log messages, the *identifier graph*. As we show, this representation captures important properties of logs and can indicate possible improvements. This representation lends itself to easy visualization; we describe one approach to visualizing identifier graphs.

Identifiers tie a log message to a specific entity within the system, making them useful for both human and automated debugging: two messages sharing an identifier are highly likely to be related. Recent work has shown that grouping messages by identifier results in substantially more precise anomaly detection [23]. As shown in [10] and [22], finite state machines are a useful technique for interpreting logs, with messages corresponding to state transitions. For systems that include multiple concurrently-executing subprocesses (such as tasks in a MapReduce framework), a separate state machine is often necessary for each subprocess. Identifiers are the glue that ties a message to a particular state machine. Human readers, too, look at the set of messages corresponding to a given subprocess to understand what state it was in when it failed.

Our identifier graphs indicate how much detail a program's logs include about each kind of subprocess in the system. They also indicate which log messages correspond to state transitions in normal execution and which correspond to anomalous behavior.

We focus on application console logs from complex software systems, rather than on whole-system `syslog` data. While our methodology covers both types of logs, applications are typically maintained by a developer community that is smaller and more cohesive than any analog for an entire system. Thus, it is often simpler to change the logging behaviour of a single application than enact system-wide changes.

We begin by describing the structure of this graph and the associated visualization. We follow this in Section 3 by presenting a visual comparison of Hadoop logs from two sites, showing that our visualization brings out interesting and relevant differences. In Section 4, we describe how we have used our visualization technique to find deficiencies in application logs and to guide improvements. Section 5 describes related work. We conclude with an assessment of how broadly applicable this style of analysis is and with a summary of our results.

We have applied our analysis to four sets of logs, from three separate applications, Hadoop [12], SCADS [3], and Chukwa [2]. Hadoop is a mature open-source systems with extensive logging. SCADS and Chukwa are less-polished systems, still in development. We present a summary of these log sources in Table 1.

2 Graphical Representations of Logs

We begin by introducing some terminology, based on that used in [23]. A *log message* is a specific string (generally a single line) printed to a log file by the execution of some *log statement* in a program. The messages from a given statement are all instances of a *message type.* Log messages often include *identifiers*, strings that act as names for some particular object or component in a program or system, usually drawn from a large set. Example identifiers include IP addresses, memory locations, or device names. Identifiers belong to *identifier classes*, which are sets of identifiers that name objects of the same type. The set of IP addresses is an identifier class.

Our visualizations capture a number of properties of an application's logging. Some of these properties are "static", essentially the same from execution to execution. Others are "dynamic", depending strongly on particular executions. We begin by describing how we depict static properties of logs.

2.1 Visualizing static properties

Every message of a given type has the same set of variable fields. As a result, there is a fixed relation between message type and the set of identifier classes referenced by messages of that type. The graph of this relation is the identifier graph. Every identifier class and message type corresponds to a graph node. There is an edge between the nodes corresponding to an identifier class and a message type if the messages of that type include identifiers of that class. Edges are undirected.

We add additional graph edges to represent *subsumption relations.* One identifier class *subsumes* another if the connection between elements of the two can be inferred from the identifier strings themselves. For instance, the URL identifier `http://example.com/page` subsumes the host name identifier `example.com`. In the case of Hadoop, a MapReduce Task ID includes within it the ID of the job that spawned that task. Subsumption of this sort is a semantic property of identifiers and detecting it may require program-specific knowledge, as the Hadoop example shows. The level of semantic insight required is generally quite limited: the only requirement is to understand which substrings in an identifier are themselves identifiers.

In drawing the graph, we use shape to distinguish nodes corresponding to identifier classes (hexagons) from those corresponding to message types (boxes). Identifier class names can be plotted directly on the graph. In both our sample logs and the supercomputer logs examined in [16], average message size ranged from 100 to 250 bytes. As a result, plotting the complete message template tends to result in hard-to-read graphs. Instead, we number each message type and use these numbers to label graph nodes. Separately, our visualizer outputs a numbered list of message templates.

