Provenance for System Troubleshooting

Marc Chiarini
Harvard University

Abstract

System administrators use a variety of techniques to track down and repair (or avoid) problems that occur in the systems under their purview. Reviewing log files, cross-correlating events on different machines, establishing liveness and performance monitors, and automating configuration procedures are just a few of the approaches used to stave off entropy. These efforts are often stymied by the presence of hidden dependencies between components in a system (e.g., processes, pipes, files, etc). In this paper we argue that system-level provenance can help expose these dependencies, giving system administrators a more complete picture of component interactions thus easing the task of troubleshooting.

1 Introduction

Most highly experienced system administrators can remember a time in their career when they were virtually clueless about the configuration of their systems. Whether learning on the job as a junior sysadmin or walking into a brand new infrastructure, nobody is ever handed a comprehensive guide to "the way things work around here." Instead, sysadmins must slowly develop a *mental model* of the systems in their care [4]. They study existing documentation and Internet sources, solicit expert advice, explore component interactions, and much more. While this process is valuable in the long run, it is also time-consuming and error prone, and competes with the efficiency of whatever task is at hand (e.g., tracking down and fixing the root causes of problems). Additionally, mental models are developed on an as-needed basis and fail to account for hidden dependencies between system components, resulting in large gaps and inaccuracies.

This paper explores how system-level provenance can effectively expose hidden dependencies, improve mental models, and help improve the troubleshooting process for system administrators.

2 Dependencies

Efficient troubleshooting requires mental models that are sufficiently accurate and complete to suggest proper courses of action. One part of a good mental model is a map of dependencies between the various components in a system. At a high level, *components* can be thought of as subsystems (e.g., the web subsystem depends upon the networking subsystem). At the lowest level of abstraction, components consist of programs and their individual configuration parameters. At this level, a good mental model maps how parameter changes affect a program's dependencies.

For the purposes of this research, we loosely define *dependency* as the relationship created when information must be transmitted from one component to another in order for the receiver of that information to function correctly. For example, when a process loads a library, functions necessary to the core behavior of the process are transmitted to it from a file. The process is *dependent* on the library being loaded into some part of memory and being made accessible. Likewise, when Apache starts, it reads necessary parameters from an external source of information (e.g., `httpd.conf`). Furthermore, Apache depends upon its runtime environment to properly specify the location of `httpd.conf`.

The PASS project [7] currently collects system-level provenance from inside a running kernel and builds a directed acyclic graph that describes ancestral relationships between files, pipes, and processes[1]. The graph can then be queried using a custom query language, called PQL [3]. PQL operates on a semi-structured data model that allows us to ask questions about ancestors and descendants as well as about paths in the graph, specified as regular expressions. We can issue simple queries such as "show me all objects with at least three immediate ancestors" and complex queries such as "find all objects

[1] This includes variables and other information about the environment in which they execute.

that result from the same (or similar) sequence of transformations", which is a path finding query.

If we think of files, pipes, and processes as system components between which information flows, then the provenance graph can be viewed as a graph of *potential dependencies*. Nodes of the graph represent components and edges represent a "may depend upon" relationship from one component to another. In practical terms, for a process P that reads from a file F, there exists a directed edge from the descendant P to the ancestor F. Likewise, if the same process writes to a pipe I, an edge from I to P will be generated in the graph. The graph describes only potential dependencies, because in the absence of code and dataflow analysis, we cannot be certain that any descendant depends upon its ancestor to *function correctly*.

3 Troubleshooting

In the past decade, there has been exciting research on improving failure diagnosis for system administrators. Some approaches use visualization to help operators rapidly detect and diagnose problems [10]. Others use event correlation in log-file analysis to identify extant and potential problems [8]. Wang et al. [11, 12] use comparisons of current system configurations against golden state configurations that have been generated via statistical analysis of machine populations. None of these contributes very much to exposing complex system dependencies.

In the absence of formal documentation, sysadmins have few resources for determining the dependencies of a program. There exist tools that support static extraction of dependencies via analysis of package management repositories [5] and program images [9], but these have quite limited capabilities. Brown et al. [1] are able to automatically construct operational dependency models by actively perturbing live systems, but this may be dangerous in a production environment.

Although one may assume that documentation is available for general-use tools, many organizations develop in-house solutions. When these solutions are intended for internal use only, there is little economic incentive to create polished user interfaces or comprehensive documentation; tools must simply be "good enough." As the number of internal libraries, scripts, and programs increases, making changes to the system becomes increasingly difficult. For example, deleting old libraries becomes virtually impossible when sysadmins have no knowledge of what programs utilize which libraries. The complexity of these poorly understood systems will continue to grow without bound as long as they are actively developed. Sysadmins in this situation would benefit greatly from a comprehensive and explorable graph of component dependencies.

As suggested earlier, a clear and accurate system model is paramount to troubleshooting. Although sysadmins already troubleshoot in the absence of such models, their efforts have been significantly hindered by complexity. When something fails in a system, knowing where to look first is usually a "gimme". Under progressively greater pressure, knowing where to look second, third, fourth, and so on, requires experience and perseverance.

