Provenance for System Troubleshooting

Marc Chiarini
Harvard SEAS
chiarini@seas.harvard.edu

Abstract

System administrators use a variety of techniques to track down and repair (or avoid) problems that occur in the systems under their purview. Analyzing 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 (metadata that records the history of files, pipes, processes and other system-level objects) can help expose these dependencies, giving system administrators a more complete picture of component interactions, thus easing the task of troubleshooting.

KEYWORDS: troubleshooting; diagnosis; dependencies; provenance; mental models.

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 [6, 15]. 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. Our goal is to build a provenance analysis engine that can automatically construct an accurate, queryable map of component interactions for single systems, networked sites, and beyond. Imagine arriving at your desk on a Monday morning and being able to explore what your site looks like based on provenance collected over the weekend.

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 filesystem). 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 flows from one component to another in order for the recipient 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* upon 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`.

These are obvious examples of dependencies, but note that the way in which we have defined dependency requires a clear understanding of what it means for a component to function correctly. Formally, *functional correctness* is determined by behavior: every input produces correct output, where the output also comprises error conditions. Thus, if a process outputs "file not found" for some input, it may still be functioning correctly. But this definition is too strict for our purposes.

System administrators have a general sense of how components are supposed to behave, and they can usually determine when something is awry. For example, misconfiguration of one or more components is a frequent cause of "abnormal" behavior. Formally, a DBMS that is configured with a parameter that directs it to the wrong dataset will produce the correct behavior *for how it is configured*, i.e., it will still answer queries as directed, etc. But the admin will see unexpected outputs because the inputs were different than expected. This leads us to an imprecise definition of "functioning correctly" as "exhibiting expected behavior".

3 The PASS Project

Digital *provenance* is metadata that describes the ancestry or history of a digital object. In non-digital domains, such as art curation, provenance is often collected manually. But in the digital domain, we have the capability to record provenance automatically. The *provenance-aware storage system* (PASS) project [26] 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 provenance graph would be virtually useless without a way of extracting pertinent information. We have developed a query language for graph-structured data called PQL [14, 13], which is capable of expressing complex queries with transitive closures. PQL operates on a semi-structured data model that allows us to ask questions about ancestors and descendants as well as about paths and subgraphs.

Consider the case in which we want to find all outputs of the `sendmail` daemon. The following SPARQL[2] query produces the desired result:

```
SELECT ?output WHERE {
   progfile "/usr/sbin/sendmail" ?process .
   ?output output-of ?process
};
```

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

[2]PQL is similar to SPARQL [32], an SQL-like query language for RDF.

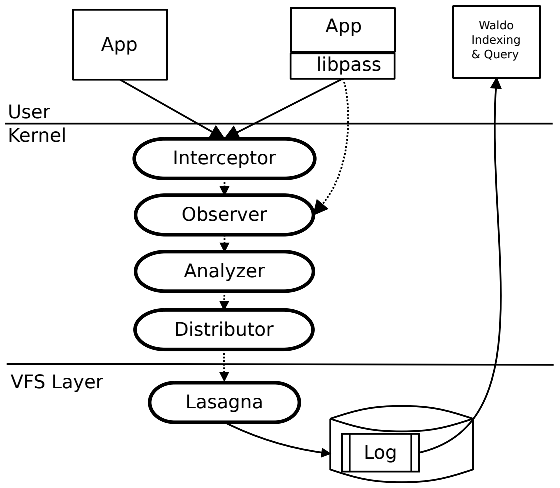

Figure 1: A diagram of the PASS Architecture.

In this example, *?output* and *?process* are variables. For every process that is an instantiation of `sendmail`, the query will return the process's output objects (e.g. files, pipes, processes, etc) in the variable *?output*. With PQL or a similar graph query language, we can issue simple queries such as the one in our example or complex queries such as "find all objects that result from the same (or similar) sequence of events", 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 ancestors to *function correctly*.

3.1 PASS Architecture

Figure 1 shows the PASS architecture.[3] The *interceptor* is a set of system call hooks that extract arguments and other necessary information from kernel data structures, passing them to the *observer*. Currently, PASS intercepts `execve`, `fork`, `exit`, `read`, `readv`, `write`, `writev`, `mmap`, `open`, `pipe`, and the kernel operation `drop_inode`. These calls are sufficient to capture the

[3]The modified image and description of architecture are used with permission from the authors [26].

rich ancestry relationships between Linux files, pipes, and processes. In addition, applications can be compiled to use `libpass`, which allows us to send application-specific provenance directly to PASS.