Some message types do not include identifiers. Plotting these on the graph conveys little information, since these nodes would have no edges to other nodes. We therefore omit these singleton message types from the graphs. We do, however, include them in the textual output produced alongside.

Figure 1 is an example log. In that sample, each message has a unique message type. The ID of that type is prefixed to each line. Figure 2 is the corresponding graph. Subsumption relations are marked with dashes.

Producing this graph requires some way to group messages by statement and to group identifiers by class. A number of machine learning techniques have been proposed for this grouping [20, 1, 14, 15, 10, 9]. As an alternative to machine learning, Xu *et al.* [23] use program analysis to generate parsers for matching particular statements. We have used our visualization in combination

	Hadoop JT at M45	Hadoop at Berkeley	SCADS	Chukwa (old/new)
Purpose	MapReduce	MapReduce	Scalable storage	Log collection
Bytes	121 M	20 M	222 K	29 K / 23 K
Messages	685 K	107 K	1607	429/ 248
Identifier types	5	8	7	4/ 4
Message Types	55	51	41	41/33

Table 1: Our sample logs

```
1:  JobTracker: Adding task 'attempt_200911091331_0010_m_000002_0' to
 tip task_200911091331_0010_m_000002, for tracker tracker_r25
2:  JobInProgress: Choosing data-local task task_200911091331_0010_m_000002
3:  JobInProgress: Task 'attempt_200911091331_0010_m_000002_0' has
 completed task_200911091331_0010_m_000002 successfully.
```

Figure 1: Sample of Hadoop JobTracker log, edited for clarity

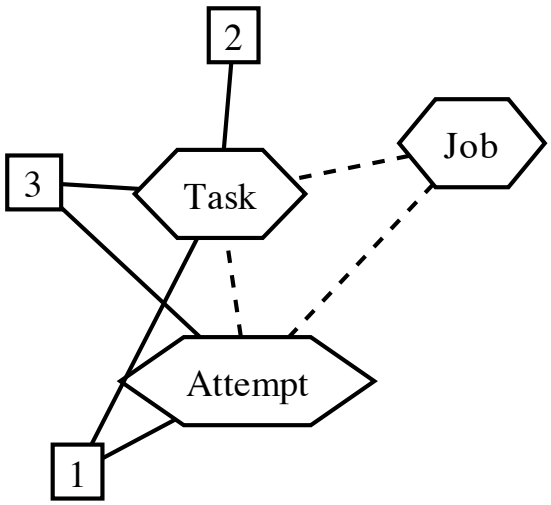

Figure 2: Example message graph corresponding to log shown in Figure 1.

with the parser generator developed by Xu *et al.* This produced usable graphs, however, identifier labels had to be adjusted by hand, since the automatically generated ones were unwieldy.

Not all identifiers are equally useful. For example, we have found that local files and IP addresses are often used by programs in several different contexts, sometimes referring to unrelated entities, making these identifiers less helpful for interpreting the log structure. As a result, we generally configure our visualization to ignore these identifier classes.

2.2 Dynamic properties

Above, we defined the basic structure of our graphs. Here, we discuss how we indicate additional information about frequency and ubiquity of messages. This information is "dynamic," since it depends on the particular execution of the program being logged.

We use pen thickness to convey the relative frequency of a given message type or identifier class. As the message or identifier associated with a node becomes more common, we use thicker lines to render its boundary. It often happens that the frequency of different messages and identifiers in a given log varies by several orders of magnitude. To avoid drowning out large relative distinctions in less-frequent nodes, we apply a form of gamma correction. Let k be the number of instances of a given message type or identifier and let max be the most-frequent such instance. Then line thickness is proportional to $(\frac{k}{max})^{\gamma}$. We find that gamma values between 0.5 and 0.75 work well. On color displays, we shift node color from blue to red, in proportion to line thickness.

