USING CAUSAL PATHS TO IMPROVE PERFORMANCE AND CORRECTNESS IN DISTRIBUTED SYSTEMS

by

Patrick Alexander Reynolds

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Amin Vahdat, Supervisor

_____
Jeffrey Chase

_____
Carla Ellis

_____
Alvin Lebeck

_____
Janet Wiener

_____
Jun Yang

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2006

ABSTRACT

(Computer Science)

USING CAUSAL PATHS TO IMPROVE PERFORMANCE AND CORRECTNESS IN DISTRIBUTED SYSTEMS

by

Patrick Alexander Reynolds

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Amin Vahdat, Supervisor

_____
Jeffrey Chase

_____
Carla Ellis

_____
Alvin Lebeck

_____
Janet Wiener

_____
Jun Yang

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2006

Abstract

With the rise of the internet has come a rise in the importance of distributed systems. Distributed systems—groups of networked computers cooperating to run a single application or service—power modern web sites and communication services, infrastructure services like content-distribution networks and overlay networks, and parallel computations. With the complexity of distributed systems comes bugs: errors that affect a system's correctness, performance, or both. Traditional debugging tools and methods do not apply well to distributed systems. Thus, new tools are needed that will enable programmers to find and fix bugs in distributed systems. These improvements will in turn improve the performance and reliability that end users experience.

Many bugs reflect discrepancies between a system's behavior and the programmer's assumptions about that behavior. In this thesis, we show that expressing distributed system behavior as a set of causal paths and helping programmers separate those paths into expected and unexpected behavior is a powerful technique for improving the performance and correctness of distributed systems. This dissertation presents three distinct debugging tools, representing several approaches to tracing, analyzing, checking, and displaying system behavior.

We applied our debugging tools to several distributed systems of varying size and complexity, accurately discovering the behavior of each system. We found and fixed bugs in several of the systems. Our experiences using the tools with real systems demonstrate a trade-off between the disruptiveness of a debugging methodology and the accuracy of its results. Namely, Project 5 can be applied without changing the target application, but its accuracy is limited. Wide-area Project 5 requires interposition and results in more accurate and more detailed analyses. Pip requires

changes to application source code and generates a behavior model detailed enough for automatic extraction of errors. It is difficult to quantify the benefits of debugging tools. However, a number of different programmers were able to use these tools to rapidly locate and diagnose subtle and previously unknown bugs, validating the utility of these techniques for improving the reliability and performance of distributed systems.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank the following people who have helped to make my graduate career a success. It has been a privilege and a pleasure to work with my advisor, Amin Vahdat. Amin has provided years worth of ideas, support, encouragement, feedback, and advice. He enthusiastically pushes my half-formed ideas in useful directions. He helped me hone my research, writing, and presentation skills. Finally, he has been particularly accommodating toward my desire to split my time between California and North Carolina.

Janet Wiener, my mentor at HP Labs, has been invaluable. She is an ideal colleague, co-advisor, editor, and friend. She has contributed to this dissertation at every level, from big ideas to small refinements of design, implementation, or presentation. My phone calls and visits with Janet have been a great opportunity to refine my thinking on existing ideas and to brainstorm new ones.

Jeff Mogul, Mehul Shah, Marcos Aguilera, Chip Killian, and Athicha Muthitacharoen, in addition to Amin and Janet, have been inspiring as collaborators and invaluable as co-authors. Additionally, Chip, plus Dejan Kostić, David Oppenheimer, Ryan Braud, Alistair Veitch, and Mustafa Uysal, helped me apply Pip to the systems they designed or maintained. My research has benefited greatly by my collaborators' assistance, ideas, and feedback.

HP Labs, particularly my managers there, John Sontag and Rich Friedrich, have provided resources, encouragement, and access to excellent colleagues. It has been a privilege to work as an HP intern for much of the last three years.

Jeff Chase, Carla Ellis, Alvin Lebeck, and Jun Yang have been kind enough to serve on my committee. Their feedback and support have helped me solidify this dissertation.

Jeannie Albrecht and Margo McAuliffe hosted all of my trips to California. Their generosity and friendship have made my travels easier and happier.

Diane Riggs tirelessly supports every graduate student in the Duke CS department. Diane takes care of the administrative business of graduate life so that we do not have to. I appreciate her loyal service, her professionalism, and her advocacy throughout my Duke career.

My parents have supported my education for more than two decades. They made sure I always had opportunities to learn, particularly about computer science. They put me through college, and they are still ready with advice to this day. As scholars, they set an example of excellence that I am still striving to reach.

Finally, my wife Kristina Killgrove makes everything worthwhile and anything possible. The work ethic she brings to her own graduate studies is an inspiration to me. I rely on her encouragement, her patience, and her love. None of this would have been possible without her.

Chapter 1

Introduction

As the internet grows, so does the academic and commercial importance of distributed systems. Distributed systems are collections of networked computers cooperating to run a single application or service, including web sites, email services, storage services, and peer-to-peer networks. They now power almost every site and service on the internet. Early internet services ran on single hosts. However, the performance, reliability, and availability requirements of modern services dictate that they run on distributed systems.

Distributed systems power much of the modern internet. First, clusters of servers [21] allow load-balanced systems to run web or email [57] services. Second, wide-area replication enables content-distribution networks [2, 22, 65], archival storage systems [41, 46], and localized versions of international websites. Content-distribution networks and localized websites, in particular, place copies of common content in many locations so that any given user request can be served from a nearby site. Finally, peer-to-peer networks connect end nodes directly to enable end-system multicast [31], file distribution [13, 38], and file sharing [25, 35, 17]. These protocols all serve to disseminate data from client to client rather than from one central server to all clients. Direct client communication relieves performance bottlenecks at the server and can reduce network load.

Four requirements justify the need for distributed systems: performance, reliability, availability, and flexibility. Often, the load delivered to a service is too high for a single computer to handle. Many internet services are highly parallelizable; thus, adding more servers increases the throughput a service can achieve. While an

individual user request might have to be processed serially on one node, different requests are largely independent and can be handled in parallel on different nodes. Additionally, servers may be distributed among many locations close to end users to take advantage of higher-bandwidth, lower-latency connections.

The second requirement is reliability. Users expect stored data such as email, uploaded files, and commercial transactions to survive individual server failures. More precisely, users prefer not to know about failures at all. Because server failures are inevitable, internet services must be built with some redundancy. User data must be stored in several locations so that individual disk or server crashes do not permanently lose information.

Third, internet services must remain available even when individual servers or network links fail. Availability is distinct from reliability in that it measures the instantaneous usability of a service, while reliability measures the probability that a service will lose data permanently. To maintain availability, an internet service must include redundant servers and network connections to maintain service even when individual servers or connections fail.

Redundancy can improve both availability and reliability. However, poorly designed redundancy can lead to data corruption or reduced performance, reliability, and availability. Programmers must design their systems carefully to provide both the required consistency and the desired performance and availability.

The final reason to use distributed systems is flexibility. New protocols like peer-to-peer systems, online games, and chat rooms are inherently distributed. There is no way to connect many users without involving many computers.

Distributed systems, while useful, are hard to build and often contain bugs. Distributed systems are inherently parallel, and programming parallel tasks is difficult. They consist of many hosts performing different tasks, which means more code to

write and more state to keep track of. Distributed systems are asynchronous, meaning that communication among hosts can be delivered late, out of order, or not at all. Nodes and network links can fail and recover unpredictably. Designing software to provide performance, reliability, availability, and flexibility on a distributed system is far more challenging than writing serial programs for a single host.

Because of the difficulty in building distributed systems, they can suffer from hard-to-find performance and correctness bugs. An error in the distributed system providing an internet service affects end users' experiences—their perceptions of reliability and performance—when using that service. Distributed systems exhibit more, and more complex, bugs than single-node systems for two reasons. First, parallelism is challenging. A distributed system may have related processing tasks occurring on many hosts at once, bound only loosely by asynchronous communication. Second, distributed systems entail new sources of failure. More components implies more faults. Communication over a network makes a system vulnerable to network delays and failures. Finally, nodes running different software and communicating on a public network implies more opportunities for security breaches, which can affect distributed systems in entirely unpredictable ways.

Bugs are not only more common in a distributed system than in traditional, single-node systems. They are also harder to find. A distributed system has more nodes, events, and messages to monitor. Useful performance metrics often involve simultaneous measurements on several nodes. A traditional debugger with breakpoints and access to local and global variables does not scale well to capturing and monitoring the state on many nodes at once. Distributed applications might cross administrative domains or might involve heterogeneous hardware and software, meaning that no single debugging tool will be allowed or compatible at all nodes. Finally, a bug might span multiple nodes. That is, symptoms could appear on one node, while the

root cause is on another.

Programmers debug distributed systems today using a combination of single-node debuggers and logging via *print* statements. The print statements provide a tunable amount of information about the state at each node, which the programmer can examine manually or with scripts to determine where to focus further attention with a single-node debugger. This ad hoc methodology fails to gather higher-level relationships among nodes, behavior changes over time, or performance measurements. It also requires extensive and often disorganized modifications to application source code. More structured techniques can automate much of the process and improve the usefulness of the information extracted.

The new debugging methodology we describe here is built on two ideas. First, performance and correctness bugs are deviations from expected behavior. By reconstructing an application's actual behavior and helping the programmer or maintainer compare it to the intended behavior, debugging tools can help to uncover bugs. Deviations from expected behavior can indicate several different conditions: errors in source code, bad user input, or host or network problems, each of which can be fixed by changes in source code. Deviations from expected behavior can also indicate incorrect or incomplete expectations. Even so, identifying unexpected behavior helps programmers identify bugs.

The second idea underlying our debugging methodology is the use of *causal paths* as a unit of application behavior. A causal path is a partially ordered set of related events on one or more nodes in a distributed system. Each event other than the first is caused by exactly one event that precedes it in time. The path may split; that is, one event may cause more than one other event. Figure 1.1 shows a sample three-tier system with one causal path indicated: the client sends a request to the web server, which sends a message to an authentication server, which reads from a database

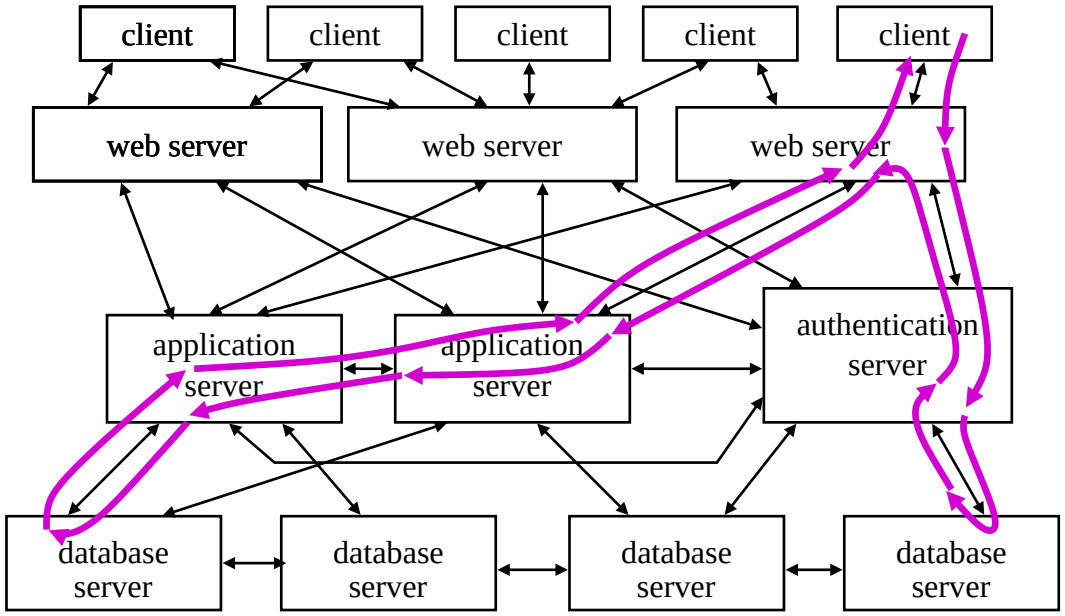


Figure 1.1: Three-tier system with one causal path indicated.

server. After receiving a reply, the web server communicates with an application server, which contacts another application server, which contacts a database. Each server replies in turn, and a response is sent to the client.

Causal paths often begin with a user request, as is the case in Figure 1.1. Less commonly, causal paths can begin with timers or error conditions. We refer to the specific set of events beginning with an individual request or impetus as a causal path *instance*. A description of causality describing many similar instances is a causal path *pattern*. A distributed system will normally have many causal path instances, conforming to one or more patterns, occurring at any given time. Each recorded event belongs to exactly one path instance.

Our hypothesis in this dissertation is that expressing distributed system behavior as a set of causal paths and helping programmers separate those paths into expected and unexpected behavior is a powerful technique for improving the performance and correctness of distributed systems. To support this thesis, we introduce three debugging tools and an expectation language, and we describe how we applied them to

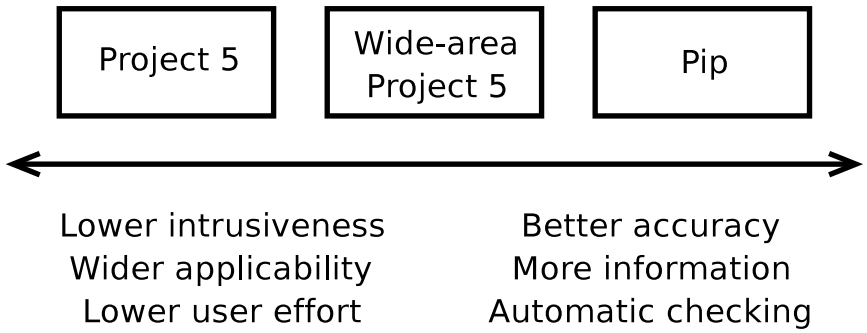


Figure 1.2: Three debugging tools illustrating the tradeoff between intrusiveness and accuracy.

understand and debug real systems.

1.1 Overview

In the remainder of this chapter, we introduce three causal path debugging tools: Project 5, Wide-Area Project 5 (WAP5), and Pip. All three tools extract causal paths from traces of distributed systems. Each tool lets the programmer explore the extracted paths, and Pip automatically checks them against programmer-specified expectations.

All three tools can also serve a purpose beyond debugging: revealing application structure and performance to programmers unfamiliar with the system. Programmers inheriting an existing code base or maintaining a deployed system have to perform *discovery*, figuring out dependencies, communication patterns, and bottlenecks. Our three debugging tools can all aid in this process.

Collectively, our debugging tools illustrate the trade-off between intrusiveness and accuracy, as shown in Figure 1.2. Project 5 requires the least user effort to apply and works with the widest variety of systems. Pip requires more effort and delivers detail and accuracy sufficient for automatic checking. WAP5 compromises on both counts: it is easier than Pip to apply, and it produces more detailed results than Project 5.

1.1.1 Target audiences

Our debugging tools have three target audiences:

- **Primary programmers:** the original authors of a system, who wish to debug or optimize their own code. They have access to application source code and are familiar enough with it to know what constitutes expected behavior.
- **Secondary programmers:** other maintainers or contributors to an application, who have access to the source code but might be unfamiliar with it. For example, secondary programmers might be those who have inherited or joined an existing project. Their first goal is to learn about an application, and they might not know what system behavior is expected.
- **Operators:** programmers or system administrators who must keep a production system correct through upgrades and other configuration changes. They often do not have access to source code, but can still use black-box techniques or existing source code annotations to monitor an application's performance and correctness.

1.1.2 Project 5

In many cases, distributed systems are best thought of as consisting of black boxes. The components involved can be difficult to examine or modify, because these components might be commercial software, hardware, or otherwise closed to system builders. For example, a web hosting site might consist of load balancers, web servers, application servers, databases, and authentication servers, any of which could be commercial software or even hardware. Even components that are open to examination or modification can be complex enough that system builders find it easier to apply black-box techniques first, to focus their efforts before undertaking more intrusive debugging

efforts. Someone designing or testing such a system might have to treat each component as a black box and examine only the messages that the components exchange.

Our first debugging tool, Project 5 [1], enables analysis and debugging of distributed systems made up of black boxes. Project 5 treats each distributed application as a collection of black-box components. That is, Project 5 uses no information about the structure of components and does not depend on modifying or perturbing component behavior. Further, it does not require any knowledge about the semantics of inter-component messages.

Project 5 infers the causal paths in a local-area distributed system from traces of messages between components. It does not require any knowledge of components' internal behavior and does not need to instrument them or modify their operations. The causal paths that Project 5 finds include processing times at each component, allowing developers to see which paths and which components account for the majority of processing time.

Project 5 infers the causal paths that requests take through local-area distributed systems. It uses two distinct algorithms, *nesting* and *convolution*, to extract causal path patterns from traces of network messages. For each path pattern inferred, the algorithms quantify the time spent at each component. Finally, Project 5 presents the paths to the developer or system builder to help inform decisions about where to add more capacity or focus lower-level debugging efforts.

Using several real and simulated systems, we explored the situations in which Project 5's inference algorithms can extract paths accurately. In most cases, the nesting algorithm's accuracy is high enough to extract the correct structure and timing for the target system.

1.1.3 WAP5

Debugging wide-area distributed systems presents additional challenges beyond those present when debugging data centers and other local-area systems. Because Project 5 is limited to local-area networks, we developed Wide-Area Project 5 (WAP5) [53] to enable analysis of distributed systems on a larger scale. WAP5 aids the development, optimization, and maintenance of wide-area distributed applications by revealing the causal structure and timing of communication in these systems. It highlights bottlenecks in both processing and communication.

WAP5 makes three contributions. First, it includes a new algorithm for inferring, *message linking*, for inferring causal path patterns from network traces. The message linking algorithm combines strengths from both of Project 5's inference algorithms. Second, it includes an interposition library for gathering application traces. Compared to network sniffing, interposition works without administrative privileges, incurs lower overhead, and provides richer semantic information in trace files. Finally, WAP5 includes our experiences tracing and analyzing three systems: CoDeeN [65] and Coral [22], two content-distribution networks in PlanetLab; and Slurpee, an enterprise-scale configuration and incident-monitoring system.