This raw “proto-provenance” goes to the *observer*, which translates proto-provenance into provenance records. For example, when a process P reads a file A, the observer generates a record that includes the fact that P potentially depends upon A (i.e., a cross-reference to A). The first time an object is created, the observer assigns to it a unique *pnode* identifier. A pnode number is similar to an inode number except that it is never recycled, even after an object is destroyed. This allows us to maintain provenance for every object of interest that ever existed. Suppose that files A and B and process P have all been assigned pnodes. When P exits, its pnode must be maintained so that the transitive potential dependency of B upon A can be queried. The same logic holds for the case in which A is deleted.

The *analyzer* then processes the stream of provenance records to eliminate *duplicates* and *cyclic dependencies*. Duplicates occur when a provenance object is used as input multiple times in the same “session” by another object. For example, after the initial read of pipe I by process P, every further read creates a duplicate record until the pipe is closed. Yet a record of the initial read is all we require to posit a potential dependency.[4] In similar fashion, the provenance of a file F to which P has written multiple times will only contain a single record of the initial write.

Unless time travel is possible[5], it is impossible for a descendant object to affect its ancestor. This is why cycles in the provenance graph must be broken or avoided by the analyzer. PASS avoids cycles by versioning. At the time of its creation, each provenance object is assigned a version number of 0. New versions of an object will be assigned monotonically increasing numbers. If process P reads from file F, and later writes to that same file, the analyzer will avoid a cycle by versioning the file’s provenance. If P reads the file again, the new record for this event will contain a cross-reference to Fv_1. That is to say that once F is written, further provenance will be collected only for subsequent new versions, and the provenance of Fv_0 will contain only whatever may have happened to F prior to the write and that didn’t involve a cycle. The versioning algorithm works well on cycles of any length, involving any type of object at any version level.

The PASS system is not limited to collecting provenance from local storage. We have implemented extensions that enable provenance collection from NFS shares and Amazon’s S3 service [3]. This capability is especially important to the multitude of organizations that have shifted their infrastructure into the cloud [27, 28].

Note that the interceptor is platform-specific by necessity, but that the observer and analyzer can be separated entirely from the operating system. The remaining components of the PASS architecture are not germane to the goals of this research. For a more complete description, we direct the reader to several prior works [25, 26].

4 Troubleshooting

4.1 Related Work

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 [36]. Others use event correlation in log-file analysis to identify extant and potential problems [1, 12, 17, 20, 34]. Wang et al. [37, 38] use comparisons of current system configurations against golden state configurations that have been generated via statistical analysis of machine populations. The HPC community has made significant strides in tracking down and diagnosing the root causes of failures in grids and clusters [2, 9, 31, 39]. Most of these approaches rely upon log analysis and can be extremely effective, especially in prescribed domains. However, log analysis may suffer from several drawbacks, including a lack of operational context (expected behavior); a “butterfly” effect on log messages that stem from small changes; corrupted messages; inconsistent log formats; and asymmetric log reports [29].

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 [18] and program images [35], but these have quite limited capabilities. For example, the former tool relies upon the correctness of package prerequisite information, and the latter tool only exposes compile-time dependencies.

Some tools [7, 33] are able to automatically construct operational dependency models by actively perturbing or probing live systems. Active perturbation involves performing multiple transactions or injecting “problems” outside of normal operation and tracking the affected components by observing likely execution paths. These methods are invasive, with the potential to cause unwanted load or unforeseen failures, and thus may be untenable in a production environment.

There are also many other approaches for exposing complex dependencies and causal relationships in dis-

[4] In general, this level of granularity imposes limitations on our ability to classify dependencies, i.e., we could keep the duplicates with a timestamp for more accurate resolution.

[5] There is now strong evidence to suggest that it is not [40]!

tributed systems[4, 8, 11, 30], but their ability to document, present, and query the models they build is limited. This makes them ill-suited for improving mental models and for generalized system and site-wide troubleshooting.

Two research projects reflect well the philosophy we wish to propagate. PDA is a tool for automated problem determination developed at IBM [16]. The tool starts with high-level health indicators that trigger custom-built probes when something is awry. The probes are built manually via analysis of trouble-ticket corpora. Their use-case scenarios reveal that a large number of problems fall into several categories to which standard troubleshooting procedures can be applied and perhaps even automated. We are optimistic that these categories will also manifest in our provenance graphs.

We recently discovered a tool that is similar–both in concept and implementation–to the framework we propose in this paper, but more narrow in scope and no longer actively developed. BackTracker [19] is designed to analyze system intrusions by tracing chains of events from a detection point (e.g., a suspicious process) back through a dependency graph to likely points of entry. The goal is to document the attack vectors that expose unknown vulnerabilities. Similar to our approach, the graph is constructed by intercepting and recording the information in system calls. The authors also provide several security-specific methods by which to prioritize and filter large portions of the dependency graph to help the user along. The requirements for system troubleshooting are more general, thus our work may be viewed as an attempt to address a superset of the issues tackled by BackTracker.