Not all identifiers appear in equivalent sets of messages. For instance, all Hadoop Task IDs are associated with a "task start", but some are associated with "normal completion" and others with error conditions. To capture this distinction, we introduce a function we call the *ubiquity* of a message type for an identifier class. The ubiquity is the fraction of identifiers of that class associated with the given message type. So if every identifier of class C is associated with a message of a given type, that message type would have a ubiquity of 1 for class C. And if only a handful of C-identifiers were associated with a message type, its ubiquity would be low. This ubiquity function is effectively the inverse document frequency, where each identifier is a term, and each message is a

document.

Ubiquity conveys how anomalous a given message is relative to the occurrence rate of the associated identifier; it is unrelated to the overall frequency of that message type. If a given error message appears many times, always referring to a single identifier out of a large class, that message would be very common but have low ubiquity. For visualization purposes, we indicate ubiquity by making edge weights proportional to ubiquity: heavier lines connect ubiquitous messages with the associated identifier class.

Sometimes, an identifier conceptually related to a given log message will be found in a previous or subsequent log message. It would be possible to add additional graph edges between messages based on their proximity in the log and in time. However, such connections are often spurious, particularly in highly concurrent systems. Combining our technique with probabilistic detection of message relations is left as future work.

2.3 Analysis

As discussed in the introduction and observed in [10, 22], application logs often have the structure of a set of concurrent state machines, with a state machine for each program component; messages are logged on state transitions. Our identifier graph is approximately the dual of these state machine graphs: transitions in the state machine (messages) become nodes in the identifier graph. States in the state machine correspond to edges in the identifier graph, since each edge indicates that the state machine (identifier node) could be in a particular state (message node). Our representation goes farther, however, since it incorporates the messages that correspond to transitions in multiple linked state machines.

The connectedness of the graph corresponds to the complexity of the underlying logging. A log message describing an interaction between components will appear on the graph as a node with multiple edges. The number of messages linked to an identifier indicates how much detail the logging gives about the activities of that kind of entity (the number of state machine states, perhaps). The number of identifier classes gives a sense how many different aspects of program behavior the logging records.

3 Characterizing and comparing logs

In the introduction, we set two goals: characterizing the logs from a program and finding omissions or weaknesses. In this section, we describe how our graphs achieve the first of those goals: characterizing logs in ways that facilitate comparison.

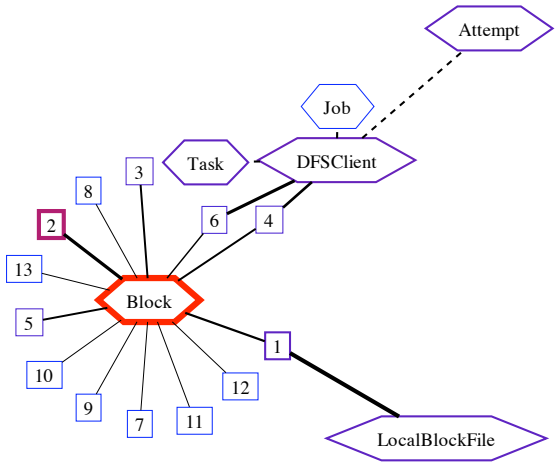

Figure 3: Hadoop DataNode logs in M45 cluster

Hadoop is an open-source implementation of MapReduce and the Google File System architecture [4, 7]. We looked at logs from two different Hadoop deployments: a 15-node Hadoop cluster at our institution and the 4000-node "M45" cluster operated by Yahoo!, inc and used by academics at many different institutions. We show how our visualization brings out important characteristics of Hadoop's logging in a workload-independent way, while also highlighting interesting and relevant differences between the two sites.

Hadoop is a large, mature, well-engineered system with extensive logging. Hadoop logs are identifier-rich, and most identifiers are easily identified lexically. For instance, all job IDs match the regular expression `job_[0-9]+_[0-9]+`. Other identifiers are similar, consisting of a sigil specifying the identifier class, followed by several numeric fields, separated by underscores. Hadoop's identifiers make heavy use of subsumption— `task_a_b` is a task required by `job_a`, and `attempt_a_b_c` is an attempt by a particular node to perform task `task_a_b`.