1.1.4 Pip

Many bugs in distributed systems reflect discrepancies between a system's behavior and the programmer's assumptions about that behavior. Our final debugging tool, Pip [52], includes a declarative language to allow programmers to express explicitly their expectations about their application's structure, timing, and resource consumption. Pip automatically compares actual application behavior to the programmer's expectations, revealing structural errors and performance problems.

Pip derives its model of application behavior from traces generated by system

instrumentation and annotation tools. Programmers annotate application source code manually or automatically. Subsequent runs of the application record system behavior, including processing tasks, network and local communication, and other interesting events. Pip includes visualization and query tools for exploring expected and unexpected behavior. Pip allows a developer to quickly understand and debug both familiar and unfamiliar systems.

1.2 Outline

Our research and experiences have led to four main contributions:

1. We introduce new debugging methodologies implemented in three debugging tools: Project 5, Wide-Area Project 5 (WAP5), and Pip. We are the first to extract causal paths from black-box distributed systems, and we are the first to apply automatic expectation checking to causal paths.

2. We describe three algorithms for inferring causality from communication events in black-box distributed systems.

3. We describe a language for expressing expectations about the structure and performance behavior of distributed systems.

4. We develop an understanding of the trade-off between disruptiveness and accuracy in distributed debugging tools.

The remainder of this thesis is organized as follows. The next chapter describes the context of our work, including an overview of debugging techniques and where they are most useful. Chapter 3 describes Project 5, Chapter 4 describes Wide-Area Project 5, and Chapter 5 describes Pip. Chapter 6 describes several related projects

in detail. Finally, Chapter 7 concludes with a summary of the contributions of this thesis and possible future research directions.

Chapter 2

Debugging

Programmers employ a variety of techniques to analyze and debug distributed systems. In this chapter, we describe desired features and traits a debugging technique can have. We describe some debugging techniques and a few representative tools that support them. At the end of the chapter, we describe the architecture of our debugging tools and place them in the context of the debugging techniques detailed here. A more thorough comparison of our tools to related work is in Chapter 6.

2.1 Goals for debuggers

The goal for any debugging technique is to find the root cause of bugs so that the programmer knows where to look and what to do to fix them. A good debugging technique enables this goal with a minimum of effort and disruption. Some debuggers even help uncover new bugs, in addition to helping programmers explore known bugs. This section explores these desired traits in more detail.

Minimal effort: Some debugging techniques require the user to recompile or even modify source code. Source code modifications help the user express expectations (e.g., invariants) or designate which parts of a program are important. However, source code modifications are time-consuming to add and can introduce new bugs if they are not correct. In some cases, they add run-time overhead, slowing a program down or increasing its memory consumption.

Recompiling or relinking an application's source code can incorporate debugging-friendly versions of components, particularly memory allocators. Recompiling can

also allow the compiler to add debugging symbols or profiling code. However, recompilation is not always an option: libraries or processes for which source is unavailable will not support debugging symbols, profiling code, or debugging components.

Lower user effort makes a debugging tool easier and sometimes less error-prone to apply. However, effort is a trade-off, and increased effort can yield finer granularity of control or inspection, or higher accuracy. Ideally, a tool should require no more up-front user effort than necessary to deliver the debugging functionality that the user needs.

Minimal disruption: Some debugging techniques require the user to restart the target system or selected components to enable or disable debugging. However, some techniques, including traditional debuggers, system-call tracers, and network sniffers, can be attached to a running system with no disruption. Some such tools run on the same host as the target system, which can interfere with performance, while network sniffers can run on other hosts and cause no interference.

As with effort, disruption is a tradeoff. Restarting an application or slowing it down might be required for some debugging tools to operate, but it might not be acceptable in a production system. A debugging tool should only disrupt an application as much as necessary for given debugging functionality.

Online operation: Debugging is often part of an edit-build-debug cycle. The shorter this cycle is, the more efficient a programmer is likely to be at removing bugs. One way to shorten the cycle is to allow the programmer to debug an application as it runs, rather than running a test to completion and then analyzing the results. However, online operation can require significant computing power and can interfere with time-sensitive applications.

Post-mortem analysis: Many bugs are not easily repeatable. Debuggers that can explore the cause of a crash after it happens can save the user the trouble of rerunning the program to recreate the bug. Traditional debuggers can analyze core files as long as the target program did not corrupt its memory space when crashing. Some other debuggers can operate offline using trace files, if tracing was enabled during the program execution.

Low-level control: Many bugs can be traced to a single incorrect line of code or incorrectly assigned variable. Some debuggers allow programmers to examine or even change the values of individual variables, or to step through code one line at a time. However, such fine-grained control is not useful or appropriate for bugs that depend on real-time system execution, like race conditions and performance bugs.

Scalability: The counterpart to low-level control is scalability. A debugger, particularly for distributed systems, must avoid overwhelming the user with a huge state space too many threads of execution. A good high-level debugger aggregates data values or program behavior over space (i.e., over many threads or hosts) and over time. Many large-system debuggers provide visualizations of program flow and component or host interaction.

Backwards exploration: Many bug symptoms are far removed from their causes in space or time. For example, careless access to heap variables, including dereferencing a variable after freeing it, might cause heap corruption that doesn't have any visible symptoms until the next memory allocation. Techniques for backwards exploration can help a programmer explore the history of data values known to be incorrect. Two techniques for backwards exploration are time-travel debugging and causality analysis. Time-travel debugging supports stepping backwards through code

and setting watchpoints on previous changes to variables. It is implemented by running the target system in a simulator or a virtual machine where previous states are stored and can be retrieved as needed. Causality analysis reveals the communication events or variable assignments leading up to the current program state. Causality analysis can be implemented by logging events and their causal relationships or by using formal languages that enable direct, static analysis.

Error detection: Some debuggers help programmers explore the causes of known errors. Other debuggers help programmers find (and then explore) previously unknown errors. Debuggers can find errors through statistical analysis or by letting the programmer express expectations about behavior. Statistical analysis looks for unusual behavior and reports it; the programmer decides if the reported behavior indicates real errors. Expectations are boolean statements about program state that are evaluated at specified times. Standard *assert* statements are one simple way of expressing expectations that can be locally evaluated. A failed assert statement causes a core dump, which can then be examined with a debugger. Other debugging systems allow more complex expectations that refer to performance, communication, or processing order in addition to simple assert statements.

2.2 Debugging techniques

In this section, we introduce several existing debugging techniques used to troubleshoot distributed systems. Each has some of the above traits, but none exhibits all of the traits.

Figure 2.1 and Table 2.1 summarize the lessons of this chapter. Figure 2.1 shows the systems described below in terms of two interesting metrics: their intrusiveness and accuracy, and the size of the systems they can debug. Intrusiveness, detail, and

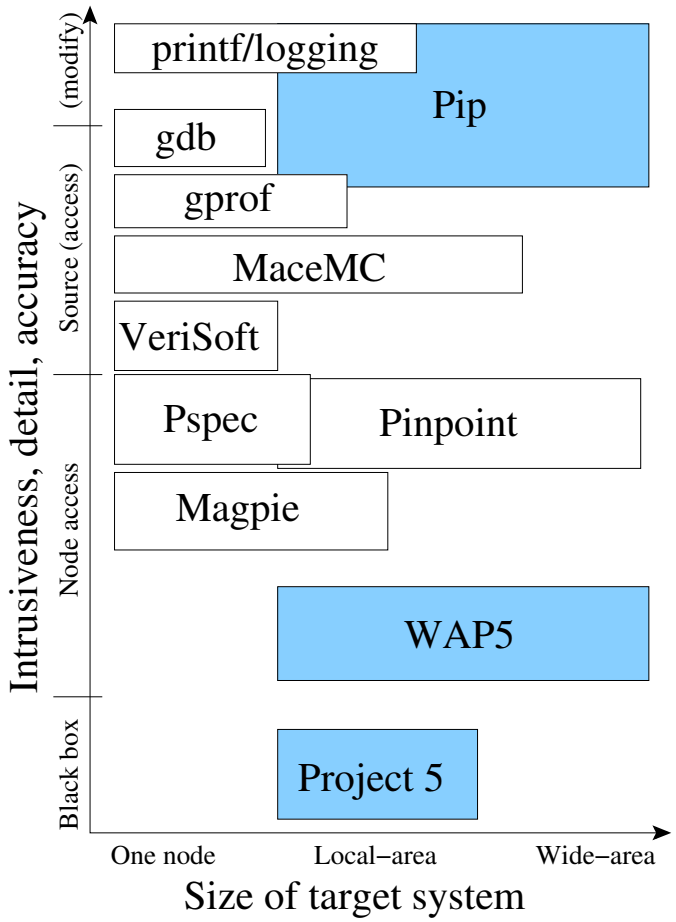


Figure 2.1: Classification of selected debugging systems by target system size and level of effort and disruption.

accuracy are a trade-off: a system that requires more effort to apply and use will normally allow finer-grained analysis and control. System scale is also a trade-off: a tool for debugging large numbers of nodes must discard some detail to remain efficient and avoid information overload.

Table 2.1 summarizes the kinds of systems for which each of our tools and each of the other approaches is most useful. Considerations beyond size and source code access are important. For example, only GDB can analyze a standard core dump. PSpec relies on applications to perform their own logging, as does Pinpoint in some cases. Magpie is restricted to Windows but does not require targeted programs to log

Approach	Scenario
logging	bugs detectable with simple, localized log analysis
GDB and gprof	low-level bugs well illustrated by a single node; core dumps
model checking	small systems with difficult-to-reproduce bugs
PSpec	performance bugs in systems with detailed logging
Magpie	bugs in well-understood Windows applications
Pinpoint	node or component failures in high-throughput distributed systems
Project 5	local-area systems (esp. black-box systems) with performance problems apparent in many paths
WAP5	wide-area systems with performance problems apparent in many paths; systems where interposition is easier than sniffing
Pip	arbitrarily large systems for which source code is available, exhibiting structural bugs or performance problems

Table 2.1: Comparison of techniques for debugging distributed systems.

interesting events. WAP5 relies on interposition, which can make it easier to apply in cases where sniffing, which P5 normally requires, is impractical.

The remainder of this section introduces several debugging techniques in more detail.

2.2.1 Traditional debugging

Traditional debuggers like the GNU Debugger (GDB) [24] are mature and powerful tools for locating and explaining low-level bugs. Traditional debuggers run on a single node, allowing a programmer or troubleshooter to step through a targeted process one function, one line, or one machine-code instruction at a time. Programmers can examine or change variable values as the program runs and can set breakpoints to pause program execution when a certain point is reached or a given condition is met. Graphical debuggers like the Data Display Debugger (DDD) [16] or debuggers included with integrated development environments provide a more user-friendly interface and can allow visualization of variable values over time.

Using a traditional debugger normally requires recompiling an application's code

to include debugging symbols. Programmers can use a debugger either as a program runs or after it has crashed. The main advantage of traditional debuggers is that they provide fine-grained control. Programmers can examine or control a program on the level of functions, lines of code, or individual instructions. This strength is also a weakness: traditional debuggers provide no useful tools for aggregating or visualizing state information across nodes, and only rudimentary tools for visualizing state information changes over time.

Standard debuggers like GDB are also limited in terms of temporal context for a bug. A debugger can trap at a breakpoint, a watchpoint, or a signal, any of which can indicate a bug, but doing so does not display recent or future events related to the trap. Some debugging systems allow stepping backward in time or tracing past changes to a variable. Time-traveling virtual machines [37] allow GDB to examine past states of an operating system running in a virtual machine. Time-travel debugging is also possible using simulators [7]. Debuggers for functional programming languages can discover the history (i.e., data dependencies) of programmer-selected variables [6].

Like single-node debuggers, profilers like the GNU Profiler (gprof) [28] are mature and are limited to single-node use. Gprof instruments function calls in user code to identify call chains and to identify the frequency and duration of calls to each function. Although gprof produces analysis for only one node at a time, it is possible that multiple nodes' results could be aggregated offline. Still, gprof has no support for tracing multi-node operations through the network. It is more useful for tuning small blocks of code than distributed systems and their emergent behavior.

In practice, the dominant tool for debugging distributed systems has remained unchanged for over twenty years: print statements to log files. The programmer analyzes the resulting log files manually or with application-specific validators written

in a scripting or string-processing language. In our experience, incautious addition of logging statements generates too many events, effectively burying the few events that indicate or explain actual bugs.

Debugging with log files is feasible when bugs are apparent from a small number of nearby events. If a single invariant is violated, a log file may reveal the violation and a few events that preceded it. However, finding correctness or performance problems in a distributed system of any scale is incredibly labor intensive. In our experience, it can take days to track down seemingly simple errors. Further, scripts to check log files are brittle because they do not separate the programmer's expectations from the code that checks them, and they must be written anew for each system and for each property being checked.

2.2.2 Causal path analysis

Several recent tools use causal paths as the basis for debugging distributed or single-node systems. Each tool groups communication and processing events into causal path instances, then aggregates the path instances. All three of the tools introduced in this dissertation are causal path analysis tools. Pip is the only causal path analysis tool to support automatically checking programmer-specified expectations.

Causal path tools vary greatly in how they obtain events, how they combine those events into paths, how they aggregate paths, and how they present the paths to a user.

Probably the work most closely related to this dissertation is Magpie [4]. Magpie enables path-based performance analysis and anomaly detection for single-node and distributed systems. Magpie uses Event Tracing for Windows, built into the Windows operating system, to collect thread-level CPU and disk usage information. Magpie does not require modifying the application, but it does require an application-specific

event schema, written by an application expert, to stitch traced information into causal paths. Associating events on several machines via network communication is not straightforward, and most of Magpie's target applications run on one node or just a few nodes. Magpie identifies paths that exhibit unusual behavior, which the programmer is left to examine for actual bugs.

Another causal path tool is Pinpoint [12], a system for locating the node or component in a distributed system likely to be the cause of a fault. A fault is a user-visible error, which can be due to a hardware failure or a programming bug. Pinpoint focuses on faults rather than performance or correctness problems. However, it can be useful for debugging, particularly when a program bug causes a fault. Pinpoint constructs causal paths by annotating applications or platforms to generate events and maintain a unique path identifier per incoming request. Like Magpie, Pinpoint assumes that anomalies indicate bugs. Pinpoint uses a probabilistic, context-free grammar to detect anomalies on a per-event basis rather than considering whole paths. In other words, Pinpoint records which events follow any given event, with which probabilities. If a call to component A can be followed by calls to B or C, then a call from A to D will indicate a fault. Using event grammars significantly underconstrains path checking, which, as the authors point out, may cause Pinpoint to validate some paths with bugs.

Statistical approaches like Magpie and Pinpoint require large traces from which they detect anomalies. They assume that only anomalies can indicate invalid behavior. However, some bugs are present in common paths and would be missed by statistical debuggers. In particular, a performance bug can affect common-case behavior, in which case automated anomaly detection would not find it.

Any system that relies on external tracing infrastructure, including sniffing, interposition, or Event Tracing for Windows, is limited by what information it can get

using these techniques on unmodified applications. In practice, many causal path tools, including Project 5 and WAP5, entail a form of gray-box debugging. That is, they leverage prior algorithmic knowledge, observation, and inferences to learn about the internal working of an unmodifiable distributed system. In contrast, some causal path debugging tools, including Pip, assume the ability to modify the source for at least parts of a distributed system. Source code annotations provide richer information for exploring systems without prior knowledge and for automatically checking systems against high-level expectations.

All of the causal path tools explored here support aggregation to reduce the amount (and redundancy) of information presented to the user. For aggregation, they combine individual path instances into some sort of behavior summary. Magpie uses a clustering algorithm to organize all path instances into sets of similar paths. Pinpoint forms a probabilistic grammar indicating which components call which other components, with what probability. Project 5 and WAP5 aggregate all path instances that reach the same hosts in the same order. WAP5 introduces support for renaming hosts to combine paths that have similar functionality on different hosts. Pip groups path instances according to which programmer expectations they do or do not match. Additionally, Pip can aggregate performance measurements into mean values that can be reasoned about or into distributions that can be viewed and compared as graphs.

Finally, causal path tools support backwards analysis. When a path instance is found to be wrong, whether through statistical inference, user examination, or expectation checking, it inherently provides context. A path instance encompasses many related events, sorted by causality, location, and time. The programmer can examine what part of the path instance was unexpected and then examine all of the events that led up to it. This form of backwards analysis is not as powerful as time-traveling debuggers, but it is available without specialized execution environments,

and it often provides the necessary context with less effort.

2.2.3 Expectation checking

Several existing systems support expressing and checking expectations about performance, structure, liveness, or system state. In general, expectation checkers can detect bugs, but they operate at too high a level to pinpoint the block of code responsible. They can run online or offline and can depend on either log files or source code annotations. Many expectation checkers focus on monitoring host-level or system-wide properties like load, liveness, throughput, or end-to-end delay. Some expectation checkers focus on debugging.

One expectation checking tool intended for debugging is PSpec [51], which allows programmers to write assertions about the performance (specifically timing) of systems. PSpec gathers information from existing application logs and checks them against performance assertions specified by the programmer. The assertions in PSpec all pertain to the performance or timing of intervals, where an interval is defined by two events (a start and an end) in the log. PSpec does not explicitly support distributed systems, and it has no support for causal paths or for checking application structure in general.

Expectation checking tools in general have important advantages. First, unlike statistical approaches, they have perfect accuracy when reporting instances of unexpected behavior. Second, they are more flexible in terms of what anomalies they can detect. Third, they can find bugs even in common behavior that a statistical approach would classify as "normal." Finally, they serve as an external description of system behavior, which can help summarize and enforce desired properties over time.