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 little 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.

4.2 A "Simple" Example

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[6]. 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 may look 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 2) 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 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.

[6]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.

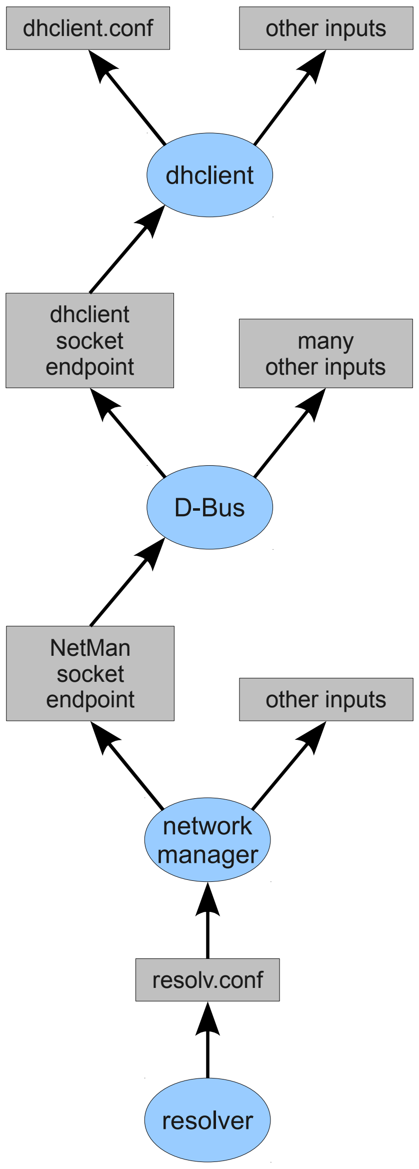

Figure 2: A partial provenance graph representing potential dependencies between components involved in Linux name resolution.

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.

The D-Bus example represents one of the worst-case scenarios in tracing root causes. The problem is twofold: at any given time, the number of ancestors and descendants of the daemon is usually very large, which results in an overwhelming path explosion; but the larger problem is that valuable provenance is hidden inside the D-Bus black box. For instance, if we had access to the internal dbus object name that identifies the connection between two clients, we could easily narrow our search to the real ancestors of the network manager. One way in which to accomplish this would be to create a provenance-aware version of D-Bus using the `libpass` library. This may be feasible for a small portion of particularly "opaque" system programs with many distinct inputs and outputs.

5 Ranking Dependencies

While the provenance of a process's outputs depends upon the process's inputs, the process itself is not necessarily dependent upon 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 "unimportant" edges will make troubleshooting more difficult. Thus we need a way to limit the scope of our queries to those ancestral objects that are most likely to have contributed to the behavior or contents of a target descendant.

5.1 Statistical Approaches

There is a statistical approach that will help us rank the contribution to dependency made by individual edges, full paths, and ancestral subgraphs.

Consider that any given snapshot of a provenance graph represents events *as they actually happened*. Suppose that we look at a snapshot of the provenance graph generated between time t_1 and time t_2. We see an edge from the process `/usr/sbin/chpasswd` to the file `/etc/pam.conf`. We also see several edges leading from other objects to `chpasswd`. Let us examine what we know. We do not track information flow, so we do not know what `chpasswd` did with information that it read from `pam.conf`. We do not know if the process or its descendants would have functioned correctly if `pam.conf` was missing or contained different content. The graph only tells us that the *provenance* of `chpasswd` and its descendants *depended upon* `pam.conf` in its current state.

Let us assume that the process functioned correctly during the snapshot period. How do we assign a dependency rank to edges in the graph? One way might be to take multiple snapshots at equally spaced intervals and count the number of snapshots in which the edge of interest appears. A high count would indicate a higher likelihood of dependence. While this may seem reasonable, it will not work.

Recall that an object is uniquely identified by a pnode number, which remains the same through successive versions (and even unto death). Once a node becomes a part of the graph, it is never removed. Any edges connected to the node remain in the graph as well. Thus, there is no difference between snapshots except for the creation of nodes and edges, and increases in object versions.

The correct approach takes advantage of the logical separation between provenance objects. A process is the running instantiation of a particular program. As such, two separate invocations of a program (processes) will be assigned distinct pnodes and appear as distinct nodes in the graph. Figure 3 shows an example of this scenario.