Figure 3 shows the identifier graph for a Hadoop DataNode in the M45 cluster. (DataNodes are the worker nodes for the HDFS filesystem.) This graph illustrates several ways that the identifier graph characterizes logs. One message type, number 2, is both commonly occurring and ubiquitous for blocks. These messages are block verification reports, which automatically generated periodically for every block, making them both ubiquitous and common. More interestingly, no other message type is particularly common or ubiquitous: all other message types are both rare (boxes drawn with thin blue lines) and also not ubiquitous (lines to "Block" drawn thin). This is a clue that we are seeing verification reports for blocks without having seen either a read or a write for them.

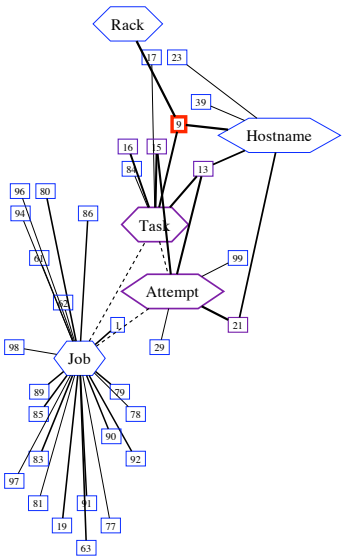

Figure 4: Hadoop JobTracker logs in M45 cluster.

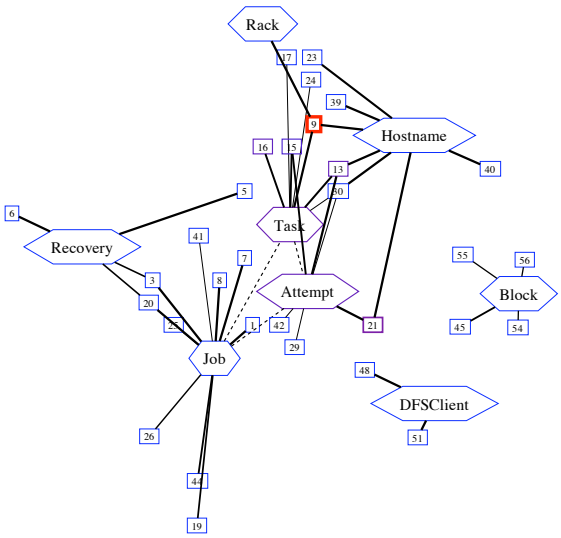

Figure 5: Hadoop JobTracker logs at Berkeley.

Given a complete log going back to the initial start-up of the DataNode, we would expect block creation messages to also be ubiquitous for block IDs.

Some blocks are associated with an identified DFS client, the external process reading or writing a block. These DFSClient IDs sometimes include within them the name of a task attempt, allowing the read or write to be associated back to a task and a job via subsumption relations. The figure compactly expresses the fact that block reads can be tied back to a MapReduce job.

Figure 4 shows the graph for the JobTracker in the M45 cluster. (JobTrackers are the master nodes for Hadoop MapReduce.) A few relevant conclusions can be drawn: Message type 9 is substantially the most frequent. It lists each host and rack holding a copy of the input for a given task. Substantially more message classes refer to Jobs as compared to tasks or attempts. Inspecting the list of messages (not shown here) shows that these job-related messages refer to recoverable errors in a submitted job.

Figure 5 is the equivalent graph of JobTracker logs from our cluster at Berkeley. Again, the most common message type is the one linking tasks to their input locations. However, the combination of different Hadoop versions and different operational circumstances leads to our Berkeley cluster having a different spectrum of error messages. A smaller range of job-related errors is seen. At Berkeley, errors sometimes come from the underlying filesystem layer, resulting in errors referencing Blocks and DFSClients. Note that, due to a deficiency in the Hadoop logs, no messages indicate the job associated with these errors.