2.2.4 Model checking

Model checking is an exhaustive approach to debugging, searching the entire state space of a program systematically to find any states that violate programmer-specified invariant conditions. The advantage of model checking is its ability to find uncommon and difficult-to-reproduce bugs. Model checking can find race conditions, deadlocks, and even security holes that would be nearly impossible to find through standard testing. Model checking even reports the sequence of states that led up to an error. The major disadvantage of model checking is its running time: the time required to check a program's entire state space is exponential in the number of decisions the program makes. Decisions can be implicit or explicit and are frequent: the order of timer events and message delivery, the content of messages, the scheduling of threads, and the generation of random numbers. In practice, model checking only works for short runs of simple programs or for hand-picked portions of larger programs.

Model checking can operate on a program specification or on the program source itself. Most large systems do not have a formal specification, or the implementation can differ from the specification; thus, we are most interested in implementation-based model checkers like VeriSoft [26] and MaceMC [36]. VeriSoft works with programs written in C and requires source code to be recompiled with VeriSoft's headers and library. MaceMC works only with programs written in Mace [45], a domain-specific language for building distributed systems. It takes advantage of properties of Mace to provide faster, deeper exploration of a program's state space.

2.3 Proposed approaches

The three new causal path debugging tools we describe here operate at a higher level—i.e., a coarser granularity—than most traditional debuggers. They focus on communication events and on a small number of tasks that happen in response to a

received message. They do not focus on variable values or individual lines of code, or even on individual function calls. Thus, they can make debugging much larger, more complex applications manageable, but they do not always produce diagnostics specific enough to pinpoint a programming error. Instead, they identify a misbehaving node or a block or module of code containing an error, after which a traditional debugger can be used.

All three of our debugging tools operate offline. That is, the user gathers a trace from a running system, then stops the system (or at least stops the tracing) and analyzes the trace. Any of these tools could be modified to monitor systems as they run (online analysis), given a network fast enough to gather event information to one host and a host fast enough to perform the analysis. Online analysis could help the user get results faster and would enable long-term monitoring of running systems.

2.3.1 Project 5

Project 5 enables analysis of unmodified, black-box applications. It infers causal paths from communication traces, normally gathered using one or more network sniffers. It then aggregates these causal paths and displays them so that the user can look for performance bottlenecks and unexpected behavior.

Project 5 treats a target distributed system as a collection of black boxes. The user does not need access to change, examine, or recompile any source code. The user does not even need access to run debugging software on any hosts involved in the application. The event gathering, using a network sniffer, can be entirely external to the application. This black-box approach has the twin benefits of low up-front user effort and low disruptiveness. The user need only start a network sniffer and begin gathering traces.

From network traces, Project 5 uses two inference algorithms to extract causality.

The *nesting* algorithm infers individual causal path instances that are later aggregated into path patterns. Nesting operates on the assumption that communication has RPC (call-return) semantics and that a call B caused by a call A will begin after A begins and end before A ends—i.e., B is nested in A. The second algorithm is *convolution*, which uses signal-processing techniques to discover causal path patterns directly. Convolution examines all messages into a host and all messages out of the same host, looking for correlations. If an incoming message commonly pairs with an outgoing message t time later, then a causal relationship might exist. Convolution does not require RPC semantics, but it requires longer traces and has much longer running times.

Project 5 is normally limited to local-area distributed systems. It assumes that both network delays and clock offsets (the absolute difference between unsynchronized clocks) will be smaller than most processing delays, which is only likely to be true in local-area networks. Project 5 also usually uses sniffer traces, which are much harder to obtain in wide-area systems.

Relative to Magpie and Pinpoint, Project 5 is more widely applicable. It does not rely on any particular platform or source code annotations to produce traces. It also does not require event schemas written by application experts; instead, it infers causal relationships statistically. While Project 5 aims to help programmers explore applications and find performance bugs, Pinpoint aims to detect and locate faults.

2.3.2 Wide-area Project 5

WAP5 shares some of its techniques and all of its high-level goals with Project 5. WAP5 aims to enable exploring and performance debugging in black-box, wide-area distributed systems. However, WAP5 introduces a new trace-gathering technique and a new inference algorithm that make it applicable to a different set of systems.

WAP5 uses library interposition instead of network sniffing, meaning that the user must restart each component using a library that captures communication system calls. Source code is still not needed, but the user does have to install the library on each system and restart each component to enable or disable debugging. Interposition, however, enables debugging at a somewhat finer granularity: it can detect individual threads or processes, as well as most interprocess communication. Project 5's granularity is entire hosts.

WAP5's inference algorithm, *message linking*, extracts individual causal path instances like nesting, but does not require or assume RPC communication semantics. Thus, WAP5 can be applied to multicast systems, recursive distributed hash tables (DHTs), and other systems that do not strictly pair call and return messages.

2.3.3 Pip

Pip is applicable to any system for which source code is available. We designed it primarily for checking distributed systems, but it can also check single-node systems. Pip relies on annotations in either the application or an underlying middleware system. Thus, it is only useful if the programmer can obtain, change, and rebuild the application source code. It requires substantial up-front effort and cannot be applied to commercial software or to hardware components.

The granularity of debugging in Pip is arbitrarily fine. Programmers can add annotations for individual functions and variables at the expense of log size and execution speed. In practice, the granularity of the annotations is limited by two factors: per-annotation overhead and the cost of reconciling and checking a trace. (See Sections 5.4.2 and 5.3.5, respectively, for details.) Annotations at too fine a granularity will lead to slow program execution and long analysis times. Pip is more useful for finding components or functions that are misbehaving, which can then be

examined further using traditional, single-node debugging tools.

Like PSpec and model checkers, Pip relies on explicit, programmer-written expectations to detect and locate unexpected behavior. However, PSpec supports only timing expectations, primarily expectations about how long an operation will take or how often it will occur. Pip supports these same timing expectations, as well as performance expectations for other metrics and structural expectations for the ordering and placement of operations. Model checkers support arbitrary expectations, possibly more general than Pip's, but they can only check short runs of small programs. Model checkers explore a program's state space exhaustively, while Pip checks only the states that actually occur. Thus, Pip can check more and longer paths, but it will miss bugs such as race conditions that occur only rarely and are hard to reproduce.

2.4 Summary

Many existing systems support debugging on one or several nodes. Several systems have employed some of the same techniques we use here: causal paths, black-box causality inference, expectation checking, and interposition. Project 5 was the first system to support inferring causal paths from black-box network traces. WAP5 was the first system to apply black-box causal path inference to wide-area systems. Pip was the first system to apply automatic expectation checking to causal paths. In the following three chapters, we will describe these systems in greater detail.

Chapter 3

Project 5: Black-Box Debugging

This chapter describes the first of three debugging tools, Project 5. Project 5 uses two novel algorithms to infer causal path patterns from communication traces of black-box distributed systems. These causal path patterns represent the structural and performance behavior of the target system, helping a programmer to find unexpected behavior and processing bottlenecks.

3.1 Overview

Many interesting and commercially important applications now consist of components distributed across several hosts and communicating over a network. Commercial web sites are often hosted on clusters of computers, ranging in size up to thousands of nodes. Any given request will touch several largely independent components: a web server, an application server, a database server, an authentication server, a credit card authorization server, and so on. These components are often black boxes: we have no advance knowledge of their internal functioning, and we cannot modify their behavior to support accounting or tracing.

Such distributed systems are prone to performance problems. One or more components might add significant delays to some or all requests that they process. From the outside, these problems appear only as added end-to-end latency. In this chapter, we describe Project 5, a system for describing the causal paths that requests take through a distributed system and identifying how much each individual component contributes to total request delays. Project 5 uses message-level traces to build a graph of request paths through the application to help in isolating bottlenecks.

Project 5 requires no advance knowledge of application implementation and little or no knowledge of the protocols used for transmitting messages between nodes.

Project 5 consists of three phases:

1. **Exposing and tracing communication:** In this *online* phase, we gather a complete trace of all inter-node messages for an operational system, under real or synthetic load. Depending on the means of communication, we might obtain a single global trace, or a set of per-edge traces for each pair of communicating nodes.

2. **Inferring causal paths and patterns:** In this *offline* phase, we post-process a trace using one of several algorithms. The algorithms must cope with traces that are potentially quite large and noisy (e.g., with missing entries, extraneous calls, timeouts and retries, unsynchronized clocks, etc.). Although this phase need not meet real-time performance goals, our algorithms must be reasonably efficient in time and space. However, the algorithms need not be fool-proof, because our tools are meant to help humans debug systems, not for automatic control. They should be robust enough that they generate relatively few false negatives (i.e., missed important causal path patterns) or false positives (extraneous, incorrect patterns). Section 3.2 describes our algorithms.

3. **Visualization:** A full system should also provide appropriate ways to visualize the results. However, our research so far has only partially addressed this issue.

An abstract trace format forms the connection between the first and second phases, which allows us to use several different techniques to gather traces, and several different offline algorithms. The trace contains, at a minimum, a <timestamp, sender, receiver> tuple for each message, but might include some additional information. Because the information we need depends on which algorithm is used, we

will describe the algorithms before describing the specifics of gathering traces.

3.2 Inference algorithms

Our key challenge in Project 5 is to infer causal path patterns and latencies from message traces. We have developed two distinct algorithms to perform this inference. The first, *nesting*, depends on the use of RPC-style communication and operates on individual messages in the trace. The second, *convolution*, is able to handle free-form message-based communication, and uses signal-processing techniques to extract causal information from traces.

3.2.1 The nesting algorithm

The nesting algorithm assumes that nodes communicate using remote procedure call (RPC) semantics, in which one node calls upon another to execute some task or retrieve some value and the callee eventually replies with the result. RPC calls are nested; that is, while processing an incoming call, a node is likely to initiate one or more outgoing calls to other nodes. For example, a web server might communicate with an authentication server to process a login, resulting in the authentication server communicating with a database server to retrieve usernames and passwords. Here, we say that the database request is *nested in* the authentication request. Request Y is nested in (and potentially caused by) request X if X's destination is Y's source, X's call was transmitted before Y's call, and X's return was received after Y's return. In Figure 3.1, the $B \rightarrow C$ and $B \rightarrow D$ calls are nested inside the $A \rightarrow B$ call. A subset of the nesting relationships are determined to be causal, after which individual causal relationships are chained together to form whole path instances. The output of the nesting algorithm is a collection of all unique paths and a count of how many times each path appears.

The nesting algorithm examines traces primarily at a local level. Each call-return pair is examined individually and designated either as caused by another call-return pair or as the root of a request. One major advantage of this local view is that it can capture unusual effects. If 95% of a trace behaves one way but 5% of it behaves another way, the nesting algorithm may find the 5%. Finding unusual cases is particularly important because they are likely to be "interesting" parts of the trace. For example, a cache miss may occur only 5% of the time but may use otherwise unused nodes and will likely take much longer than cache hits. The disadvantage of path-level examination is that it leads to higher levels of noise. A trace with unsynchronized clocks or high parallelism may lead to some incorrect inferences about causation.

Any causality determination algorithm makes a trade-off between false positives and false negatives. False positives are paths that are not truly causal but appear in the output. False negatives are paths—usually uncommon ones—that are causal but do not appear in the output. Convolution errs on the side of false negatives because uncommon paths disappear into the noise. Nesting errs on the side of false positives because every path that occurs even once will appear in the output.

Implementation details

A *call pair* is a tuple describing a single call from one node to another and its matching return. It contains the timestamps of the two messages and the names of the nodes. The nesting algorithm consists of four steps:

1. Find call pairs in the trace.

2. Find all possible nestings of one call pair in another, and estimate the likelihood of each candidate nesting.

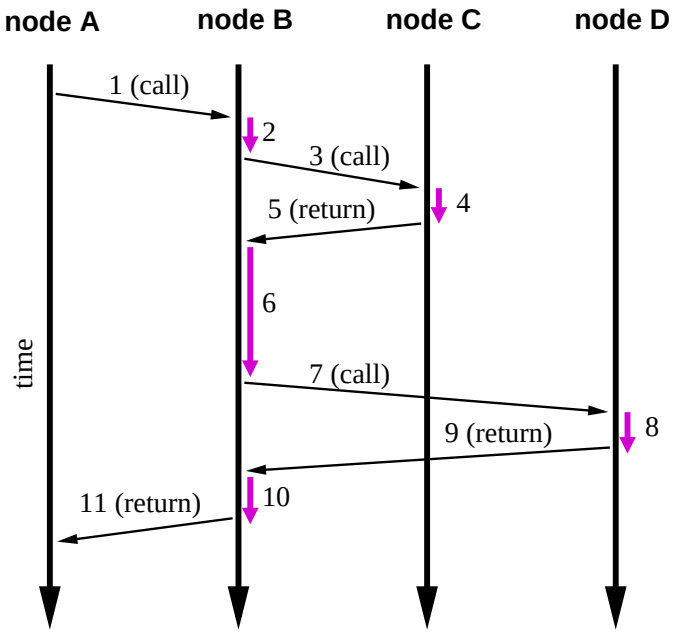


Figure 3.1: Timelines for an example system with four nodes.

3. Pick the most likely candidate for the causing call for each call pair.

4. Derive call paths from the causal relationships.

We first illustrate the algorithm using the example in Figure 3.1. Step (1) groups trace entries 1 and 11—the call $A \rightarrow B$ and the return $B \rightarrow A$—into call pair $(A, B, 1, 11)$; entries 3 and 5 into call pair $(B, C, 3, 5)$; and entries 7 and 9 into call pair $(B, D, 7, 9)$. (For ease of explanation, in this example we use the message numbers as the timestamp values.)

Step (2) examines each call pair to determine the set of calls that might have caused it. Here, $(B, C, 3, 5)$ and $(B, D, 7, 9)$ both occur between the beginning and end of $(A, B, 1, 11)$. $(A, B, 1, 11)$ is the only call that encloses $(B, C, 3, 5)$ and $(B, D, 7, 9)$. In a more complex example, a call pair might be nested within several different "parent" calls, which would have to be ranked by estimated likelihood.

Step (3) chooses the most likely parent call for each call pair in the trace as its causal parent, based on aggregate information from all other call pairs between the

same nodes. Step (4) again examines each call pair and creates a call path starting from each call pair that was not nested in any other call pair. Since $(A, B, 1, 11)$ is the parent for two call pairs, it creates the path $A \rightarrow B \rightarrow C; D$. The call pairs $(B, C, 3, 5)$ and $(B, C, 7, 9)$ do not initiate paths because they are nested in $(A, B, 1, 11)$.

Scoring potentially-causal nestings: A call pair (B, C, k, l) might be nested in many (A, B, i, j) call pairs, but it is only directly caused by one such parent. The nesting algorithm uses a scoreboard to estimate the likelihood that each nesting relationship is really a causal relationship. The scoreboard records the prevalence, in the entire trace, of the delays between the two call messages in a potentially-causal nesting.

The scoreboard represents the set of all nesting-delay tuples (A, B, C, δ), where δ is the time difference between the call from A to B and the subsequent call from B to C; each tuple has an associated value. The scoreboard entries for a given nesting thus form a histogram of these delay values. However, each increment to a histogram count is weighted by the number of possible parent calls: if there are N possible parent calls for a given child call, then the scoreboard value for each of these N tuples is incremented by $1/N$.

We actually store each histogram as a set of exponentially-sized bins, efficiently representing the large range of delay values that might appear in real traces. We find that 325 bins (indexing the histogram by $\log_{1.05} \delta$) gives reasonably accurate results for intervals between 1 ms and 2 hours, a range that encompasses all delays discovered in our traces. A larger range can be achieved easily by allocating more bins. The number of histograms is equal to the number of (A, B, C) triples such that a call $B \rightarrow C$ is nested in a call $A \rightarrow B$ at least once. This number, which is independent of trace length, is at most n^3, for n nodes; in practice it is significantly lower.

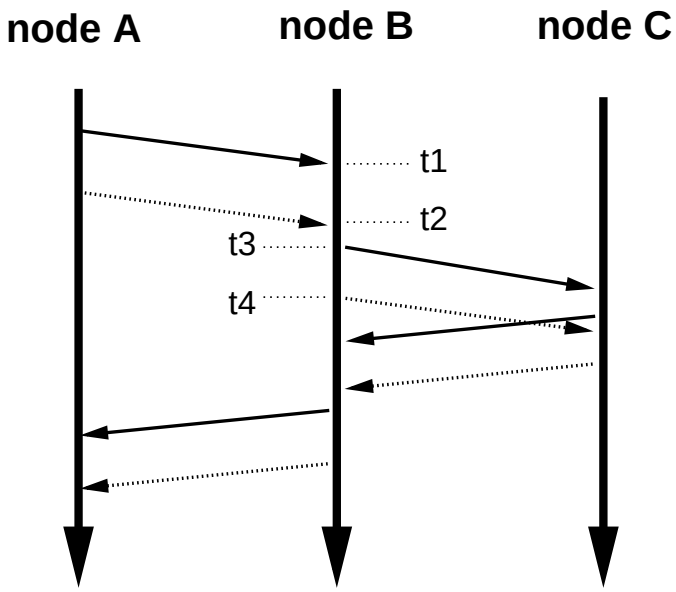


Figure 3.2: Example of parallel calls.

After scoring all of the call pairs, we optionally smooth the histograms by convolving them with a Gaussian normal curve. Smoothing helps accuracy when the hosts in the trace have unsynchronized clocks (i.e., non-zero clock offset). It has little effect in traces with well synchronized clocks.

Figure 3.2 shows an example in which two $B \rightarrow C$ calls are each nested in two $A \rightarrow B$ calls, creating four possible sets of parent-child pairings. However, the "medium-length" delay ($t_3 - t_1$ and $t_4 - t_2$) occurs twice as often as the "long" delay ($t_4 - t_1$) or the "short" delay ($t_3 - t_2$). Thus, the histogram for (A, B, C) has a peak at the medium-length delay.