Process P_1 has read file A, written file B and then

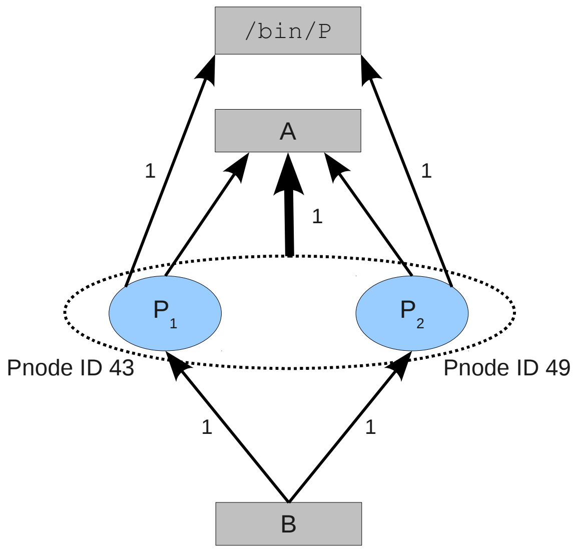

Figure 3: Processes with distinct pnodes. The program P (grouped processes) depends upon A with a ranking of 1 (thick edge).

terminated. Some time later, process P_2 takes the exact same actions. Notice that both processes have a provenance edge that points to the program executable `/bin/P`. For all processes that have a given executable as input, we can query whether or not they have a particular input (A in this case). If the same input object appears in the provenance of every process, then we declare that the current version of the program depends upon the input object with a ranking of 1.0. We denote this by grouping all such processes, drawing an edge from the group to the file, and labeling the edge with its rank. Alternatively, we can merge process nodes into a super-node to keep the graph clean.

There will be cases where only some instances of a program read from the same file. In these cases, only those instances are grouped and an edge is drawn to the file with a dependency rank given by

$$\frac{\texttt{\# of instances that read}}{\texttt{total \# of instances}}$$

We must not apply the same logic to rank the dependency of files upon programs. We do not know for certain what happens to the information that is read from a file by a process, e.g., whether it changes the behavior of the process. By contrast, a file always depends upon the process(es) that created and/or wrote to it. The reason is that a file is a passive object whose existence and content is governed by processes only. There is never any doubt that every bit of information in a file came from the process(es) that wrote to it.[7]

[7]Note that conceptually, if a process Q removes from a file all infor-

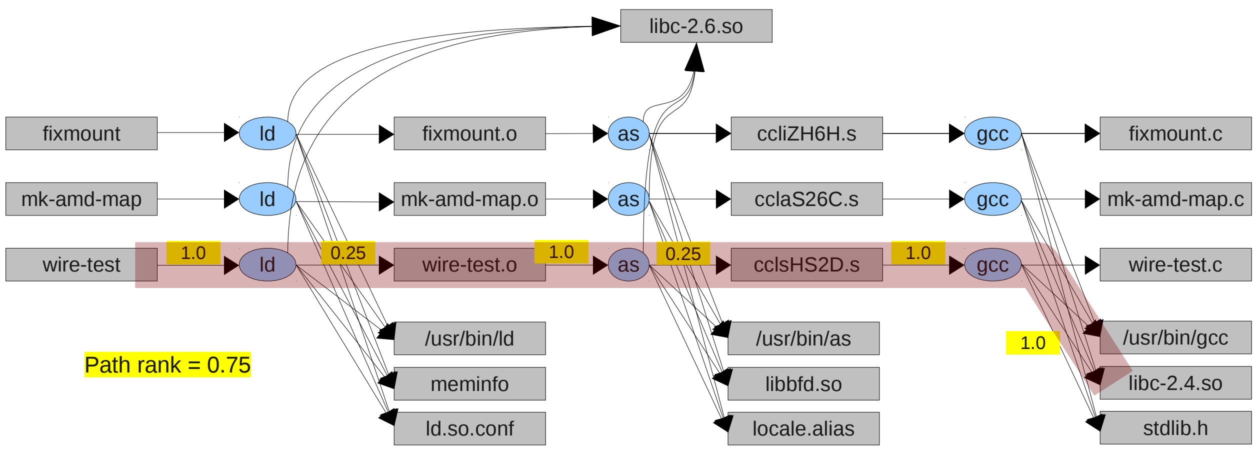

Figure 4: Dependency ranking for the path from `wire-test` to `libc-2.4.so`. The graph represents a mostly real provenance trace but the edge ranks are for demonstration only. If the executable really compiled, it would be difficult to identify the discrepancy between the two versions of libc.

Unnamed pipes are also passive objects that depend with certainty upon the process(es) that create them. Like processes, every new pipe is identified by a unique pnode number, and there always exists an edge to the process that created it. It might seem strange to claim that a pipe depends upon the processes that write to it, especially since we think of it as a simple channel by which processes communicate. The information sent to a pipe is meant to be consumed by one or more processes during the pipe's (relatively fleeting) lifetime. Unlike a file, none of the information in a pipe persists after it is torn down. Nonetheless, except for the timeframe, a pipe is serving one of the same purposes as a file; it is an information conduit between two processes.