Our tool caches the mapping from message template to message type IDs across different invocations. Messages in Figures 4 and 5 with the same templates will have the same ID numbers. This means that numbering will generally not be consecutive in any particular graph.

4 Improving Logs

In this section, we discuss our experiences using visualizations to guide improvements to the logging of two different systems under development, SCADS and Chukwa.

Log statements are often added to programs to isolate specific problems or to trace particular activities in some component. As a result, context and information not directly needed for the task at hand is not reliably included. This naturally leads to programs that output a hodgepodge of logs of varying quality that may not deliver a clear message when a portion of the complete system fails. There is seldom a systematic design of a system's logging, and there is usually no attempt to harmonize the overall structure of the logging after development is completed [23]. Identifiers are often added, after the fact, as developers find that they would be helpful [10].

We focus on three specific problems with logging. Sometimes, identifiers that would have been useful are not logged. In other cases, the same entity is identified in two different ways. Also, sometimes the same identifier is used ambiguously, to label two different entities.

Our graph-based visualization highlights all three kinds of defects. A node with insufficient identifiers will be missing edges that the semantics of the program indicate should exist. A node that has no identifiers will

appear as a singleton, a single point with no graph edges. If entities are identified inconsistently and no messages relate the different naming schemes, then messages describing the same entity will be unconnected in the graph. Ambiguous usage will show up as unexpected ratios of messages to identifiers, visible as anomalous ubiquity ratios.

4.1 SCADS

SCADS, the Scalable Consistency-Adjustable Data Store, is an ongoing research project developing a low-latency data store with performance-safe queries [3]. SCADS includes both a distributed key-value store and the Director, a centralized controller responsible for making data placement decisions and for starting and stopping storage nodes in response to workload. We obtained console logs from the SCADS Director, produced during various experiments. The SCADS logs have significant structure. Each *high-level* Director action triggers a series of log messages, as the action is initiated and completed. Each Director action, in turn, triggers a series of *low-level* actions, each with start and finish messages.

Figure 6 shows the identifier graph for a sample SCADS log. There are a handful of identifier types, each corresponding to a system action (effectively, a transaction or operation ID.) For each type of action, there is a fixed set of associated messages. Each message type is ubiquitous for its associated identifier class. In other words, every action of a given type has the same set of messages associated with it. Low-level actions have just a start and stop message; each type of higher-level actions has several message types associated with it. No observed message types indicate anomalous events.

Our analysis helped us find a subtle ambiguity in the SCADS logs. For each type of action, we had expected proportional numbers of messages and identifiers. There should be one start and stop for each action. Instead, we found that sometimes there were several times more messages associated with a given action type than unique action identifiers. On the visualization, this mismatch shows up as a difference in node color and boundary thickness between the identifier type and associated message types in a graph component.

Inspecting some of the unusually common statements helped us spot the problem: an ambiguity in how actions were named. High- and low-level actions were both identified by the hosts and data items in question. This information is insufficient to uniquely identify an action. More than one action might involve the same participating hosts and the same data. As a result, there will sometimes be identical messages corresponding to distinct actions. This naming approach would frustrate

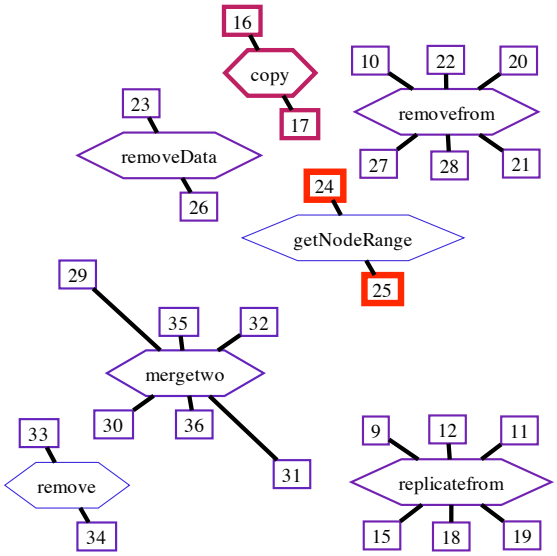

Figure 6: SCADS identifier graph. Larger connected components are high-level actions, smaller components are low-level.

several kinds of log analysis: it makes it impossible to compute metrics like average task run time unambiguously from the logs. The SCADS developers confirmed this as a bug, and they intend to add time-stamps or some other additional disambiguating information to their action identifiers. Note that a per-message timestamp is insufficient here, because several concurrent actions can take place with the same participants.