Choosing unique parents: After building the scoreboard, the nesting algorithm chooses the most likely causal parent for each individual call pair. The inference that any given nesting is a causal relationship is based on the scoreboard and on simple heuristics about the parent's calls to other possible children. For each call pair (B, C, t_2, t_3) in the trace, we consider each possible parent (A, B, t_1, t_4) and generate a score for the relationship. The raw score is simply the value of $(A, B, C, t_2 - t_1)$ in

the scoreboard. The raw score is then scaled using three *penalties*:

- Overlapping-child penalty: We count the number of children $c_{overlap}$ already assigned to the given parent that overlap in time with the current call pair, and multiply the score by ${c_{overlap}}^{-x}$.
- Same-child penalty: We count the number of children c_{same} already assigned to the given parent that have the same destination as the current call pair, and multiply the score by ${c_{same}}^{-y}$.
- Generic-child penalty: We count the number of children c_{any} already assigned to the given parent, and multiply the score by ${c_{any}}^{-z}$.

The parameters x, y, and z are configurable. In our experiments, we get the most predictable, near-optimal performance across all workloads with $x \approx 2$ and $y = z = 0$. An experiment to determine good values for x is described in Section 3.4.2. Some traces suggest other values for x, y, and z than those specified here. For example, a trace in which a node makes several calls in parallel should have x near zero. A value of exactly zero is often worse, because a non-zero penalty helps prevent too many parallel children from being assigned to a parent. A trace in which a node makes parallel calls to different children would benefit more from a non-zero y than a non-zero x. Even a non-zero z can be useful; for example, a trace in which A calls B, which makes a short call to either C or D but never both will benefit most from the generic-child penalty, which broadly penalizes all assignments of more than one child. The results for this scenario are shown in Figure 3.3. Each line represents varying a single penalty from 0 to 4 while holding both others constant at 0.

In Figure 3.2, each $B \rightarrow C$ child call has two possible $A \rightarrow B$ parents, but each child has one parent for which the scoreboard includes a peak at the medium-length delay ($t_3 - t_1$ and $t_4 - t_2$). Based on this inference, the nesting algorithm assigns

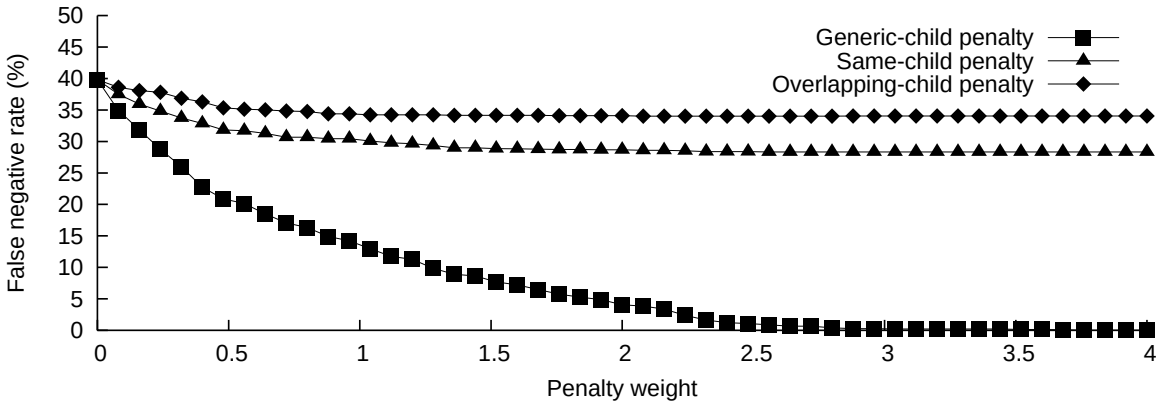


Figure 3.3: The effect of each penalty on a trace containing the paths $A \rightarrow B \rightarrow C$ and $A \rightarrow B \rightarrow D$.

each $B \rightarrow C$ child call pair to one of the $A \rightarrow B$ parent call pairs, as shown with the solid and dashed lines in the figure. The overlapping-child penalty favors assigning the two children to different parents. Tie scores when considering parents for a given child are broken by assigning the child to the earliest tied parent.

Creating and aggregating call paths: The final step in the nesting algorithm coalesces the causal relationships found in step (3) into call paths, and keeps aggregate latency statistics for each path pattern. The *latency* of a node is the total time spent in processing at that node, including at any nodes that it calls. The *call_delay* of a node is computed as the time between the call to its parent and the inferred causally-related call to this node.

Distributions of node times and call delays: In some cases, the processing time in a node has an interesting distribution not captured well with a simple mean and standard deviation. The nesting algorithm uses exponentially sized histograms similar to those in the parent-selection scoreboard to store the distributions of processing times and call delays. The code emits these distributions in formats suitable for plotting with Gnuplot or for viewing with the Project 5 visualization tool.

Time and space complexity

Finding call pairs is linear in both time and space in the size of the trace: each trace entry is examined once and put into one call pair. Finding nested call pairs is linear in both time and space in the total number of nesting relationships. This number is the product of the number of trace entries and the mean per-node parallelism during the trace. We define per-node parallelism as the average number of candidate parents for each child. However, finding nested call pairs requires that the list of call pairs be sorted, which takes $O(m \lg m)$ time for m messages in the trace. Creating and aggregating call paths is linear in the number of messages in the trace: each message either begins a new call path or belongs to exactly one existing call path. Overall, the algorithm takes $O(m \lg m + mp)$ time, for m messages and p mean per-node parallelism. In practice, the mp term dominates.

3.2.2 The convolution algorithm

Unlike the nesting algorithm, our second algorithm finds causal relationships by considering the aggregation of multiple messages, rather than by examining messages individually. The algorithm separates a whole-system trace into a set of per-edge traces, and treats each of the per-edge traces as a time signal. The central idea of the algorithm is to convert traces into time signals and then use signal processing techniques to find the cross correlations between signals. It considers the trace of messages from A to B separately from the trace of messages from B to A, so this algorithm can be used on traces of free-form message-based communication, not just RPC-style traces.

The results of this algorithm are directed graphs, in which a node might appear several times (e.g., $A \rightarrow B \rightarrow A \rightarrow C$). To avoid confusion between the graph of the distributed system itself and the output graph, we use the term *vertex* for the graph

vertices and *node* for the components of the system.

Given a root node i, the algorithm first creates a vertex x_i in the output graph. Then it considers the messages with source i: for each different destination node j in those messages, there is a causal relationship between i and j, so the algorithm creates a vertex x_j and adds an edge from x_i to x_j.

The algorithm then continues the path from j. The algorithm finds the sets of messages with source j that appear to be caused by the messages from i to j. Each set contains messages with a single destination node k and a common delay d: the set indicates that a message from j to k was sent exactly d time units after a message from i to j. For each set, it adds a vertex x_k with label k and edge (x_j, x_k) with label d to the graph, and recursively continues along the path from k (i.e., it creates the graph in depth-first order).

Finding the set of messages leaving j that are caused by messages entering j is the heart of the algorithm. The algorithm computes the causal delays d, which are *time shifts* between the messages arriving at j and the messages leaving j. To find these time shifts, it converts the messages V from i to j into an indicator function $s_1(t)$. This function is defined to be

$$s_1(t) = \begin{array}{l} 1 \text{ if } V \text{ has a message in time interval } [t-\epsilon, t+\epsilon] \\ 0 \text{ otherwise} \end{array}$$

where ϵ is a small fixed constant and $[t-\epsilon, t+\epsilon]$ is a short closed interval. It similarly converts all messages sent from node j into an indicator function $s_2(t)$. It then computes the *cross correlation* $C(t)$ of $s_2(t)$ and $s_1(t)$. $C(t)$ is defined to be the *convolution* of s_2 and the time inverse of s_1[1], which is why we call this the "convolution algorithm." Roughly speaking, $C(t)$ has a spike at position d if and only

[1]The convolution of two functions $f(t)$ and $g(t)$ is another function, denoted $f \otimes g(t)$, defined by $f \otimes g(t) = \int_{-\infty}^{+\infty} f(u)g(t-u)du$. The discrete version of this definition is $(f \otimes g)_i = \sum_{j=-\infty}^{+\infty} f_j g_{i-j}$.

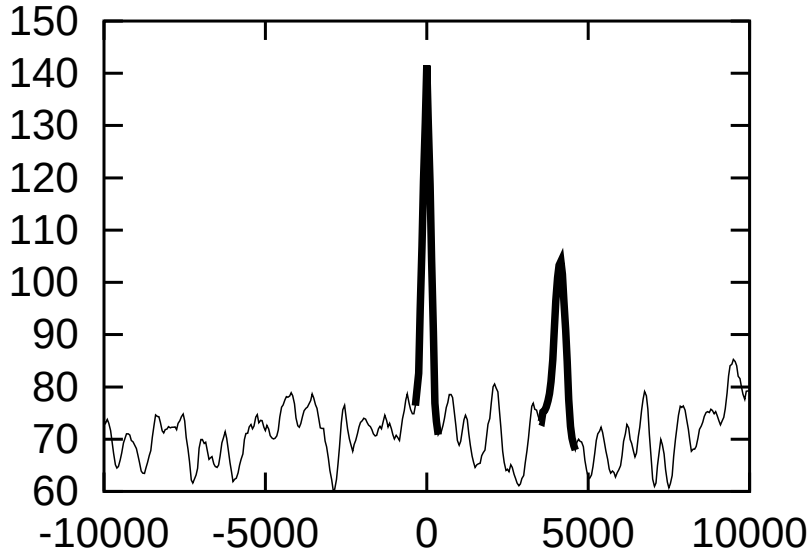


Figure 3.4: Example of convolution output, showing two spikes with bold lines. The x-axis represents the time shift; the y-axis roughly estimates the number of messages matching a given shift.

if $s_2(t)$ contains a copy of $s_1(t)$ time-shifted by d. Figure 3.4 shows the convolution for an example $s_1(t)$ and $s_2(t)$.

To detect the spikes, if any, in $C(t)$, we compute the mean and standard deviations of C. We consider a point to be a "spike" if it is a local maximum N standard deviations above the mean, where the parameter N is a small number (e.g., 4). There may be many such local maxima close together. Rather than consider each one to be a separate spike, we require at least one point that is less than S standard deviations above the mean between spikes, where $S < N$ is another small number (e.g., 3). Among the candidate points for a given spike, we choose the largest to represent the spike.

Discretization of the indicator function

The definition for $s_1(t)$ assumes that t is a continuous time parameter. In practice, we need to discretize time. To do so, we choose a time quantum μ and then treat t

as an integer multiple of μ. The definition of $s_1(t)$ is then modified as follows:

$$s_1(t) = \quad \text{square root of number of messages in } V \text{ during time interval } [t\mu, (t+1)\mu), \text{ where } t \text{ is an integer.}$$

Several discretizations are possible but the above definition produces the most accurate results and is what we implemented. Note that $s_1(t)$ can be represented by an array. When there are time quanta with lots of messages, if $s_1(t) = x$ and $s_2(t+d) = x$ then the (discrete) convolution of $s_2(t)$ and $s_1(-t)$ at position d includes an x^2 term. The square root in the definition compensates for this square. We similarly change the definition of $s_2(t)$.

Time and space complexity

The convolution algorithm must store the m messages in the trace, and the vectors containing discretized indicator functions. At any time, there is a constant number of such vectors. The size of each vector is bounded by $S = T/\mu$, where T is the duration of the longest trace and μ is the time quantum. Hence, the overall space complexity is $O(m + S)$.

The time complexity of the algorithm is proportional to the time to traverse the trace and the time to compute convolutions of discretized indicator functions. Convolutions of vectors of size S can be computed in time $O(S \log S)$ using fast Fourier transforms. The number of times the trace is traversed and a convolution is computed is proportional to the number e of edges in the output graph G. Hence, the overall time complexity is $O(em + eS \log S)$. In practice, we find that the second factor, $eS \log S$, dominates the running time.

3.2.3 Comparison of the two algorithms

Our two inference algorithms have different strengths and weaknesses. Often these strengths are complementary: sometimes one algorithm works better, sometimes the other. Here we contrast the algorithms in terms of their utility.

RPC vs. free-form messages

The nesting algorithm explicitly works only with systems that use RPC-style communication. The convolution algorithm can find causal relationships in any form of message-based system. The limited applicability of the nesting algorithm is not without benefits, though: because it assumes that a system is RPC-based, it provides a more concise representation of such systems than the convolution algorithm can.

Some common forms of RPC-based systems pose a problem for the nesting algorithm as we have implemented it, and currently can only be analyzed with the convolution algorithm. If a system forwards RPC calls or returns asymmetrically (e.g., A calls B, B forwards the call to C, and C replies directly to A) then we fail to detect this as a single RPC call. Also, if a called node replies to a call *before* issuing a causally-related subsequent call, there is no obvious nesting relationship between the two calls. This can happen when the second call is asynchronous, as in the case of write-back caching. Finally, the accuracy of the nesting algorithm is sensitive to accurate pairings of calls and returns. Some protocols provide these pairings, but in others, the algorithm must infer them, which significantly impairs the accuracy of the output.

On the other hand, the convolution algorithm has some drawbacks with RPC-style path patterns. Given a path pattern $A \rightarrow B \rightarrow C \rightarrow B \rightarrow A$, the algorithm will not only report this path, but also $A \rightarrow B \rightarrow A$. This is because there is a causal relation between $A \rightarrow B$ and $B \rightarrow A$. If a node appears many times on a path, the

algorithm will report a large number of derived paths that are not very interesting. It may be possible to automatically filter out such paths, while preserving legitimate paths, by using frequency counts. The nesting algorithm correctly finds the right number of instances of each pattern, as we show in Section 3.4.1.

Rare events

The convolution algorithm looks for spikes in the cross correlation of two signals. Therefore, it cannot be used to search for rare events, especially those with high delay variance.

The nesting algorithm explicitly analyzes every RPC message for its relationship with other messages, and therefore can find rare events. However, distinguishing the rare events of interest from the more frequent but uninteresting events is an unsolved problem. Also, the scoreboard mechanism described in Section 3.2.1 currently biases the algorithm away from rare events: they will be found most easily when there are few overlapping calls among the same nodes.

Detail required in traces

Our tools would ideally require no information about message formats. In practice, this goal means that the algorithms should use only information available from widely deployed standards with self-describing formats. The convolution algorithm effectively meets this ideal; it requires only timestamps and sender and receiver identifiers.

The nesting algorithm further requires that trace entries be marked as either RPC calls or returns. (In a few cases, this information can be inferred based on *a priori* knowledge of address formats, such as UDP's "well-known" port numbers.) The algorithm also performs much better if the trace system can extract call identifiers

from the RPC messages, to pair calls with returns.

Time and space complexity

As discussed in Section 3.2.1, the nesting algorithm runs in linear space and $O(m \lg m + mp)$ time in the number of traced messages m and the mean per-node parallelism p. Generally, the mp term dominates. As we will show in Section 3.4.3, practical running times are quite low—much lower than the duration of the traces themselves—and the space overhead is more likely to be the limiting factor.

The convolution algorithm, as discussed in Section 3.2.2, has space complexity linear in the length of the trace (measured either by message count or total number of time quanta, whichever is larger), with a modest constant factor. Running time is the dominant cost for the convolution algorithm; as we show in Section 3.4.3, it can be much slower than the nesting algorithm. In practice, there is a tradeoff between increased precision of the delay results (decreased μ) and longer running time.

Visualization

The two algorithms provide different visualizations, even when applied to the same trace. For RPC-based systems, the nesting algorithm provides a more compact output, because the convolution algorithm does not combine calls and returns into one graph edge.

3.3 Experimental framework

To help quantify the performance of both algorithms, we developed Maketrace, a synthetic trace generator. Maketrace has several advantages over real traces:

- **Determinism:** In Maketrace, each path is specified explicitly. As a result, we know exactly what output the algorithms should produce and we can compare

this "ground truth" to the experimental output to evaluate the accuracy of each algorithm.

- **Flexibility:** Maketrace makes it easy to create traces that differ in only one aspect: drop rate, noise ratio, parallelism, duration, or presence of a given path.
- **Repeatability:** Maketrace is a single, deterministic process. From a given configuration and random seed it will always produce the same trace.
- **Speed:** Maketrace generates synthetic traces rather than measuring a running system. Thus, it can run in substantially less than real time.

Maketrace configurations consist of several *tracelet* files and a *master file* that specifies how to combine the tracelets into a trace. Each tracelet file contains the messages forming a single causal path. A tracelet for the four-message path $A \rightarrow B \rightarrow C$ is shown in Figure 3.5. Each message in the tracelet is either a call or a return and has a sender, receiver, delay distribution, and call-return ID. The delay distribution is a Gaussian normal distribution specifying the amount of time between the given message and the next message in seconds. It is specified as $t_m \pm \sigma$, where t_m is the mean delay time and σ is the standard deviation. All delays below zero are truncated up to zero. We selected a Gaussian normal distribution for think times to give them a configurable amount of variation but also a dominant "normal" case.

The trace master configuration file specifies which tracelets to use and how to interleave them. A master file with two tracelets is shown in Figure 3.6. Each line indicates a tracelet to run, the think time between instances of the tracelet, how many threads of the tracelet to run in parallel, and when to start and stop threads running the tracelet. The think time is specified as the center and radius of a uniform distribution. Think-time distributions are truncated at zero so that no negative delays are generated. Think times between invocations are uniformly distributed so

# call/return	sender	receiver	delay mean	delay stddev	callpair-id
call	a	b	0.05	0.005	1
call	b	d	0.05	0.005	2
return	d	b	0.01	0.001	2
return	b	a	0	0	1

Figure 3.5: a-b-d.tl: sample tracelet file for the path $A \rightarrow B \rightarrow D$.

# template	think	radius	parallelism	start	stop
a-b-cc.tl	10	2	5	0	500
a-b-d.tl	10	2	5	250	750

Figure 3.6: Sample trace master file with two tracelets.

that they have no "normal" case, which would lead to apparent causality between the end of one invocation and the beginning of the next.

Maketrace can also simulate added noise and dropped packets. The amount of noise is proportional to the number of non-noise messages in the trace. Noise messages travel from a random source to a random (but different) destination selected from all sources and destinations in the tracelets. Noise messages are sent at random times with a 50/50 chance of being a call or return. To model dropped messages, Maketrace simulates an event capture queue with capture rate r events/sec and queue length s. Any event that overflows the queue is dropped, often leading to bursts of dropped events.