Since named pipes are implemented as device special files, they remain usable after the process that created them exits. Thus we can use the same grouping technique to identify all processes that read from the same pipe, i.e., special file. With unnamed pipes, we need to go a step further. They are only connected between two single processes. It does not make much sense to claim that a program reads from the same pipe on every invocation. But we can claim that one program (i.e., every process from the same executable) always receives information from another program via a pipe, which implies a high-ranking transitive dependency upon the writer by the reader. A good way to represent this is to draw a directed edge from the group of processes to the group of pipes.

Armed with this metric, we can rank the potential dependence of paths or ancestral subgraphs. A path ranking is the average rank of all edges in the path. Similarly, a subgraph ranking is the average rank of all edges in the subgraph. For example, Figure 4 shows the rank of the highlighted path from the `wire-test` executable back to the ancestral file `libc-2.4.so`. We have omitted process grouping for clarity. When we query for the candidates that are likely causes of root problems, our tools should suggest exploration of the highest ranking paths first (accounting for rank adjustments from rules, filters, etc).

There are three caveats regarding this approach. First, it is has yet to be empirically tested. But our knowledge of operating systems provides a solid foundation. Second, the approach requires a bootstrap period during which rankings may be heavily influenced by existing abnormal behavior. This has the potential to mislead system administrators during analyses. We must therefore provide the ability for admins to manually adjust rankings in the graph, either permanently or via a "what-if" mode in a query session. The last warning is that while we expect the accuracy of rankings to improve over time, a large number of abnormal events may throw certain subgraph rankings into chaos at any time. We might be able to mitigate this by having the provenance subsystem alert us to statistical changes that exceed a certain threshold.

5.2 Heuristics

Statistical methods (and others) will carry us a fair distance in compressing the query space. But there is no reason to exclude existing knowledge about dependencies or rules of thumb. We now present several observations that will help us improve our rankings and refine our queries even further:

mation written by another process *P*, we might want to say that the file no longer depends upon *P*. But we have no way of representing this at our current level of granularity.

- Our current rules assign a dependency rank to edges based upon how many instances of a program read from the same input object. Informally, this says that for an edge with a higher rank than another, there is a greater chance that the input object affects the program's behavior.

  We might make the assertion that all but the simplest of daemons will always attempt to open and read from their associated configuration files. But note that if a daemon accepts a parameter that prevents loading of config files or specifies a different config file than the default (as many do), its input edges may receive a much lower rank than expected.

  Whether the daemon is dependent upon a specific config file is usually *conditioned* upon its starting parameters. Fortunately, PASS includes provenance about the environment in which a program is started. If we observe that a daemon always opens the same configuration file in the presence of some starting parameter, then we will be able to rank dependencies for different instances of the daemon (e.g., when '-c' is provided, the daemon *always* loads config file *C*, but in the absence of '-c', the daemon always loads config file *A*.). That is to say that we will group processes as usual but the group edge rank will indicate how many instances of the program read from an input object *when started with a given parameter*.

- The first-order dependencies of many programs are known a priori, either via direct experience, documentation, or technical detail, e.g. statically-linked programs. We can assign a rank of 1 to the outbound edges of these programs automatically upon first invocation.

- Popular objects, as measured by descendant subgraph size, are less likely to be the singular cause of a problem. It is a reasonable assumption that if a popular object is the cause of a problem, descendants along more than one path would exhibit unexpected behavior. In the case that we are only seeing one or a few objects with unexpected behavior, we can have our query engine dynamically reduce the dependency ranking for paths or subgraphs that include popular ancestors. The triggers for such a reduction and the amount of rank reduction will need to be determined by experiment.

  Example: almost every program has `libc` as a core library. This means that almost every edge that points to the `libc` node will have a dependency rank of 1. But this node is uninteresting exactly because so many programs depend upon it. If `libc` is broken or missing, we are likely to know immediately.

- Edges to files residing in well-known configuration directories or files with well-known names can be labeled with a high rank when all other indicators are equal or nearly so.

  For example, if a program *P* opens a file called `logrotate.conf` in directory `/etc`, then we have two more pieces of evidence to support the assertion that *P* depends upon `logrotate.conf`. The weight of this evidence will need to be adjusted according to several factors, which is left for future work.

  Of course, we must also provide a means by which we can fix the dependency or non-dependency of an object upon another object. This allows us to correct edges in the graph for which our algorithm has failed. There may be a semi-automated way in which to do this, which is also left for future work.

- Edges to files residing in well-known log directories can be labeled with a low rank.

  For example, `/var/log/messages` is a file that is frequently written, but certain log viewing/analysis/filtering/aggregation tools, such as Splunk, will frequently read the file as well.[8] In many cases, the absence or corruption of a log will not affect the proper functioning of the reading process. But it is difficult to know how far such a failure might propagate.

- Edges to files residing in well-known temporary directories can be labeled with a low rank.