The statement graph also illustrates a second problem with the SCADS logs. Even though there is a well-defined correspondence between high- and low-level actions, the logs do not reflect this. No messages link low- and high-level actions. In our taxonomy above, this qualifies as an absence of expected identifiers. This is problematic because if an unexpected low-level action appears in the logs, there is no straightforward way to find out what high-level task spawned it. Likewise, there is no convenient way to see which low-level actions were spawned by a given high-level task. The SCADS developers hope to fix this problem in their next release by explicitly logging the dependence between a low-level action and the high-level action that caused it.

4.2 Chukwa

Chukwa is an open source log collection and processing framework, currently in production use at several sites [2]. It is a fairly substantial distributed system and produces its own console logs describing what data sources are being monitored and the flow of data through

the system. In Chukwa, data is produced by system components called *adaptors*. Like many open-source efforts, Chukwa is the work of several developers, each of whom instrumented the portion of the system they were working on. As a result, Chukwa's logging uses several different schemes for referring to adaptors. In some places, adaptors are referred to by an ID string, and in other places by their functional description. In the first version of Chukwa we looked at, the mapping between these two naming styles was never explicitly recorded: there was no way, given the logs, to know the function of an adaptor, given the ID.

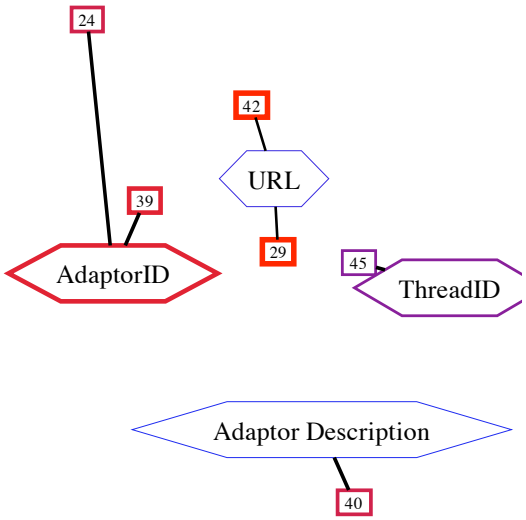

Figure 7: Chukwa identifier graph before improvement

We obtained logs from a standard Chukwa test case and applied our analysis. The logs had several types of identifiers but never recorded the relationships between them. We show the statement graph in Figure 7. In addition, the logs included a large number of singleton messages, *i.e.* messages that lack any sort of identifier. These singletons have been omitted in Figures 7 and 8, but they are listed in the textual output produced by our analysis tool.

In the case of Chukwa, we were able to go beyond merely spotting problems in the logging and submitted patches to fix them. Improving the Chukwa logs required several changes. We added identifiers to several statements that previously omitted them. Chukwa uses HTTP Post requests to send data; a new identifier type, "Post", was added to help group together messages involved in sending a specific bundle of data across the network. A new message was added to link together adaptor descriptions and IDs. We removed a number of obsolete statements. The Chukwa developers agreed that recording

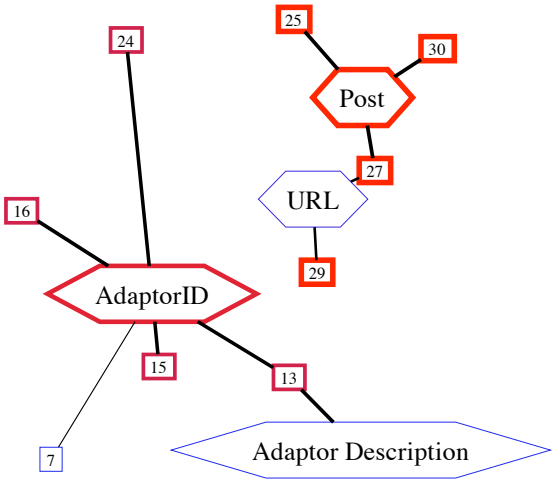

Figure 8: Chukwa identifier graph after improvement

thread IDs no longer made sense, and the log statement printing them was removed.