3.4 Experimental results

We performed experiments on the nesting algorithm to see how accurately it could identify paths in several difficult real-world scenarios. In this section, we describe the metrics we used to quantify the accuracy of the algorithm's output, the traces we used for experimentation, and the experiments we performed.

The goal of Project 5 is to highlight the most significant causal paths in a distributed system, either the paths most frequently called or the paths accounting for the majority of the delay. Our simplest metric captures this goal in a mostly quali-

tative fashion by asking whether or not the top N paths truly present in the system are the same as the top N paths found by Project 5. In nearly all cases, the nesting algorithm finds the correct top N paths.

Comparing the top N actual paths to those identified experimentally does not have fine enough granularity to detect small changes in the accuracy of an experiment. Therefore, the results presented here compare counts of path instances instead. We compare the frequencies of all paths found against the true path frequencies. Incorrect instances found (either an incorrect path or a correct path found too often) are *false positives*, while missed instances are *false negatives*.

False positives and false negatives are represented as a percentage of true path instances in the system. (These percentages may be more than 100.)

3.4.1 Traces

The main purpose of our experiments was to estimate how well the nesting algorithm will work in a variety of real-world cases. We performed experiments on two real traces, one realistic manufactured trace, and several synthetic, corner-case traces. We also varied several factors that might decrease the accuracy of the nesting algorithm in order to quantify their effects.

PetStore

The first real trace we used for experiments was PetStore. PetStore is an example J2EE application running on top of JBoss v.3.0.6 server on a 2-CPU 1GHz Pentium III with Linux 2.4.9. We used Stanford's Pinpoint system to capture traces of EJB component calls and insert artificial delays. We performed experiments on three different Pinpoint/PetStore configurations: one with no delays, one with a constant delay of 50 ms in *mylist.jsp*, and one with a uniformly random delay of 1-100 ms in

mylist.jsp. The nesting algorithm had no trouble identifying the major paths in the PetStore trace and successfully found and characterized both artificial delays.

SFS

The second real trace was from the Self-certifying File System (SFS) project at MIT. The SFS trace combined information that SFS reported combined with a tcpdump packet trace. Although the paths in the SFS trace were not interesting (there were only a few paths, all short and linear), the quirks in the trace gathering highlighted some weaknesses of the nesting algorithm. First, the nesting algorithm does not handle forwarded requests. That is, if a client sends a call to a redirector, which forwards the request to a server, which replies directly to the client, the nesting algorithm will discard the operation. The ability to identify forwarded-request messages might make handling forwarded requests feasible, but doing so is future work. Second, the nesting algorithm does not handle asynchronous (delayed) calls such as write-back caching. If a server responds to the client but then performs an operation on behalf of the client at a later time, the operation will be assigned to another request or will be orphaned and will show up as the root of a new path (i.e., as spontaneous behavior). The nesting algorithm could handle asynchronous calls with a technique similar to its handling of clock offsets, but doing so is future work.

Multi-tier Maketrace systems

We used Maketrace to generate a variety of traces simulating the multi-tier configuration shown in Figure 3.7. Some of these traces have an additional 200 ms delay inserted at node *ws2*, between the serial calls to *auth* and to either *ap1* or *ap2*. The nesting algorithm found all 34 paths through the multi-tier system in its top 36 outputs, with false positives identified at positions 27 and 28. It had no trouble

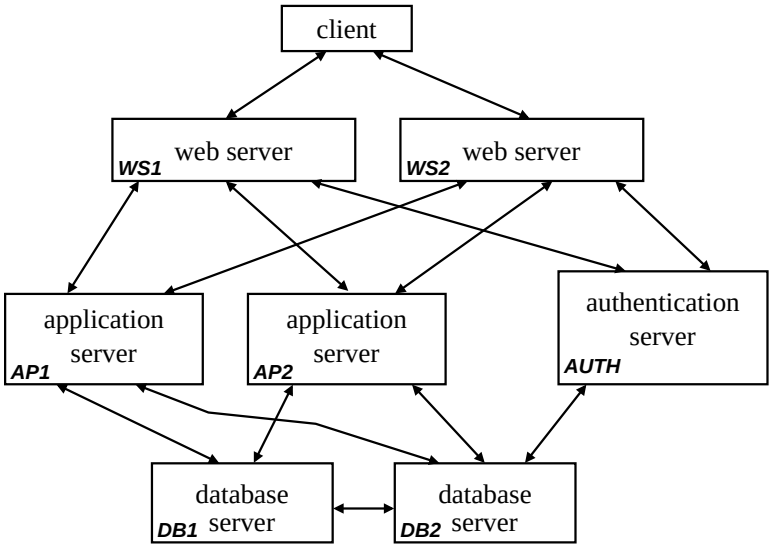


Figure 3.7: Multi-tier configuration.

identifying or locating the added delay.

Figure 3.8 illustrates the primary visualization for the output of the nesting algorithm. The output of the nesting algorithm is a collection of all unique causal paths found in the trace along with a count for each path. These paths can be converted to graphs using the program *dot* [23]. In these graphs, each tree represents a single causal path discovered by the nesting algorithm. Each node in the tree represents a host or component, and each edge represents a call. Node labels indicate the time in seconds spent at the node and all of the nodes it calls. Edge labels indicate the call delay between the call into the parent and the call from the parent to the child. Each tree also indicates the number of times the path occurred in the trace and how much total processing time it consumed.

Synthetic cases

We created several synthetic cases to try specific possible failings of the nesting algorithm. The four cases used here are shown in Figure 3.8. *Children-parallel* contains two requests issued in parallel, intended to violate the overlapping-children heuristic.

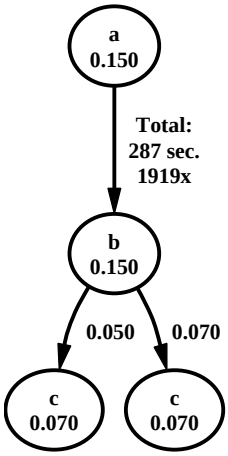
Children-0/2 contains paths with zero and two children, intended to confuse the algorithm into assigning some of the children to parents that make no calls (i.e., should have no children). *Children-d/cc* is similar to *Children-0/2*, with an added call to D. This scenario makes the $A \rightarrow B$ calls more similar in length and thus more likely to swap children accidentally. *Penalty-breaker* is more complex than either of the above because it contains two calls each to C and D as well as a path in which B does not call anything. The $A \rightarrow B \rightarrow C; C$ and $A \rightarrow B \rightarrow D; D$ paths frequently swap children. This system is called *Penalty-breaker* because it violates two of the three heuristics: generic-child and same-child.

3.4.2 Experiments

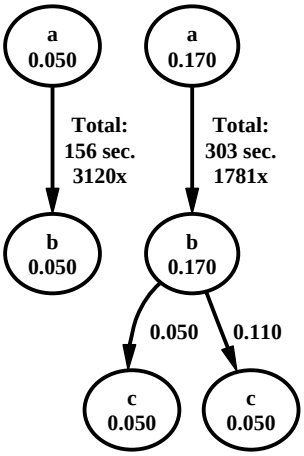
The goal of our experiments was to quantify, one at a time, the effects of challenging features a trace might have. In this section, we present each experiment, its justification, and the results.

Child penalties

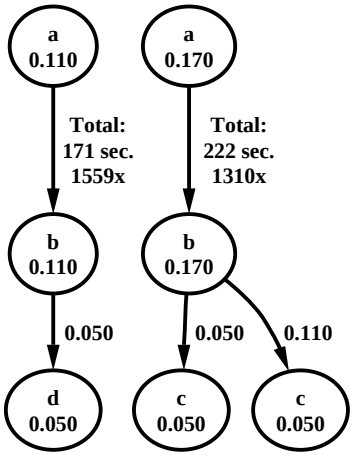
The nesting algorithm has three heuristics regarding the children a parent node is likely to have, as described in Section 3.2.1. These are free parameters, and we wanted to determine reasonable default values that would behave acceptably on all of the traces available. Figure 3.9 shows the results of varying the overlapping-child penalty with the other penalties set at zero for several different traces. As the figure shows, values of approximately 2 are the best when all the traces shown are considered. Another experiment, not shown here, varied the overlapping-child penalty with the other penalties given non-zero values. The results were sometimes slightly better and sometimes substantially worse; on average, it is best to use only the overlapping-child penalty and to leave the generic-child and same-child penalties at zero. In the absence



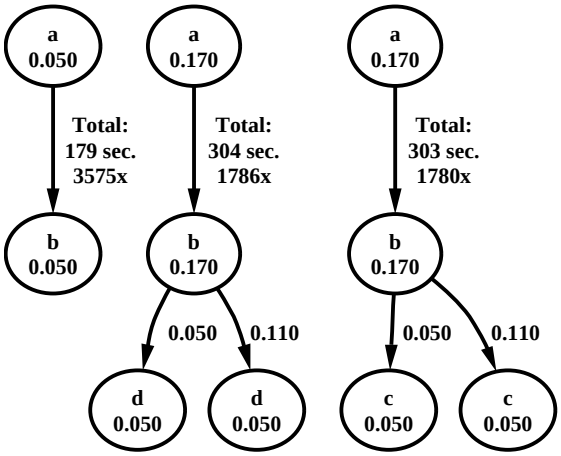
(a) Children-parallel



(b) Children-0/2



(c) Children-d/cc



(d) Penalty-breaker

Figure 3.8: Path patterns for pathological cases.

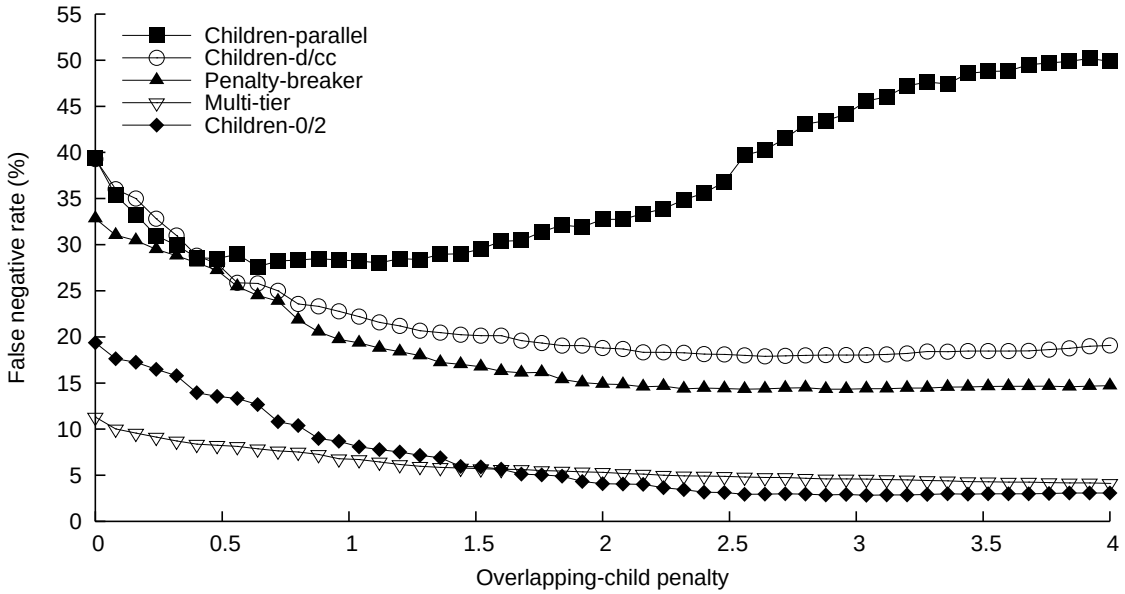


Figure 3.9: Effects of the overlapping-child penalty on accuracy for several traces.

of the overlapping-child penalty, either the generic-child or same-child penalty should be non-zero.

3.4.3 Results: execution costs

We measured run time and memory costs for the experiments in the previous sections. Note that neither program has been fully optimized, and the convolution algorithm presents several tradeoffs between accuracy and speed that may require some trial and error.

Table 3.1 shows the costs for the nesting algorithm, and Table 3.2 shows the costs for convolution. *Length* gives the trace length in messages; *duration* gives the elapsed time of the trace; *memory* gives the amount of data space allocated (not counting stack or code); *CPU time* gives the user-mode CPU time (kernel mode is negligible in all cases). The table also shows the (computed) *mean per-node parallelism* for the nesting algorithm, and the time quantum μ for the convolution algorithm. We ran the nesting algorithm on a 1.7 GHz Pentium 4 running Linux 2.4.20, and the

convolution algorithm on a 667 MHz AlphaServer running Tru64 UNIX V5.1.

We ran experiments to verify the scaling properties described in Sections 3.2.1 and 3.2.2. The nesting algorithm's run-time and space requirements should be $O(m \lg m + mp)$, where m is the trace length in messages and p is the mean per-node parallelism. To confirm this analysis, we ran the algorithm on the same trace configuration (multi-tier) with different trace durations and differing levels of parallelism. Additionally, we ran the algorithm on another trace (PetStore) with a similar length. The results, which are shown in Table 3.1, confirmed that the running time is roughly linear given constant parallelism (i.e., the mp term dominates), even with significantly different trace configurations and with significantly different amounts of output. The effect of parallelism is less clear. Analytically, both the running time and the storage requirements should increase at most linearly with increasing parallelism. In practice, the "high parallelism" trace appears to exhibit poor cache behavior, as it takes twice as long to run as would be expected.

The convolution algorithm's running time is mostly dependent on the trace duration and time quantum and not much on the trace length. We did not run the convolution algorithm on the longest traces in Table 3.1. With our current resources, the algorithm's running time becomes prohibitive if the trace duration is more than about 100,000 times the desired time precision (i.e., the time quantum).

Clock offset

To quantify the effects of clock offset we ran experiments with varying amounts of clock offset and with the nesting algorithm's offset compensation techniques turned on and off. The trace configuration we used was multi-tier, and we added offset at the *ws2* node (i.e., simulated *ws2*'s clock running fast). Figure 3.10 shows the results given a clock-offset window fixed at 30 ms and three different levels of compensation.

Trace	Length (messages)	Duration (sec)	Mean per-node parallelism	Memory (MB)	CPU time (sec)
Multi-tier (short)	20,164	50	1.793	1.5	0.23
Multi-tier	202,520	500	1.641	13.8	2.27
Multi-tier (long)	2,026,658	5,000	1.612	136.8	23.1
Multi-tier (low parallelism)	769,638	5,000	1.146	54.0	7.54
Multi-tier (medium parallelism)	770,344	500	5.116	54.2	11.15
Multi-tier (high parallelism)	775,254	50	45.057	132.1	233.61
PetStore	234,036	2,000	1.322	18	3.09

Table 3.1: Nesting algorithm running times for two traces with varying length and parallelism.

Trace	Length (messages)	Duration (sec)	μ (secs)	Memory (MB)	CPU time (sec)
Multi-tier	202,520	500	0.1	0.2	6684
PetStore	234,036	2,000	0.2	26	12780

Table 3.2: Convolution algorithm running times for two of the same traces as in Table 3.1.

The choices for compensation are none, offset window only, and offset window with smoothing. The smoothed offset window is best in all cases. An offset window of 30 ms is sufficient to compensate for up to 40 ms of clock offset, most likely because the most prominent feature is 10 ms. With offset greater than $10 + 30$ ms this feature begins to disappear.

The final line in Figure 3.10, marked "Rev," shows the results when we subtracted the given amount of offset from *ws2* instead. That is, we simulated *ws2*'s clock running slow rather than fast. There, the prominent delays are both 10 ms and 50 ms; thus significant degradation appears at 40 ms and 80 ms given an offset window of 30 ms.

In many cases, we may be able to estimate the worst-case clock offsets. However, at times we may have to guess and pick a offset-window size that may not be exactly matched to actual offsets. We ran an experiment in which we varied the offset amount (again simulating *ws2*'s clock running fast) and the offset-window size independently. The results are shown in Figure 3.11. Each curve represents a fixed amount of offset,

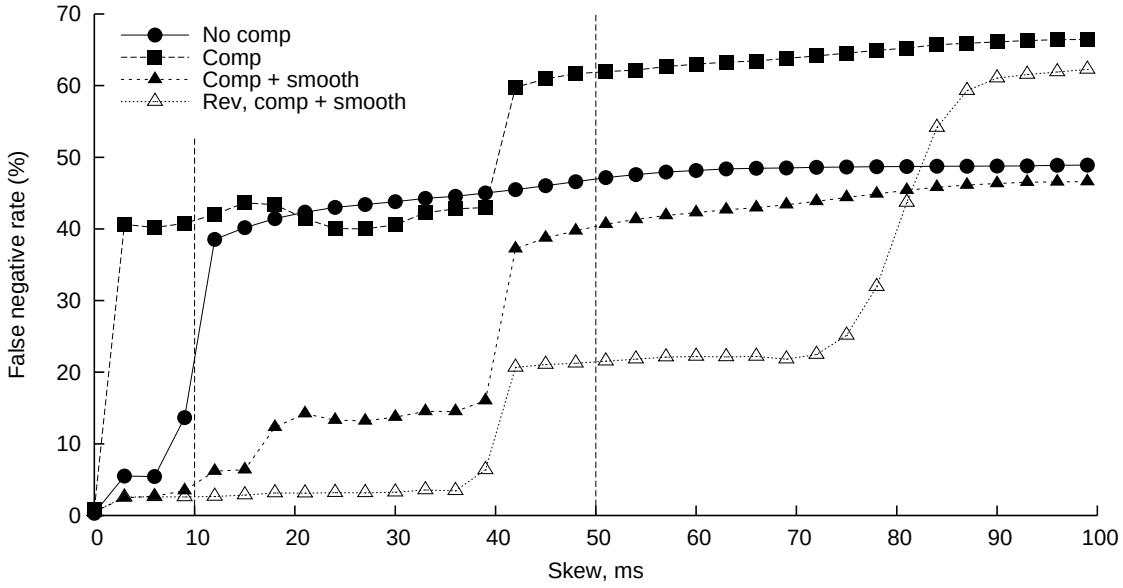


Figure 3.10: Effects of clock offset on nesting algorithm accuracy, clock-offset window = 30.

and the minimum of each curve indicates the optimal size for the offset window for that given amount of offset. As in Figure 3.10 the optimal size for the offset window is approximately equal to actual offset minus the size of the most prominent feature (once again 10 ms). Here, we can see that the penalty for underestimating offset is severe while the penalty for overestimating it is minor. It is best to err on the side of slightly larger offset windows.