  By definition, programs should not rely upon any data stored in a temporary directory (e.g. `/tmp`). However, programs do sometimes use such directories to create temporary pipes or to communicate information to themselves in the near future. These kinds of dependencies will need to be reviewed.

- Edges to files that are created by and opened for reading and writing in short intervals and across multiple invocations by a single program may be safely labeled with a low rank.

  For example, applications such as Emacs create backup files during editing. While the user may rely upon such backups, Emacs does not require these files to function correctly.

- Files that are created/written by an editor like `vi` are not dependent upon `vi`. They are dependent upon

[8] Many of these tools avoid the local filesystem altogether by logging to a centralized host via the network. In this case, provenance would be captured using network service extensions to PASS.

the human being using `vi`. This is a dependency that we can capture because we record the (E)UID of every process. If the file is created or written via shell redirection, we can still capture the dependency based upon the shell owner.

- Files created or modified by a script *are* dependent upon the script and probably many of its ancestors. But the path must ultimately lead back to the process that generated the script, whether manual or automatic.
- Any troubleshooting tools we build can integrate the use of whitelist, blacklist, Bayesian, and other filters. These will give the user flexibility in their queries and will certainly encourage use of the tool for purposes other than troubleshooting.

Acting intelligently upon the given observations will reduce the size and density of the query space. Note that none of our algorithms or heuristics is modifying the graph. Edge rankings will be applied only at query time based upon specified rules and filters, and will be computed in a lazy fashion. We do not want to rank one million edges for a single query unless it is necessary. For example, if a filter limits the query space to files in a particular directory, we do not need to rank edges from or to files in other directories, nor unnamed pipes.

As an example of where filtering may fail, suppose we determine that a program is behaving abnormally. It has file *A* as input, amongst others. A conventional rule of thumb may lead us to filter based upon time; the program was working until a certain point in time, so it is reasonable to ask which process most recently wrote to *A* around that time. But this may not help us because at the granularity of our provenance, the information that was most recently written to *A* might not be the information that is causing a malfunction. It is possible that some previous write is causing a malfunction. Perhaps the program did not run during the period between the previous write and the most recent write. Thus the effect of the previous write to *A* did not manifest until the program was run again.

6 Under-specified Queries

Filters and rules will help, but they are not sufficient. Even if we assume that the graph contains only actual dependencies, we still need the ability to limit the scope of *under-specified queries*. Such a query has the potential to return a very large subgraph because it does not sufficiently constrain ancestral breadth and depth. For example, if we query on the full lineage of `/var/log/dmesg`, we are likely to see all ancestors going back to installation of the operating system. Depending upon the context, this may be unhelpful. The ability to specify queries precisely assumes the existence of an excellent mental model by which to navigate the provenance graph. As the graph expands, "surgical" queries demand a familiarity that is unsustainable without aid. Thus, our tool needs to be able to guess at good places to stop in the lineage of a target object.

Several researchers in our group are attempting to tackle this problem based upon ideas inspired by web search [23]. *Provrank* is an algorithm that judges the importance of objects based upon their *frequency* across all possible lineage queries. Objects with a high frequency appear in too many lineage queries. Thus, if some process appears in the query path of every descendant object of interest, it does not add any important information to a query result and represents a good cutoff point. Another metric – *frequency dissimilarity* – captures the relative frequency of an object. That is to say, it measures how often an object appears in query results that contain objects of the same kind (based upon some criteria). Thus, the `bash` shell will have a lower frequency dissimilarity in queries that ask for the lineage of `mkdir`, than in queries that ask for the lineage of a random user document (i.e., the `bash` node would be a good cutoff point for queries about user documents).

Further work is required in this area to help sysadmins semi-automatically constrain their queries.

7 Building Tools

With a few decent algorithms under our belt, a gaggle of heuristics, and a good working knowledge of operating systems, what capabilities do we want for our tools and their interfaces?

In our resolver example, we guided the reader through a troubleshooting session that uses the provenance graph. Although based on a real use case, the example was directed and abbreviated for clarity. In a real session, a sysadmin would need a guide as well; something to improve their chances of diagnosing the problem.

Ideally, our tool must be able to present a relatively small group of root-cause candidates. But we are also helping admins build mental models. We expect to introduce several interfaces that leverage web technologies as well as the familiar command-line interface for conventional programmatic control.

Since PQL is the primary method of querying the provenance graph, we also plan to introduce a set of predefined query classes that will help users learn how to construct and refine more complex queries. A graphical tool is in the works that will enable the construction of queries via example as well.

Finally, integration is paramount. The user must be able to build the toolchain with relative ease and connect it to existing monitoring and troubleshooting frameworks. For example, as problems are solved, relevant snippets of the graph and associated queries can be entered into a trouble ticket system and reviewed in subsequent incidents that exhibit similar symptoms.