Running the same test scenario after the improvements to logging resulted in the statement graph shown in Figure 8. The visualization highlights the fact that the set of identifiers in the logs has been changed. The revised logging breaks into two clear halves: some messages are about the life-cycles of adaptors (data collection), while others are about sending data across the network. This split structure accurately reflects the design of Chukwa. The two halves are indeed separate, with a data buffer as their sole connection.

It is now possible to follow the life-cycle of an adaptor, whereas previously the initial parameters were not visibly connected to the ID. This makes debugging adaptor problems significantly easier. More log messages are in the same component of the graph as URLs: instead of simply logging the beginning and end of a data transfer, the revised logging records additional details, such as the number of chunks of data being sent. This makes it easier for developers to tie transmission errors back to a particular receiving host and to understand the behavior of the retry-and-failover code in Chukwa. The changes received emphatic and enthusiastic responses from the Chukwa development community, and they have been incorporated into the latest release.

5 Related work

We summarize three areas of related work: efforts to characterize logs, efforts to visualize logs, and efforts to help developers produce more useful logs.

Surprisingly few papers have set out, as a primary goal, to describe actual real-world log data. One notable effort in this regard is [16].

Visualizing logs is a natural solution to understanding the immense quantity of seemingly disconnected log messages that a typical application produces. Guzdial *et al.* [11] motivated the log visualization in 1994, outlining a list of theoretical benefits from displaying log information with a focus on noticing temporal correlations. In [20], Stearley describes the Sisyphus toolkit for log analysis, which includes an interactive log browser for examining the clustered results. Mielog [21] and SEESOFT [8] use visualization for compact data representation, allowing the user to quickly scan a large log for aberrations and line-similarities. Log lines are identified by message type but there is no broad sense of correlation between message types.

Our work is distinct because we show that we can make useful assessments of log quality without using message adjacency, which is unreliable in logs from highly concurrent system.

Web logs have particularly benefited from visualization. Webviz [17] provides a user-filterable interface for examining the connections between webpag accesses. Like us, they use a node-edge model and convey information in line thicknesis, but their goal is to track access patterns. A three-dimensional variant of web access tracking is presented by Chi *et al.*.

In this paper, we seek not merely to describe and visualize logs, but to advise developers on how to improve them. We are aware of two other papers on this topic. Cinque *et al.* [6] argue that logs should contain a start/end pair for each interaction between system components. Our approach is compatible with this policy but does not require it. Their criterion effectively says that for every identified interaction, there should be a start and a stop message, each ubiquitous for the interaction class. The SCADS example above shows how our visualization can highlight violations of this policy.

Another proscriptive paper is by Salfner *et al.* [19]. The authors offer a number of rules intended to make logs more convenient for automated analysis, such as requiring timestamps on every message, structuring messages as a set of key-value pairs, and categorizing statements hierarchically. Regardless of how useful these constraints on logging would be for administrators and analysis developers, they are burdensome for application developers and have had limited uptake.

6 Conclusion

Logging in many applications exists primarily to help debug during development rather than to spot operational problems. As a result, application logs may lack information that would have been very valuable in tracking down problems in production. Our identifier graph representation illustrates both how much information is recorded about each entity in a system and how these entities are related. Thus, our graph helps developers reason about their logging and spot gaps. It also helps developers compare the overall logging structure of different applications or different configurations of the same application.