Drop rate

Many traces have dropped packets, usually in bursts, if traffic is bursty or if the tracing mechanism gets interrupted. We used Maketrace as described in Section 3.3 to create traces with dropped packets. The effect of dropped packets on accuracy is shown in Figure 3.12. Note that the x axis is a dependent variable. We varied the simulated capture rate and plotted the resulting false-negative rate against the resulting drop rate. As Figure 3.12 shows, the nesting algorithm can tolerate small numbers of dropped packets, up to about 5% of the total.

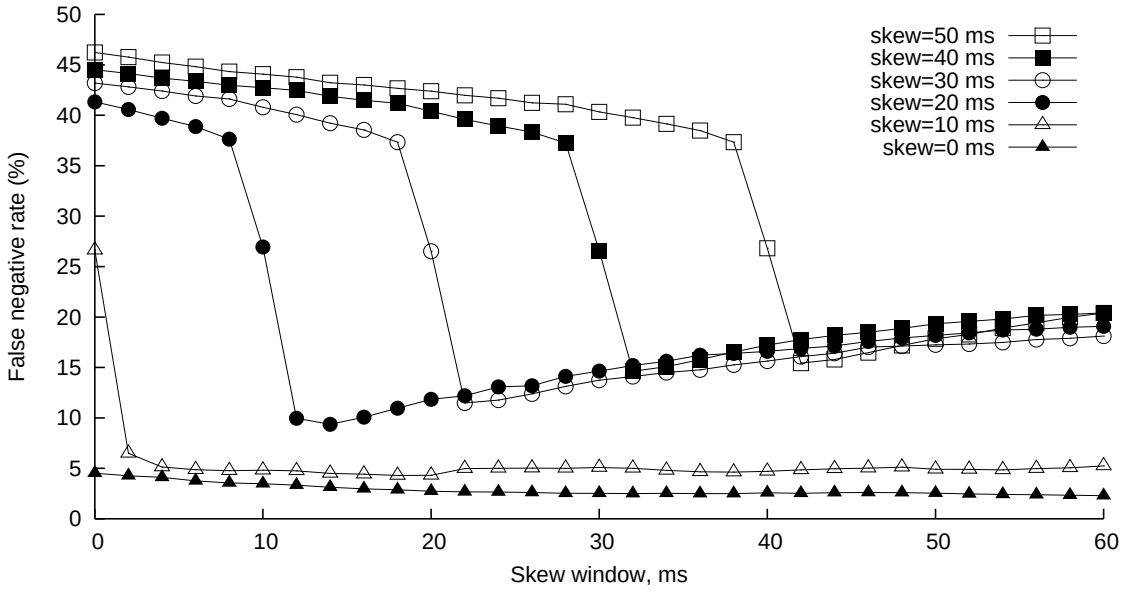


Figure 3.11: Varying clock-offset window sizes for several levels of offset.

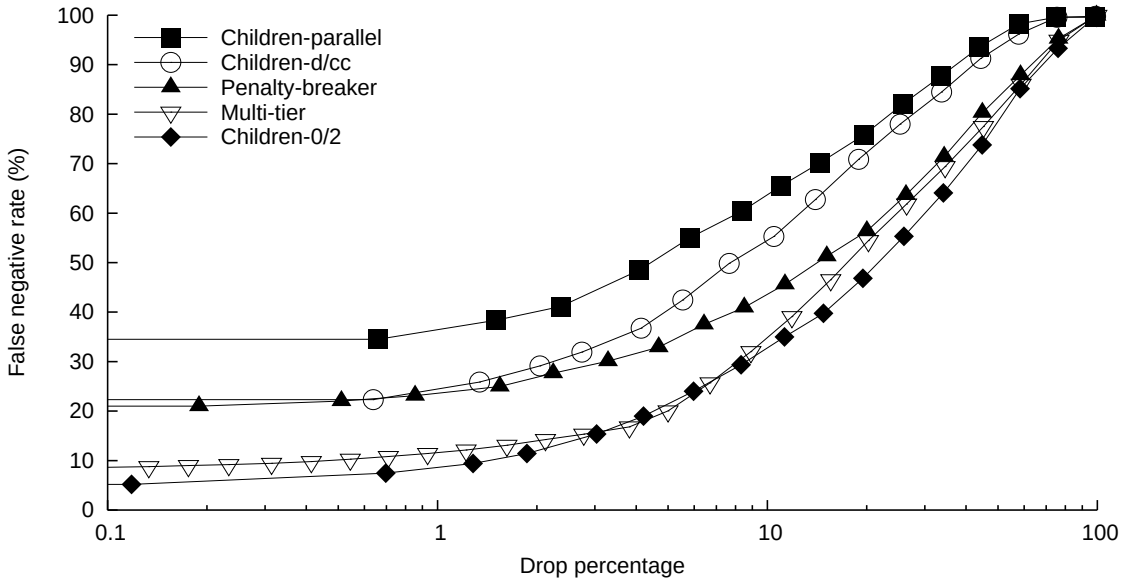


Figure 3.12: Effects of message drop rate on nesting algorithm accuracy.

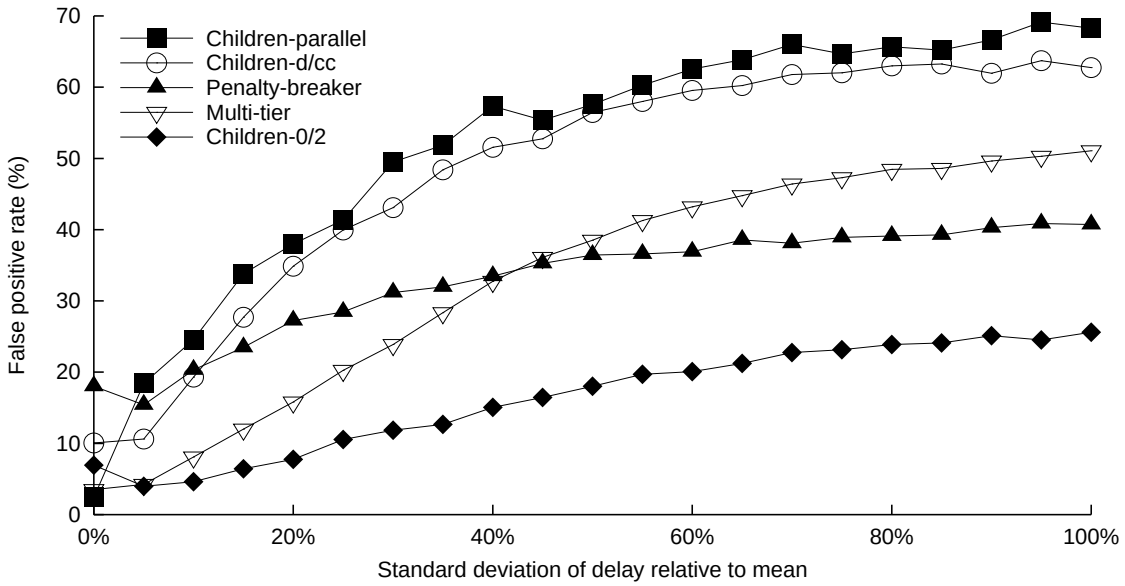


Figure 3.13: Effects of delay variation on nesting algorithm accuracy.

Delay standard deviation

Delays are often not deterministic. In Maketrace, we assume that delays have Gaussian distributions, as described in Section 3.3. In most tracelets, we use a standard deviation of 10% of the mean. We ran an experiment in which we tried standard deviations from 0% to 100% of the mean in order to measure the nesting algorithm's sensitivity to delay variance. As shown in Figure 3.13, delay variance has a substantial effect. The nesting algorithm performs quite well given deterministic delays, but its accuracy degrades quickly as the standard deviation increases to approximately 30% of the mean delay. Above 30% accuracy continues to degrade, but more slowly.

Note that this statement only holds for single-mode delay distributions. A bimodal distribution may have a large standard deviation, but nesting will handle it easily as long as the standard deviation within each mode is relatively small.

Parallelism

As described in Section 3.4.3, parallelism has at least a linear effect on the running time and storage requirements of the nesting algorithm. Its effect on accuracy is less clear. We ran an experiment in which we varied the level of parallelism for each trace configuration and measured the accuracy. The results are shown in Figure 3.14. Note that the x axis in Figure 3.14 is a dependent variable. We varied the parallelism scaling (the `-p` flag) in Maketrace and plotted the accuracy against the resulting number of messages per second. *Children-parallel* improves significantly with increases in parallelism, while *Children-d/cc* and *Penalty-breaker* both see decreased accuracy with increases in parallelism. However, all five trace configurations eventually see some improvement with increases in parallelism.

Our working hypothesis is that increased parallelism does not increase the likelihood of choosing the exact right parent, but it does increase the likelihood of choosing a incorrect parent that is close enough to the correct parent as to be indistinguishable in terms of path creation and delay averages. Higher parallelism means that possible parents are closer together, meaning that if a call-return pair picks the wrong parent it at least has many nearly-right parents to fall back on.

Noise

The nesting algorithm performs acceptably well in most cases with up to 15% noise. Figure 3.15 shows the effect of noise on accuracy. The x axis shows the percentage of noise added, which is the ratio of noise packets to non-noise packets as described in Section 3.3. Thus, the points at 15% represent a trace in which approximately 13% of the traffic is noise. Note that "noise" here refers only to traffic between hosts of interest that does not take the form of a causal path. Any noise between other hosts can be ignored, and any noise that takes the form of a causal path will be detected

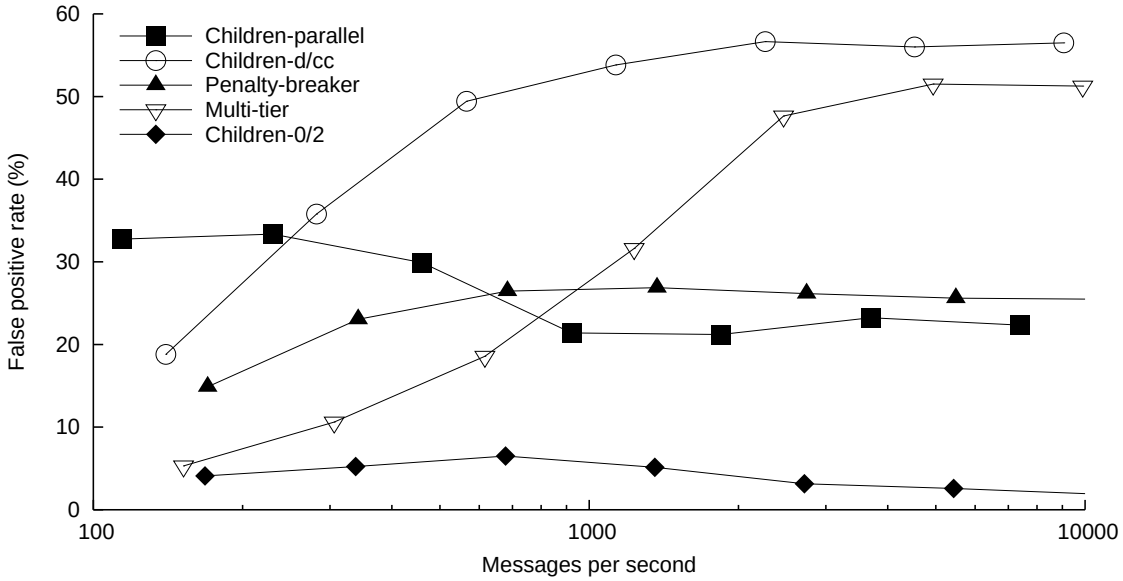


Figure 3.14: Effects of trace parallelism on nesting algorithm accuracy.

as such.

Noisy traces exhibit additional types of false positives that do not appear in other traces. For example, a long-lived call-return pair may get many other call-return pairs assigned to it as children. To prevent this, we used non-default penalty weights in the experiment that generated Figure 3.15. The weights we used were 1.5, 0.5, and 1.0 for the overlapping-child, same-child, and generic-child penalties, respectively.

3.5 Summary

Project 5 is an effective tool for characterizing the paths that requests take through black-box distributed systems and attributing the sources of delay encountered by each request. In this chapter, we presented two algorithms, nesting and convolution, for building causal paths from message traces. Further, we described an experimental framework and experimental results demonstrating the nesting algorithm's behavior on several real and synthetic traces. The experimental framework includes a trace generator and several real and synthetic traces chosen to explore the limits of the

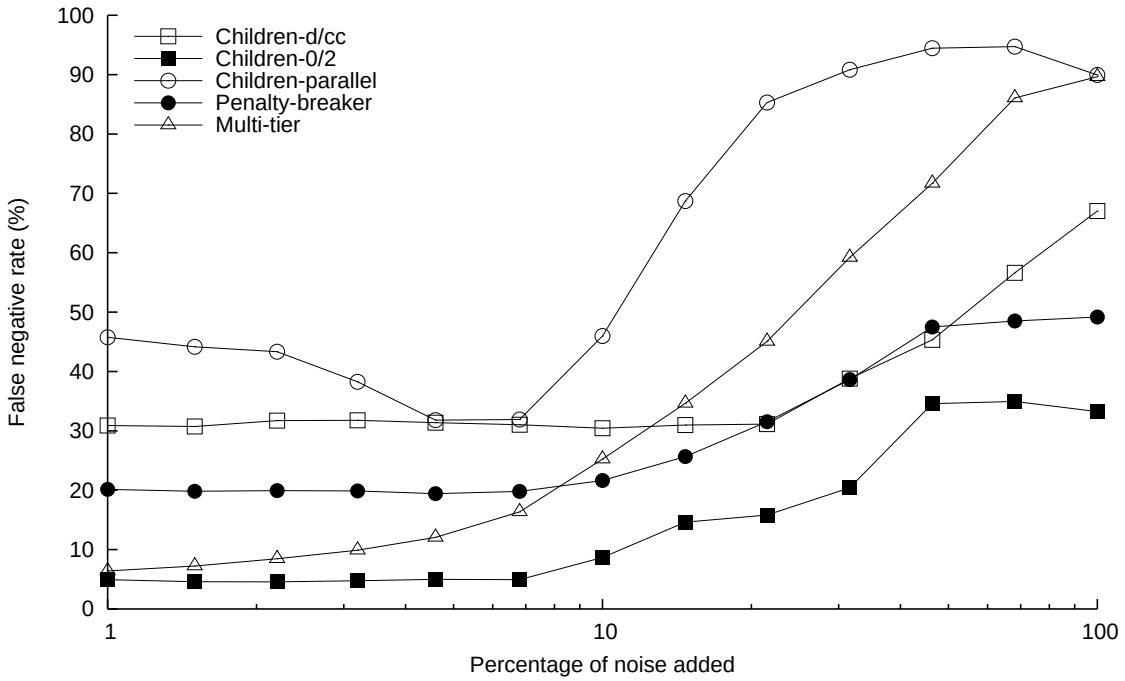


Figure 3.15: Effects of noise on nesting algorithm accuracy.

nesting algorithm. The experiments include free parameter exploration, the effect of trace size and composition on running time, and the effects of clock offset, drop rate, noise rate, delay variance, and parallelism on accuracy.

Chapter 4

WAP5: Wide-Area Black-Box Debugging

This chapter introduces the second of three debugging tools, Wide-Area Project 5 (WAP5). WAP5 extends the goals of Project 5 to wide-area distributed systems, where network latencies are larger, clocks can be unsynchronized, administrative privileges are often unavailable, and protocols in use can be more diverse. WAP5 introduces a new inference algorithm, an interposition library for tracing communication events, and an analysis of two content-distribution networks and a configuration and incident-monitoring system.

4.1 Overview

Wide-area distributed systems are difficult to build and deploy because not all debugging tools scale well across wide-area network links or across administrative domains. Compared to local-area distributed systems, wide-area distributed systems introduce new sources of delays and failures, including network latency, limited bandwidth, node unreliability, and asynchronous parallel programming. Furthermore, the sheer size of a wide-area system can make it daunting to find and debug underperforming nodes or to examine event traces. Often, programmers have trouble understanding the communications structure of a complex distributed system and have trouble isolating the specific sources of delays.

In this chapter, we describe the Wide-Area Project 5 (WAP5) system, a set of tools for capturing and analyzing traces of wide-area distributed applications. The WAP5 tools aid the development, optimization, and maintenance of wide-area distributed applications by revealing the causal structure and timing of communication

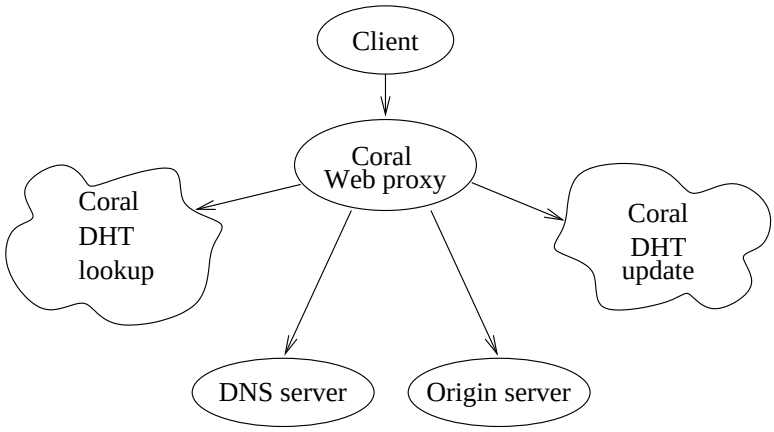


Figure 4.1: Example causal path through Coral.

in these systems. They highlight bottlenecks in both processing and communication. By mapping an application's communication structure, they highlight when an application's data flow follows an unexpected path. By discovering the timing at each step, they isolate processing or communication hotspots.