7.1 Visualization

Provenance graphs can grow to enormous proportions, which tends to work against building robust mental models. Visualization can dramatically improve the ability for users to absorb and understand complex structures. As such, it is one of the most important aids to provenance analysis.

We have already built a tool called Orbiter [22] that can, among other capabilities, display provenance graphs with adjustable magnification, perform rudimentary filtering (e.g., degree, object type, timestamp, etc) and querying of ancestors and descendants, and summarize subgraphs at customized levels of granularity. We plan to extend Orbiter's capabilities with query subgraph highlighting, regular expression filters, process grouping, annotations, and programmable views. We will encourage system administrators to describe the most useful aspects of the tool, as well as their thoughts on whether and how to eliminate or improve its failings.

8 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, message queues, shared memory, semaphores, 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.

Another drawback in our current implementation is the inability to collect provenance from root volumes or to aggregate provenance from multiple disparate volumes. We are working to address these shortcomings by building a new collection platform [21] based in the Xen hypervisor [5] that obtains provenance directly from system calls inside of guest VMs. We expect this reorientation to yield new benefits, which include supporting a better case for adoption than a patched Linux kernel.

There are many other technologies that might be employed to help build and answer domain-specific troubleshooting queries, including further analysis of graph structure, more advanced statistical techniques, and a community-based query database. We also plan to incorporate ideas from machine-learning, not only to help conduct semi-automatic analyses of provenance graphs and provide better dependency rankings, but to augment graphs with information gleaned from interactions of system administrators with our tools [10, 24].

9 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 great 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.

10 Acknowledgments

The author would like to thank several members of the PASS group at Harvard University SEAS for their comments and insights on this research: Uri Braun, Peter Macko, Daniel Margo, and Margo Seltzer. He would also like to thank his LISA shepherd, Nicole Forsgren Velasquez, and the LISA program committee.

11 Availability

A working prototype is not yet available. However, readers are encouraged to periodically check the website below for news and updates.

`http://www.eecs.harvard.edu/syrah/pass/`

References

[1] Splunk. Web, June 2011. `http://www.splunk.com/`.

[2] AHARON, M., BARASH, G., COHEN, I., AND MORDECHAI, E. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I* (2009), Springer-Verlag, pp. 227–243.

[3] Amazon Simple Storage Service (Amazon S3). Web, June 2011. `http://aws.amazon.com/s3`.

[4] BAHL, P., CHANDRA, R., GREENBERG, A., KANDULA, S., MALTZ, D. A., AND ZHANG, M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *ACM SIGCOMM Computer Communication Review* (Aug. 2007), vol. 37, ACM, pp. 13–24.

[5] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (2003), 164–177.

[6] 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 (CSCW)* (November 2004), J. D. Herbsleb and G. M. Olson, Eds., ACM, pp. 388–395.

[7] 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.

[8] CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the IEEE/IFIP 32nd International Conference on Dependable Systems and Networks (DSN)* (2002), IEEE Computer Society, pp. 595–604.

[9] CHUAH, E., KUO, S.-H., HIEW, P., TJHI, W. C., LEE, G., HAMMOND, J., MICHALEWICZ, M. T., HUNG, T., AND BROWNE, J. C. Diagnosing the root-causes of failures from cluster log files. In *Proceedings of the 2010 International Conference on High Performance Computing (HiPC)* (Singapore, Dec. 2010), pp. 1 –10.

[10] 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.

[11] ENSEL, C. New approach for automated generation of service dependency models. In *Proceedings of the Second Latin American Network Operations and Management Symposium (LANOMS)* (Belo Horizonte, Brazil, Jan. 2001).

[12] HANSEN, S. E., AND ATKINS, E. T. Automated system monitoring and notification with swatch. In *Proceedings of the 7th USENIX Conference on System Administration* (1993), USENIX Association, pp. 145–152.

[13] HOLLAND, D. PQL language guide and reference. Web, June 2011. `http://www.eecs.harvard.edu/syrah/pql/docs/guide.pdf`.

[14] 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 (IPAW)* (June 2008).

[15] HREBEC, D. G., AND STIBER, M. A survey of system administrator mental models and situation awareness. In *Proceedings of the 2001 ACM SIGCPR Conference on Computer Personnel Research* (2001), ACM, pp. 166–172.

[16] HUANG, H., JENNINGS, III, R., RUAN, Y., SAHOO, R., SAHU, S., AND SHAIKH, A. PDA: a tool for automated problem determination. In *Proceedings of the 21st Conference on Large Installation System Administration (LISA)* (2007), USENIX Association, pp. 153–166.

[17] HUANG, L., KE, X., WONG, K., AND MANKOVSKII, S. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research (CASCON)* (2010), ACM, pp. 313–326.