This paper has described an of abstract representation for application console logs, the identifier graph. We have argued that these graphs helps developers spot several related classes of deficiencies in logs: identifiers that are absent, inconsistent, or ambiguous. Each of these defects implies a relationship between system entities that was determinable at runtime, but which was not recorded in the logs and which cannot be easily reconstructed afterwards. We have given examples of these deficiencies in several real systems.

Our current graph construction is fairly simple. We expect that more sophisticated models can enhance the results presented here. In particular, some cases of ambiguity and inconsistency might be resolved using timestamps, an approach we hope to explore in the future.

The approach taken in this paper has been to construct an abstract graphical representation of application logs and to assess the logs by analyzing and reasoning about the model. We believe that this approach is more broadly applicable. Analyzing and characterizing graphs is a well-developed field, and representing logs in this way lets us reuse a great deal of theory and many existing tools. We expect that this characterization framework will enable others to create more useful logs as well as better, more informed log analyses.

Acknowledgements

This research was supported by California MICRO, California Discovery and the following Berkeley RAD Lab sponsors: Sun Microsystems, Google, Microsoft, Amazon, Cisco, Cloudera, eBay, Facebook, Fujitsu, HP, Intel, NetApp, SAP, VMware, and Yahoo!.

We thank Peter Alvaro, Michael Armbrust, Peter Bodik, Rean Griffith, and Beth Trushkowsky for commenting on drafts of this paper.

References

[1] 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.

[2] Apache Software Foundation. Chukwa. `http://hadoop.apache.org/chukwa/`, November 2009.

[3] 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.

[4] D. Borthakur. HDFS Architecture. `http://hadoop.apache.org/common/docs/r0.20.0/hdfs\_design.html`, April 2009.

[5] 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.

[6] M. Cinque, D. Cotroneo, and A. Pecchia. A logging approach for effective dependability evaluation of complex systems. In *Proceedings of the 2009 Second International Conference on Dependability*, Athens/Glyfada, Greece, 2009. IEEE.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, Volume 51(Issue 1):107–113, 2008.

[8] S. Eick, M. Nelson, and J. Schmidt. Graphical analysis of computer log files. *Communications of the ACM*, 37(12):50–56, 1994.

[9] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual Symposium on Principles of Programming Languages*, January 2008.

[10] Q. Fu, J.-G. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM'09)*, Washington, DC, 2009.

[11] M. Guzdial, P. Santos, A. Badre, S. Hudson, and M. Gray. Analyzing and visualizing log files: A computational science of usability. In *HCI Consortium Workshop*, 1994.

[12] Hadoop. `http://hadoop.apache.org/`.

[13] W. Jiang, C. Hu, S. Pasupathy, A. Kanevsky, Z. Li, and Y. Zhou. Understanding Customer Problem Troubleshooting from Storage System Logs. In *7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, February 2009.

[14] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution: Research and Practice*, pages 249–267, 2008.

[15] 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.

[16] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *Dependible Systems and Networks*, 2007.

[17] J. Pitkow and K. Bharat. WEBVIZ: A tool for World-Wide Web access log analysis. *Comunicación*, 1996.

[18] S. Sabato, E. Yom-Tov, and A. Tsherniak. Analyzing system logs: A new view of what's important. In *Second Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML '07)*, Cambridge, MA, 2007.

[19] F. Salfner, S. Tschirpke, and M. Malek. Comprehensive Logfiles for Autonomic Systems. In *International Parallel and Distributed Processing Symposium (IPDPS 2004)*, April 2004.

[20] J. Stearley. Towards informatic analysis of syslogs. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[21] T. Takada and H. Koike. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *Proc. USENIX Conf. on System Administration*, pages 133–144. Citeseer, 2002.

[22] J. Tan, X. Pan, S. Kavulya, R. Gandhi, and P. Narasimhan. SALSA: Analyzing Logs as StAte Machines. In *First USENIX Workshop on Analysis of System Logs (WASL '08)*, San Diego, CA, December 2008.

[23] 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.