For example, in most UNIX distributions, the resolver, which sends DNS queries to translate names into IP addresses, loads its configuration from the file `/etc/resolv.conf`. Traditionally, this file was edited manually. In modern distributions such as Ubuntu, the file is now automatically generated and modified by the `NetworkManager` daemon. Various options for the network manager can be configured via GUI or the command line, but not resolver-specific options. Instead, if the host obtains its network configuration via DHCP, changes to `resolv.conf` are governed by the network manager's communication with the `dhclient` daemon, using D-Bus IPC[2]. The behavior of `dhclient` is in turn configured via the file `/etc/dhcp3/dhclient.conf`.

Given the dependencies just described, where does the system administrator look when she determines there is a problem with name resolution? The first place she looks is `resolv.conf`. Luckily for her, there is a comment in the file that states it has been automatically generated by the network manager. However, this is where the trail goes lukewarm. The manual page for the network manager says nothing about the resolver. Perhaps the sysadmin recalls that name resolution failures can be symptomatic of DHCP misconfiguration, leading her to check the dhclient manpage and subsequently `dhclient.conf`. She may find some useful information there, but she is hard pressed to discover that the network manager is modifying the resolver's configuration by talking to the DHCP client. Also, `dhclient.conf` may have been configured by an automated script. The trail goes cold until Google is consulted and a solution is discovered. But this is unsustainable as a standard procedure for troubleshooting; eventually, even Google is out of answers.

Using a provenance graph (Figure 1) and the right query types (or tools we build specifically for this purpose), our fearless administrator would more quickly discover the dependencies in our example. Let us walk through the troubleshooting session once more with the help of provenance. The graph has been trimmed and condensed for clarity, so the steps taken in an actual session may be more involved. Also, the following analysis

[2]The D-Bus implements inter-process communication (IPC) via Unix sockets, with each endpoint represented as an inode object and two file objects in the kernel.

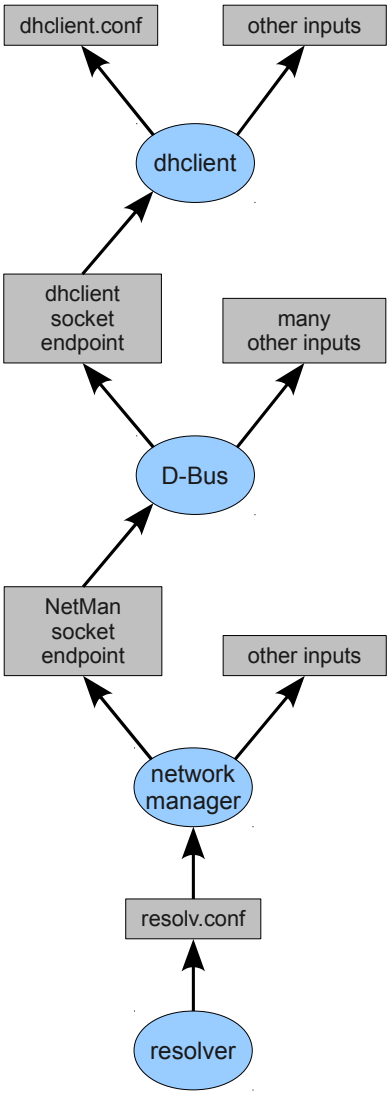

Figure 1: A partial provenance graph representing potential dependencies between components involved in Linux name resolution.

suggests that we are able to collect provenance for remote sockets. This is not currently the case for PASS, but we are working on such a mechanism.

We may safely start at the network manager node (hereafter referred to as *netman*), since we already know the source of the generated `resolv.conf`. The ancestors of netman include a socket endpoint (a "special" file) and various other inputs, one of which will be a configuration file. We can probably safely exclude the configuration file, because there is nothing in netman's documentation about resolver options. But why has netman received information from a socket via the D-Bus? It has obviously communicated with another process. Here is where we run into a slight snag: D-Bus often has a plethora of socket endpoints as inputs (in addition to other inputs), so how can we determine the right ancestor? In many cases we may not be able to directly identify the most important ancestor but we can probably narrow down our choices.

One possibility involves checking timestamps of the provenance edges between objects of interest. In this case we could compare the timestamp of outputs to netman's socket endpoint with the timestamps of D-Bus inputs from any of its ancestors. We would discard D-Bus inputs that occurred after outputs to the socket as well as inputs that occur too long before outputs. Other technical solutions are also possible, including the recording of socket descriptors in provenance objects.

Once we have reasonably narrowed our choices, we will have to rely on experience to take us the rest of the way. Knowing that our machine receives network configuration parameters via DHCP will allow us to discard many other D-Bus ancestors, such as the audio, printing, and display subsystems. Once we reach the dhclient ancestor, we can determine which of its configuration options found in `dhclient.conf` are likely to be involved in name resolution. Snippets of the provenance graph and discoveries made therein can be added to a troubleshooting knowledgebase that integrates with, e.g., a trouble ticket system.