[18] KAR, G., KELLER, A., AND CALO, S. B. Managing application services over service provider networks: architecture and dependency analysis. In *Proceedings of the IEEE/IFIP 7th Network Operations and Management Symposium (NOMS)* (2000), J. W.-K. Hong and R. Weihmayer, Eds., IEEE, pp. 61–74.

[19] KING, S. T., AND CHEN, P. M. Backtracking intrusions. *ACM Transactions on Computer Systems 23*, 1 (Feb. 2005), 51–76.

[20] KRIZAK, P. Log analysis and event correlation using variable temporal event correlator (VTEC). In *Proceedings of the 24th International Conference on Large Installation System Administration (LISA)* (2010), USENIX Association, pp. 1–11.

[21] MACKO, P., CHIARINI, M., AND SELTZER, M. Collecting provenance in the xen hypervisor. In *Proceedings of the 3rd Workshop on the Theory and Application of Provenance (TaPP)* (June 2011), USENIX Association.

[22] MACKO, P., AND SELTZER, M. Provenance map orbiter: Interactive exploration of large provenance graphs. In *Proceedings of the 3rd Workshop on the Theory and Practice of Provenance (TaPP)* (June 2011), USENIX Association.

[23] MARGO, D., MACKO, P., AND SELTZER, M. Constraining provenance queries. NEDB Poster Session, January 2011. Presented at poster session of New England Database Summit 2011.

[24] MARGO, D., AND SMOGOR, R. Using provenance to extract semantic file attributes. In *Proceedings of the 2nd Workshop on the Theory and Practice of Provenance (TaPP)* (2010), USENIX Association.

[25] MUNISWAMY-REDDY, K. *Foundations for Provenance-Aware Systems*. Dissertation, Harvard, Mar. 2010.

[26] MUNISWAMY-REDDY, K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference ATC* (2009).

[27] MUNISWAMY-REDDY, K., MACKO, P., AND SELTZER, M. Making a cloud Provenance-Aware. In *Proceedings of the 1st Workshop on the Theory and Practice of Provenance (TaPP)* (2009).

[28] MUNISWAMY-REDDY, K., MACKO, P., AND SELTZER, M. Provenance for the cloud. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies FAST* (2010), USENIX Association, pp. 197–210.

[29] OLINER, A., AND STEARLEY, J. What supercomputers say: A study of five system logs. In *Proceedings of the IEEE/IFIP 37th Annual International Conference on Dependable Systems and Networks (DSN)* (2007), IEEE Computer Society, pp. 575–584.

[30] OLINER, A. J., AND AIKEN, A. Online detection of Multi-Component interactions in production systems. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (June 2011).

[31] OLINER, A. J., AIKEN, A., AND STEARLEY, J. Alert detection in system logs. In *Proceedings of the IEEE International Conference on Data Mining* (2008), IEEE Computer Society, pp. 959–964.

[32] PRUD'HOMMEAUX, E., AND SEABORNE, A. SPARQL Query Language for RDF. W3C Recommendation, 2008.

[33] RISH, I., BRODIE, M., ODINTSOVA, N., MA, S., AND GRABARNIK, G. Real-time problem determination in distributed systems using active probing. In *Proceedings of the IEEE/IFIP 11th Network Operations and Management Symposium (NOMS)* (2004), pp. 133–146.

[34] ROUILLARD, J. P. Real-time log file analysis using the simple event correlator (SEC). In *Proceedings of the 18th Conference on Large Installation System Administration (LISA)* (2004), USENIX, pp. 133–150.

[35] SUN, Y., AND COUCH, A. L. Global impact analysis of dynamic library dependencies. In *Proceedings of the 15th Conference on Large Installation System Administration (LISA)* (2001), USENIX, pp. 145–150.

[36] TAKADA, T., AND KOIKE, H. Mielog: A highly interactive visual log browser using information visualization and statistical analysis. In *Proceedings of the 16th Conference on Large Installation System Administration (LISA)* (2002), USENIX, pp. 133–144.

[37] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of the 6th Conference on Operating Systems Design & Implementation OSDI* (2004), pp. 245–258.

[38] WANG, Y.-M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. Strider: A blackbox, state-based approach to change and configuration management and support. In *Proceedings of the 17th Conference on Large Installation System Administration (LISA)* (2003), USENIX, pp. 159–172.

[39] XU, W., HUANG, L., FOX, A., PATTERSON, D., AND JORDAN, M. I. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating Systems Principles (SOSP)* (2009), ACM, pp. 117–132.

[40] ZHANG, S., CHEN, J. F., LIU, C., LOY, M. M. T., WONG, G. K. L., AND DU, S. Optical precursor of a single photon. *Physical Review Letters 106*, 24 (June 2011).