We are focusing on applications running on PlanetLab [5], perhaps the best collection of widely distributed applications for which research access is feasible. In particular, we have applied our tools to the CoDeeN [65] and Coral [22] content-distribution networks (CDNs). WAP5 constructs causal structures, such as the one shown for Coral in Figure 4.1, which matches a path described by Figure 1 in a paper on Coral [22].

Our tool chain consists of four steps, depicted in Figure 4.2. First, our dynamically linked interposition library captures one trace of socket-API calls per application process on each participating machine. Second, we reconcile the socket-API traces to form a single trace with one record per message containing both a sent and a received timestamp. These timestamps reflect the clocks at the sender and receiver machines, respectively, and are used to quantify and compensate for clock skew and to measure network latency. Third, we run our causality analysis algorithm on the reconciled trace to find causal paths through the application, like the one in Figure 4.1. Finally,

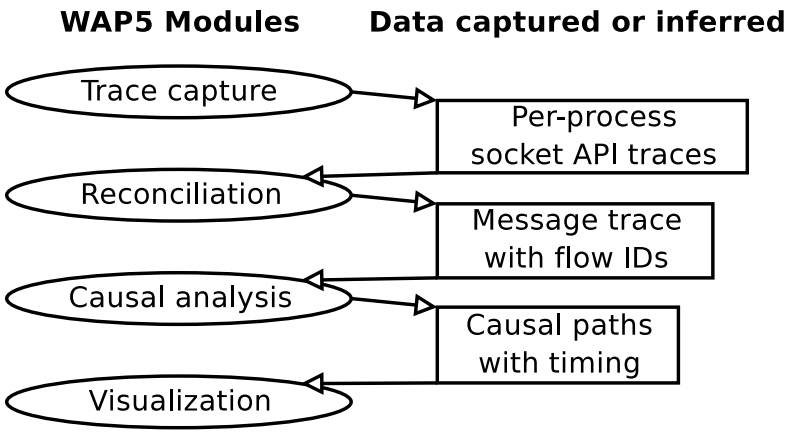


Figure 4.2: Schematic of the WAP5 tool chain.

we render the causal paths as trees or timelines.

In the space of tools that analyze application behavior for performance debugging, our approach is among the least invasive and works on the largest scale of systems: wide-area distributed systems. Other causal-path analysis tools differ in their invasiveness or in the scale of systems they target. Project 5 (see Chapter 3), targets heterogeneous local-area distributed systems and is minimally invasive because it works using only network traces. Pip (see Chapter 5) supports wide-area systems but requires manual annotations, an instrumented platform, or both.

WAP5 makes the following contributions:

- A new causal-path inference algorithm, the *message-linking algorithm*, that introduces support for wide-area systems. Some of the new features of linking also make it easier to analyze local-area systems. We provide a full comparison with our previous algorithms [1] in Section 4.5.3.

- A discussion of several previously unaddressed problems with causal path analysis, including naming issues, DHT issues, wide-area network latencies, clock skew, and network address translation.

- Results from applying our tools to three real systems, the CoDeeN and Coral

CDNs running on PlanetLab, and Slurpee, an enterprise-scale incident-monitoring system.

In the next section, we define the problem we are solving more explicitly. We then describe the three main tools in our approach, which perform trace capture, trace reconciliation, and causal path analysis over the trace. Finally, we present results from three systems.

4.2 Problem definition

In this section, we define the problem we are solving. We include a description of the target applications, discussion of some distributed hash table (DHT) issues, definitions of our terminology for communication between components, our model of causality, and several issues related to naming of components.

4.2.1 Target applications

Our primary goal is to expose the causal structure of communication within a distributed application and to quantify both processing delays inside nodes and communication (network) delays. In this chapter, we specifically focus on wide-area distributed systems (and other systems) where the network delays are non-negligible. We further focus on PlanetLab applications because we can get access to them easily. However, nothing in our approach requires the use of PlanetLab.

We aim to use as little application-specific knowledge as possible and not to change the application. We can handle applications whose source code is unavailable, whose application-level message formats are unknown, and, in general, without *a priori* information about the design of the application.

Our tools can handle distributed systems whose "nodes" span a range of granularities ranging from entire computers down to single threads, and whose communica-

tion paths include various network protocols and intra-host IPC. We aim to support systems that span multiple implementation frameworks; for example, a multi-tier application where one tier is J2EE, another is .Net, and a third is neither.

We currently assume the use of unicast communications and we assume that communication within an application takes the form of messages. It might be possible to extend this work to analyze multicast communications.

4.2.2 DHT issues

Several interesting distributed applications are based on DHTs. Therefore, we developed some techniques specifically for handling DHT-based applications.

DHTs perform lookups either iteratively, recursively, or recursively with a shortcut response [15]. In an iterative lookup, the node performing the query contacts several remote hosts (normally $O(\lg n)$ for systems with n total hosts) sequentially, and each provides a referral to the next. In a recursive lookup, the node performing the query contacts one host, which contacts a second on its behalf, and so on. A recursive lookup may return back through each intermediary, or it may return directly along a shortcut from the destination node back to the client.

With an iterative DHT, all of the necessary messages for analysis of causal paths starting at a particular node can be captured at that node. With a recursive or recursive-shortcut DHT, causal path analysis requires packet sniffing or instrumentation at every DHT node. The algorithm presented in this chapter handles all three kinds of DHT.

DHTs create an additional "aggregation" problem that we defer until Section 4.2.5.

4.2.3 Communications terminology

Networked communication design typically follows a layered architecture, in which the protocol data units (PDUs) at one layer might be composed of multiple, partial, or overlapping PDUs at a lower layer. Sometimes the layer for meaningfully expressing an application's causal structure is higher than the layer at which we can obtain traces. For example, in order to send a 20 KB HTTP-level response message, a Web server might break it into write() system call invocations based on an 8 KB buffer. The network stack then breaks these further into 1460-byte TCP segments, which normally map directly onto IP packets, but which might be fragmented by an intervening router.

We have found it necessary to clearly distinguish between messages at different layers. We use the term *packet* to refer to an IP or UDP datagram or a TCP segment. We use *message* to refer to data sent by a single write() system call or received by a single read(). We refer to a large application-layer transfer that spans multiple *messages* as a *fat message*. Fat messages require special handling: we combine adjacent messages in a flow into a single large message before beginning causal analysis. Conversely, several sufficiently small application-layer units may be packed into a single system call or network packet, in protocols that allow pipelining. In the systems analyzed here, such pipelining does not occur. In systems where pipelining is present, our tool chain would see fewer requests than were really sent but would still find causality.

4.2.4 Causality model

We consider message A→B to have caused message B→C if message A→B is received by node B, message B→C is sent by node B, and the logic in node B is such that the transmission of B→C depends on receiving A→B. In our current work, we

assume that every message B→C is either caused by one incoming message A→B or is spontaneously generated by node B. This assumption includes the case where message A→B causes the generation of several messages, B→X, B→Y, B→Z, etc.

An application where one message depends on the arrival of many messages (e.g., a barrier) does not fit this model of causality. WAP5 would attribute the outgoing message to only one—probably the final—incoming message. Additionally, if structure or timing of a causal path pattern depends on application data inside a message, WAP5 will view each variation as a distinct path pattern and will not detect any correlation between the inferred path instances and the contents of messages.

We cannot currently handle causality that involves asynchronous timers as triggers; asynchronous events appear to be spontaneous rather than related to earlier events. This restriction has not been a problem for the applications we analyzed for this chapter. We also cannot detect that a node is delayed because it is waiting for another node to release a lock.

4.2.5 Naming issues

Our trace-based approach to analyzing distributed systems exposes the need for multiple layers of naming and for various name translations. Clear definitions of the meanings of various names simplify the design and explanation of our algorithms and results. They also help us define how to convert between or to match various names.

Causal path analysis involves two categories of named objects, computational nodes and communication flow endpoints. Nodes might be named using hostnames, process IDs, or at finer grains. Endpoints might be named using IP addresses, perhaps in conjunction with TCP or UDP ports, or UNIX-domain socket pathnames. These multiple names lead to several inter-related challenges:

- **Which level of name to use**: While we want our tools to avoid incorporating

application-specific knowledge, their *use* may require some knowledge of the application. In particular, the user of our tools may have to decide whether to treat a host as a single node or as a collection of process-level nodes. The process-level view might add useful detail if each process has a distinct role, or it might just add confusion if processes on a host are interchangeable, as in servers built using a process pool.

The selection of naming granularity interacts with the choice of tracing technology. Packet sniffing, the least invasive tracing approach, makes it difficult or impossible to identify processes rather than hosts. Use of an interposition library, such as the one we describe in Section 4.3, allows process-level tracing.

- **How to match node names and endpoint names**: A host might include several process nodes and multiple communication endpoints. For example, a Web proxy process could accept HTTP requests on port 8090 and send forwarded requests using a series of ephemeral port numbers; in this case, all these connections belong to one process. However, the same host might run both a Web server and an FTP server, in which case the two different server ports correspond to distinct processes. Using an interposition library, we capture enough information to match endpoints to processes; it is much harder using packet-sniffing.

- **How to find both ends of a path**: Whenever possible, we capture trace records at each host in a distributed system. Thus, each message within the system generates two trace records: one at the sender and one at the receiver. In order to get both sender and receiver timestamps for a message, we need to match up the two trace records – in effect, finding a common name for each message. This task is usually straightforward but can be complicated by multihomed hosts, reordered or lost datagrams, or clock offset.

The distinction between node names and endpoint names allows the analysis of a single distributed application using multiple traces obtained with several different techniques. For example, we could trace UNIX-domain socket messages using the interposition library and simultaneously trace network messages using a packet sniffer.

Table 4.1 shows the various names (the columns) captured by our interposition library, and where they are used in our analysis. We include this table to illustrate the complexity of name resolution; it may be helpful to refer to it when reading Sections 4.4 and 4.5.

The table columns are as follows:

- **IP addr, port.** The addresses for the source and destination of a network connection or of a connectionless message.
- **Socket path.** The file-system location of a Unix-domain socket.
- **File descriptor.** The system-level handle for an open socket. It is normally bound to a source and destination endpoints, or to a Unix-domain socket path.
- **Length.** The length in bytes of a message.
- **Hostname.** The host on which part of a distributed system runs.
- **PID.** The process identifier. The combination of hostname and PID identifies a single process, where an incoming message may cause an outgoing message.
- **Peer PID.** The process identifier of the remote endpoint of a Unix-domain socket. The peer PID is used to differentiate among several connections active through a single socket in the file system.

	Socket API parameters				Other captured information				
	both ends: (IP addr,port)	socket path	file descriptor	length	hostname	PID	peer PID	checksum	timestamp
trace file header					C	C			C
new connection: TCP	C		C						C
new connection: UDP or raw IP	C		C						C
new connection: UNIX domain		C	C				C		C
message: TCP or UNIX domain			C	C					C
message: UDP or raw IP	C		C	C				C	C
reconciliation: matching send & rcv recs	U	U	U	U			U	U	
causal analysis (message linking)					U	U			U
aggregating nodes	U				U				

Table 4.1: Where different naming information is captured (C) or used (U).

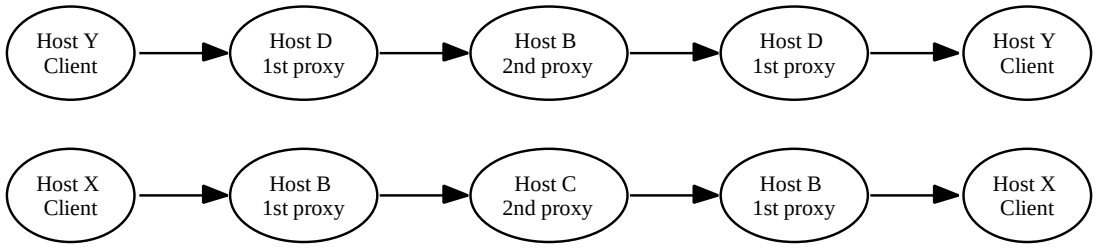
- **Checksum.** A 16-bit summary of the contents of a message. The checksum, in combination with message size, is used to reconcile the send and receive events for UDP messages.

- **Timestamp.** A 64-bit counter of the second and microsecond that an event occurred.

Aggregation across multiple names

Causal path analysis aggregates similar path instances into path patterns, presenting to the user a count of the instances inferred along with average timing information. The simplest form of aggregation is combining path instances with identical structure, i.e., those that involve exactly the same nodes in exactly the same order. More advanced aggregation techniques look for isomorphic path instances that perform the same tasks via different nodes, perhaps for load balancing. Without aggregation, it is difficult to visualize how a system performs overall: where the application designer thinks of an abstract series of steps through an application, causal path analysis finds a combinatorial explosion of rare paths going through specific nodes. Aggregation is particularly important for DHTs, which are highly symmetric and use an intentionally wide variety of paths for reliability and load balancing. Aggregating paths is useful for finding performance bugs that are due to a design or coding error common to all hosts.

Once causal path analysis has identified a set of isomorphic paths, it is possible to aggregate the results based on the *role* of a node rather than its *name*. For example, Coral and CoDeeN have thousands of clients making requests; trees starting at one client would not normally be aggregated with trees starting at another. As another example, Figure 4.3(a) shows two causal paths with the same shape but different hostnames. In the top path, Host D fills the role of the first-hop proxy, and Host



(a) unaggregated paths

(b) corresponding aggregated path

Figure 4.3: Example of aggregation across multiple names.

B fills the role of the second-hop proxy; in the other path, Host B is the first-hop proxy and and Host C is the second-hop proxy. What we might like to see instead is the aggregated path in Figure 4.3(b), which aggregates the clients, first-hop, and second-hop proxies. Of course, the unaggregated paths should still be available, in case a performance problem afflicts specific nodes rather than a specific task.

Currently, our code aggregates clients, but we have not yet implemented aggregation across servers. To aggregate clients, we designate each TCP or UDP port as *fixed* or *ephemeral* and each node as a client or a server. A port is fixed if it communicates with many other ports. For example, a node making a DNS request will allocate a source port dynamically (normally either sequentially or randomly), but the destination port will always be 53. Thus, causal path analysis discovers that 53 is a fixed port because it talks to hundreds or thousands of other ports. A node is considered a server if it uses fixed ports at least once, and a client otherwise. Our algorithm replaces all client node names with a single string "CLIENT" and replaces all ephemeral port numbers with an asterisk before building and aggregating trees.

Thus, otherwise identical trees beginning at different clients with different ephemeral source-port numbers can be aggregated.

4.3 Trace collection infrastructure

We now describe how we capture traces of inter-node communication. We wrote an interposition library, LibSockCap, to capture network and inter-process communication. LibSockCap captures mostly the same information as *strace -e network* (i.e., a trace of all networking system calls), plus additional needed information, with much lower overhead. The extra information is needed for reconciliation and includes fingerprints of UDP message contents, the PID of peers connecting through a Unix socket, the peer name even when *accept* does not ask for it, the local name bound when *connect* is called, and the number assigned to a dynamic listening port not specified with *bind*. Further, LibSockCap imposes less than $2\mu s$ of overhead per captured system call, while strace imposes up to $60\mu s$ of overhead per system call. Finally, LibSockCap generates traces about an order of magnitude smaller than strace.

LibSockCap traces dynamically linked applications on any platform that supports library interposition via `LD_PRELOAD`. LibSockCap interposes on the C library's system call wrappers to log all socket-API activity, for one or more processes, on all network ports and also on UNIX-domain sockets. For each call, LibSockCap records a timestamp and all parameters (as shown in Table 4.1), but not the message contents. In addition, LibSockCap monitors calls to *fork* so that it can maintain a separate log for each process. On datagram sockets, it also records a message checksum so that dropped, duplicated, and reordered packets can be detected.

There are several advantages to capturing network traffic through library interposition rather than through packet sniffing, either on each host or on each network segment.

- **Logical message semantics**: messages are captured with the same order and boundaries that the application sees, rather than after the network potentially fragments or combines them.

- **Finer granularity**: LibSockCap attributes communication to individual processes rather than to whole hosts. Also, LibSockCap can capture UNIX-domain sockets, while sniffing cannot.

- **Efficiency**: LibSockCap adds less overhead than running a sniffer on the same host, as is necessary on PlanetLab, because it runs in the memory space of the processes being traced and so does not require context switches or buffer copies to record messages.

Interposition does have disadvantages relative to sniffing.

- **No control packets**: only sniffing can capture network control messages. However, our work focuses on the causal relationships between logical messages, not control messages.

- **Lack of packet boundaries, fragments, and retransmissions**: problems arising in the network stack or in the network, such as excessive fragmentation or retransmission, are not visible to our interposition library.

- **Timestamps added by user process**: any delays introduced by the network stack happen after LibSockCap timestamps the event and get attributed to network delay.

The advantages are significant enough that even in environments where sniffing is feasible, we prefer to use LibSockCap.

4.3.1 Runtime overhead

To verify that LibSockCap imposes negligible overhead on the applications being traced, we ran Seda's HttpLoad [66] using Java 1.4.1_01 against Apache 1.3.1. Both the client and the server were dual 2.4 GHz Pentium 4 Xeon systems running Linux 2.4.25. LibSockCap had no measurable effect on throughput or on average, 90th-percentile, or maximum request latency for any level of offered load. However, the server CPU was not saturated during this benchmark, so LibSockCap might have more impact effect on a CPU-bound task.

We measured the absolute overhead of LibSockCap by comparing the time to make read/write system calls with and without interposition active, using the server described above. LibSockCap adds about $0.02\mu s$ of overhead to file reads and writes, which generate no log entries; $1.03\mu s$ to TCP reads; $1.02\mu s$ to TCP writes; and $0.75\mu s$ to UDP writes. In our benchmark, Apache made at most 3,019 system calls per second, equal to an overhead of about 0.3% of one CPU's total cycles.