4 Graph Compression

While the provenance of a program's outputs depends upon the program's inputs, the program itself is not necessarily dependent on every input to function correctly. For example, the program `cat`, which reads the contents of an input stream, only depends upon three shared libraries to function correctly, yet a provenance graph includes edges to *every* distinct input object that `cat` opens. Though the absence of these inputs may cause a script to fail, none of them is essential to the core behavior of `cat`. This is why we have described the provenance graph as a graph of potential dependencies only.

A similar fact holds for many programs; almost every file (or other input) that is necessary for them to function properly is loaded with their image or shortly thereafter. There are notable exceptions: programs such as Apache and PERL frequently load modules on-demand; daemons may reload their configuration files when a HUP signal is received, but will rarely reload a library; and shell scripts frequently defy all notions of predictability.

It would appear that the generated graph contains too much information for our purposes. Too many potential dependencies will make troubleshooting more difficult. Thus we need a way to increase the probability that an edge truly represents a *program* dependency. We can use several simple observations to guide us:

- If a program opens the same file one or more times on (nearly) every invocation, there is a high likelihood of dependency.
- The first-order dependencies of many programs are known a priori, either via direct experience, documentation, or technical detail, e.g. statically-linked programs.
- Files residing in well-known configuration directories such as `/etc` can be labeled with a high probability when all other indicators are (nearly) equal.
- Files residing in well-known log directories can be labeled with a low probability.
- Files that are created by and opened for reading and writing in short intervals (e.g., Emacs temporary files) or across multiple invocations by a single program might be safely excluded.

Acting intelligently on these observations will greatly reduce the size and density of the graph. Certain edges of which we are unsure may of course be left in the graph until we are better able to classify them. Once we have a graph of reasonable size and density, we can utilize PQL, elementary graph algorithms, statistical techniques, and even machine learning [6] to help answer troubleshooting queries.

5 Future Work

The current implementation of PASS examines provenance as expressed only via pipes, shared memory (`mmap`), process environments, and the filesystem. Unfortunately, more sources of provenance (and potential dependencies) are expressed via other information vectors, e.g. signals, sockets, and exit codes. As a result, provenance graphs generated by our implementation are not comprehensive. We believe that analysis of network I/O will prove to be a powerful technique. By tracking socket pairs, we can identify dependencies that span physical machines. For example, a network-aware approach would be able to identify dependencies between a web server and a DNS server. Expanding the collection and analysis phases in this way will require considerable effort.

We plan to develop configurable tools that will leverage our existing knowledge of system administration to construct efficient, domain-specific queries. We will also explore the utility of graph visualization in narrowing down the root causes of system problems. Finally, we plan to incorporate ideas from machine-learning, not only to help generate automatic analyses of provenance graphs, but to augment graphs with information gleaned from interactions of system administrators with our tools [2].

6 Conclusions

In our introduction, we made the claim that complete and accurate mental models are necessary to most tasks performed by system administrators, including troubleshooting and maintenance. As such, any tool that aids in the timely development of accurate mental models will be of huge benefit to sysadmins at both the junior and senior level. In this paper, we have explored the idea that analysis of provenance graphs can aid system administrators in troubleshooting problems that involve complex hidden dependencies. We are confident that if system administrators are amenable to automatic provenance collection, then this idea will emerge as an effective utility in everyday system administration.

References

[1] BROWN, A., KAR, G., AND KELLER, A. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management* (2001), pp. 377 –390.

[2] CUNNINGHAM, S. J., WITTEN, I. H., AND LITTIN, J. Applications of machine learning in information retrieval. *Annual Review of Information Science 34* (1999), 341–384.

[3] HOLLAND, D. A., BRAUN, U., MACLEAN, D., MUNISWAMY-REDDY, K., AND SELTZER, M. Choosing a data model and query language for provenance. In *Proceedings of the 2nd International Provenance and Annotation Workshop* (Salt Lake City, Utah, June 2008).

[4] HREBEC, D. G., AND STIBER, M. A survey of system administrator mental models and situation awareness. In *SIGCPR* (2001), M. A. Serva, Ed., ACM, pp. 166–172.

[5] KAR, G., KELLER, A., AND CALO, S. B. Managing application services over service provider networks: Architecture and dependency analysis. In *Proceedings of NOMS* (Hawaii, 2000).

[6] MARGO, D., AND SMOGOR, R. Using provenance to extract semantic file attributes. In *Proceedings of the 2nd Conference on Theory and Practice of Provenance* (Berkeley, CA, USA, 2010), TAPP'10, USENIX Association, pp. 7–7.

[7] MUNISWAMY-REDDY, K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *2009 USENIX Annual Technical Conference* (San Diego, California, 2009).

[8] ROUILLARD, J. P. Real-time log file analysis using the simple event correlator (SEC). In *LISA* (2004), USENIX, pp. 133–150.

[9] SUN, Y., AND COUCH, A. L. Global impact analysis of dynamic library dependencies. In *LISA* (2001), USENIX, pp. 145–150.

[10] TAKADA, T., AND KOIKE, H. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *LISA* (2002), USENIX, pp. 133–144.

[11] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI* (2004), pp. 245–258.

[12] 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* (2003), USENIX, pp. 159–172.