4.3.2 Deployment experience

To capture the traces used for our experiments, we sent LibSockCap sources to the authors of Coral and CoDeeN. Both reported back that they used their existing deployment mechanisms to install LibSockCap on all of their PlanetLab nodes. After the processes ran and collected traces for a few hours, they removed LibSockCap, retrieved the traces to a single node, and sent them to us.

While LibSockCap is more invasive than packet sniffing (in that it requires additional software on each node), packet sniffing is more logistically challenging in practice, as we discuss in Section 4.7.

4.4 Trace reconciliation

The trace reconciliation algorithm converts a set of per-process traces of socket activity (both network and inter-process messages) to a single, more abstract, trace of inter-node messages. This algorithm includes translation from socket events to flow-endpoint names and node names. The output is a trace containing logical message tuples of the form (sender-timestamp, sender-endpoint, sender-node, receiver-timestamp, receiver-endpoint, receiver-node).

Name translation: In the LibSockCap traces, each *send* or *recv* event contains a timestamp, a size, and a file descriptor. Reconciliation converts each file descriptor to a flow-endpoint name. With UNIX domain and TCP sockets, we can easily find the flow-endpoint name (i.e., UNIX path or <IP address, port> pair) in prior *connect*, *accept*, or *bind* API events in the trace. File descriptors for datagram (UDP or raw IP) sockets, however, may or may not be bound to set source and destination addresses. If not, the remote address is available from the *sendto* or *recvfrom* parameter and we use the host's public IP address or the loopback address as the local address.

Timestamps: We include the timestamps from both the sender and receiver traces in the final trace. With both timestamps, it is simple to obtain the network latency of each message, as we describe in Section 4.5.2. Whenever we have only one timestamp for a message (because we only sniffed one endpoint), we use *nil* for the other timestamp.

4.5 The message linking algorithm

Causal path analysis looks for causal relationships in the logical messages produced by trace reconciliation. Its output is a collection of path patterns, each annotated with one or more scores indicating importance. *Message linking*, or *linking* for short, is a

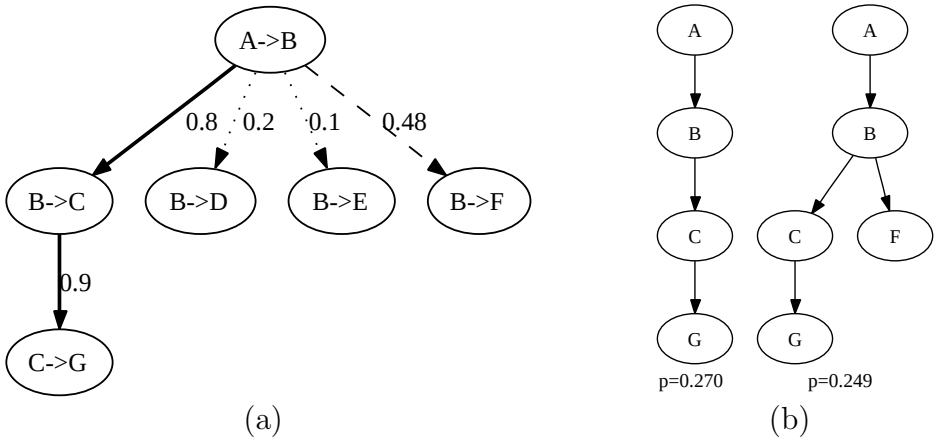


Figure 4.4: Sample link probability tree and the two causal path instances it generates. Solid, dotted, and dashed arrows indicate "probably-true," "probably-false," and "try-both" links, respectively.

new causality analysis algorithm for distributed communication traces. Linking works with both local-area and wide-area traces, which may be captured using LibSockCap, a sniffer, or other methods. We compare linking with other causal path analysis algorithms at the end of this section.

4.5.1 Algorithm description

For each message in the trace, linking attempts to determine if the message is spontaneous or is caused by another message. In many cases, the cause is ambiguous, in which case linking assigns a probability for the *link* between the *parent* message into the node and the *child* message out of the node. The probabilities of the links for all parent messages to a given child message sum to one.

Linking then constructs path instances from these links and assigns each path instance a confidence score that is the product of all of the link probabilities in the tree. The total score across all instances of a given path pattern represents the algorithm's estimate of the number of times the pattern appeared in the trace. If a link is sufficiently ambiguous (e.g., if it has a probability near 0.5), two path instances will be built, one with the link and one without it. Figure 4.4 shows an example link

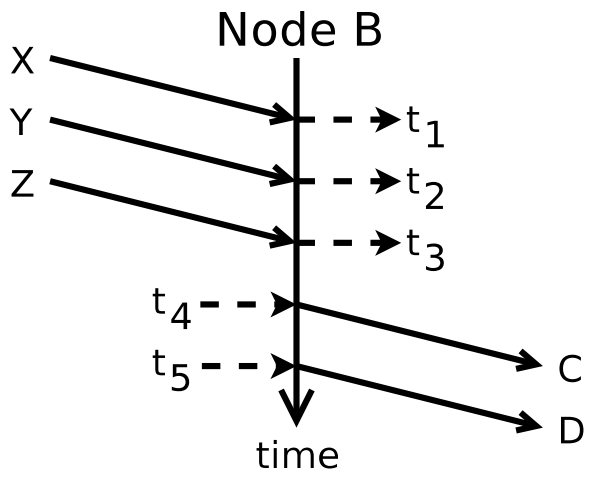


Figure 4.5: Three calls into B that might have caused B→C.

probability tree and the causal path instances it generates. The tree on the left shows all of the messages that might have been caused, directly or indirectly, by one specific A→B message, with a probability assigned to each possible link. From this tree, the linking algorithm generates the two causal path instances on the right, each with a probability based on the decisions made to form it. Here, two path instances are generated because the link between A→B and B→F has probability close to $p = 0.5$.

In broad terms, the linking algorithm consists of three steps: (1) estimating the average causal delay for each node, (2) determining possible parents for each message, and (3) building path instances and then aggregating them into path patterns. We describe the algorithm in more detail in the sections that follow.

Step 1: Estimating the average causal delay

The probability of each link between a parent message into B and a child message out of B is a function of how well it fits the causal delay distribution. Causal delays represent the service times at each node. Therefore, as is common in system modeling, we fit them to an exponential distribution $f(t) = \lambda e^{-\lambda t}$[62], where λ is a scaling parameter to be found. Figure 4.6 shows a sample exponential distribu-

tion. An exponential distribution exactly models systems in which service times are memoryless—that is, the probability that a task will complete in the next unit time is independent of how long the task has been running. However, not all systems have memoryless service times. Even in systems with other service time distributions, the exponential distribution retains a useful property: because it is a monotonically decreasing function, the linking algorithm will assign the highest probability to causal relationships between messages close to each other in time. Thus, the exponential distribution works well even if its scaling factor is incorrect or the system does not exhibit strictly memoryless service times.

We also considered $f(t) = \lambda t e^{-\lambda t}$, a gamma distribution in which $\alpha = 2$. This gamma distribution assigns the highest probabilities to delays near $1/\lambda$, which causes the linking algorithm to produce more accurate results if λ is estimated correctly, but much worse results otherwise.

We use an independent exponential distribution for each B→C pair, by estimating the average delay $d_{B\to C}$ that B waits before sending a message to C. The delay distribution scaling factor $\lambda_{B\to C}$ is equal to $1/d_{B\to C}$.

Correctly determining $d_{B\to C}$ requires accurate knowledge of which message caused which; thus, linking only approximates $d_{B\to C}$ and hence $\lambda_{B\to C}$. Linking estimates $d_{B\to C}$ as the average of the smallest delay preceding each message. That is, for each message B→C, it finds the latest message into B that preceded it and includes that delay in the average. If there is no preceding message within x seconds, B→C is assumed to be a spontaneous message and no delay is included. The value of x should be longer than the longest real delay in the trace. We use $x = 2$ sec for the Coral and CoDeeN traces, but $x = 100\ ms$ for the Slurpee trace. The value of x is user-specified, depends only on expected processing times, and does not need to be a tight bound.

In the presence of high parallelism, the estimate for each d may be too low, because the true parent message may not be the most recent one. However, because the exponential distribution is monotonically decreasing, the ranking of possible parents for a message is preserved even when d and λ are wrong. It is possible to iterate over steps (1) and (2) to improve the estimate of λ, but linking does not currently do so.

Step 2: Finding and scoring parent messages

After estimating $\lambda_{B\rightarrow C}$ for each communicating pair of nodes B→C, the linking algorithm assigns each causal link a weight based on its delay. The weight of the link between X→B and B→C in the example in Figure 4.5 is set to

$$f(t_4 - t_1) = e^{-\lambda_{B\rightarrow C}(t_4 - t_1)},$$

where $(t_4 - t_1)$ is the delay between the arrival of X→ B and the departure B→C. Additionally, B→C may not have been caused by any earlier message into B, and instead might have been spontaneous. This possibility is given a weight equal to a link with delay $y \cdot d_{B\rightarrow C}$. y should be a small constant; we use $y = 4$. A larger y instructs the algorithm to prefer longer paths, while a smaller y generates many short paths that may be suffixes of correct paths. Spontaneous action is the most likely choice only when there are no messages into B within the last $y \cdot d_{B\rightarrow C}$ time. Figure 4.6 shows the weights assigned to all three possible parents of B→ C, as well as the weight assigned to the possibility that it occurred spontaneously.

Once all of the possible parents for this B→C message have been enumerated, the weights of their links are normalized to sum to 1. These normalized weights become the probability for each link. Figure 4.7 shows the possible parents for the B→C and B→D calls, with their assigned probabilities.

Hosts or processes that were not traced result in *nil* timestamps in the reconciled trace. That is, if node A was traced and B was not, then A→B messages will

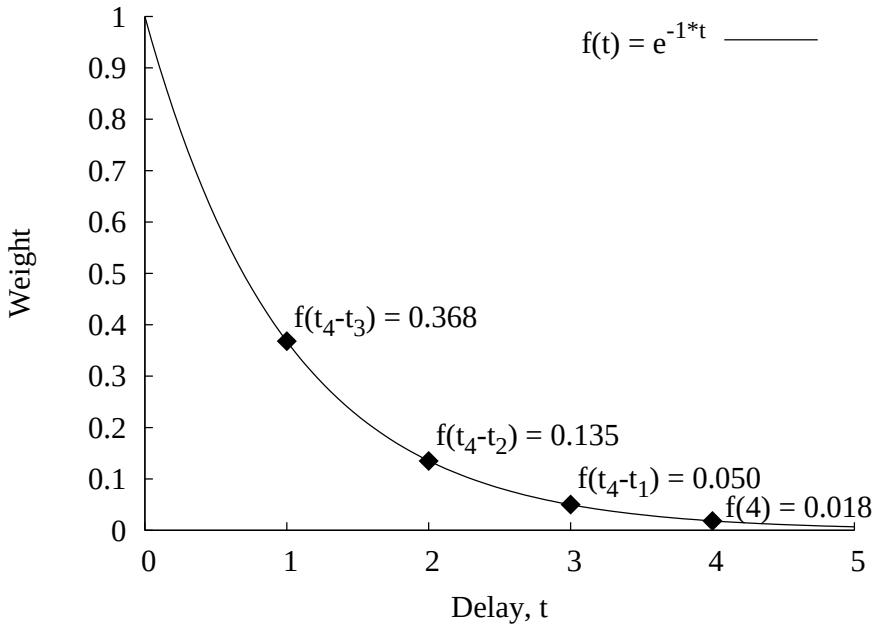


Figure 4.6: An exponential distribution with $\lambda = 1$, showing the weights assigned to all possible parents of B→C. $f(4)$ represents the possibility of a spontaneous message, given $y = 4$.

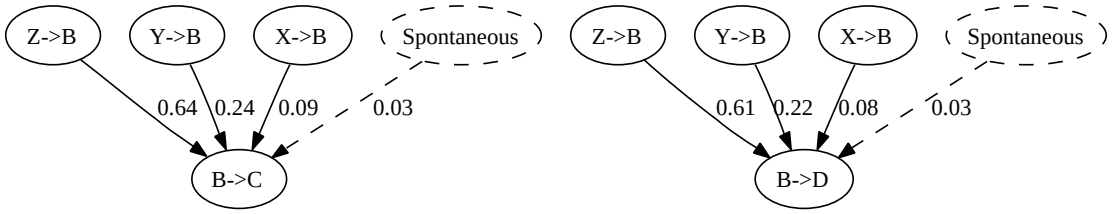


Figure 4.7: Possible-parent trees for the messages in Figure 4.5.

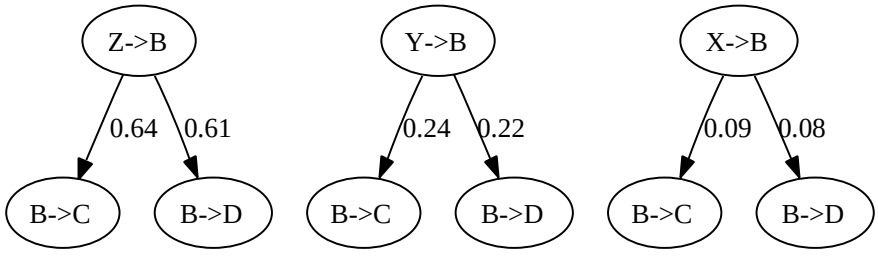


Figure 4.8: Possible-child trees formed from the trees in Figure 4.7.

be present with send timestamps but not receive timestamps, and B→A messages will have only receive timestamps. Both estimating $d_{B \to A}$ and assigning possible parents to B→A rely on having timestamps at both nodes. In this case, we use A's timestamps in place of B's and only allow causality back to the same node: A→B→A. This assumption allows calls from or to nodes outside the traced part of the system but avoids false causality between, e.g., several unrelated calls to the same server.

After enumerating and weighting all possible parents for each message, the linking algorithm uses these links to generate a list of the possible children for each message, preserving the link probabilities. This inversion, shown in Figure 4.8, is necessary because causal path instances are built from the root down.

Step 3: Building trees

The final step of the linking algorithm builds path instances from the individual links, then aggregates them into path patterns. That is, if step (2) finds the relationships shown in Figure 4.4(a), it would generate the two causal path instances shown in Figure 4.4(b), with the following probabilities:

$$\begin{aligned} p_1 &= 0.8 \cdot 0.9 \cdot (1 - 0.2) \cdot (1 - 0.1) \cdot (1 - 0.48) \\ &\approx 0.270 \\ p_2 &= 0.8 \cdot 0.9 \cdot (1 - 0.2) \cdot (1 - 0.1) \cdot 0.48 \\ &\approx 0.249 \end{aligned}$$

Each causal link included contributes a factor p corresponding to its probability. Each causal link omitted contributes a $1 - p$ factor.

For each link in the tree (e.g., did A→B cause B→ C?), step (3) treats it as *probably-false*, *probably-true*, or *try-both*, based on its probability. Decisions are designated try-both if their probability is close to 0.5 or if they represent one of the

most likely causes for a given message. That is, in Figure 4.4, if A→B is the most likely cause of B→D, then the A→B→D link will be made a try-both even though its probability is not near 0.5, ensuring that at least one cause of B→D is considered even if each possible cause has probability $p < 0.5$. The number of path instances generated from a given root message is $O(2^k)$, where k is the number of ambiguous links from that message or its descendants that are treated as try-both. Therefore, k must be limited to bound the running time of linking.

Linking assigns a probability to each tree equal to the product of the probabilities of the individual decisions—using $(1 - p)$ for decisions to omit a causal link—made while constructing it. If a specific path pattern is seen several times, we keep track of the total score (i.e., the expected number of times the pattern was seen) and the maximum probability. Path patterns in the output are generally ordered by total score.

Big trees will have low scores because more decisions (more uncertainty) go into creating them. This behavior is expected.

4.5.2 Node and network latency

The latency at each node B is the time between the receive timestamp of the parent message arriving at a node B and the send timestamp of the child message that node B sends. Since both timestamps are local to B, clock offset and clock skew do not affect node latency. For aggregated trees, the linking algorithm calculates the average of that node's delays at each instance of the tree, weighted by the probability of each instance. In addition to the average, we optionally generate a histogram of delays for each node in the tree.

The network latency of each message is the difference between its send and receive timestamps. These timestamps are relative to different clocks (they come from

LibSockCap logs at different hosts), so the resulting latency includes clock offset and skew unless we estimate it and subtract it out. We use a filter on the output of the linking algorithm to approximate pairwise clock offset by assuming symmetric network delays, following Paxson's technique [50]. For simplicity, we ignore the effects of clock skew. As a result, our results hide clock offset and exhibit symmetric average delays between pairs of hosts.

4.5.3 Algorithm comparison

Project 5 presented two causal-path analysis algorithms, *nesting* and *convolution*. The nesting algorithm works only on applications using call-return communication and can detect infrequent causal paths (albeit with some inaccuracy as their frequency drops). However, messages must be designated as either calls or returns and paired before running the nesting algorithm. If call-return information is not inherently part of the trace, as in the systems analyzed here, then trying to guess it is error-prone and is a major source of inaccuracy. Linking and nesting both try to infer the cause, if any, for each message or call-return pair in the trace individually.

The nesting algorithm only uses one timestamp per message. It is therefore forced either to ignore clock offset or to use fuzzy timestamp comparisons, which only work when all clocks differ by less time than the delays being measured. Since clocks are often unsynchronized—PlanetLab clocks sometimes differ by minutes or hours—our approach of using both send and receive timestamps works better for wide-area traces.

The convolution algorithm uses techniques from signal processing, matching similar timing signals for the messages coming into a node and the messages leaving the same node. Convolution works with any style of message communication, but it requires traces with a minimum of hundreds of messages, runs much more slowly than

Trace	Date	Number of messages	Trace duration	Number of hosts	Number of processes
CoDeeN	Sept. 3, 2004	4,702,865	1 hour	115	230
Coral	Sept. 6, 2004	4,246,882	1 hour	68	168

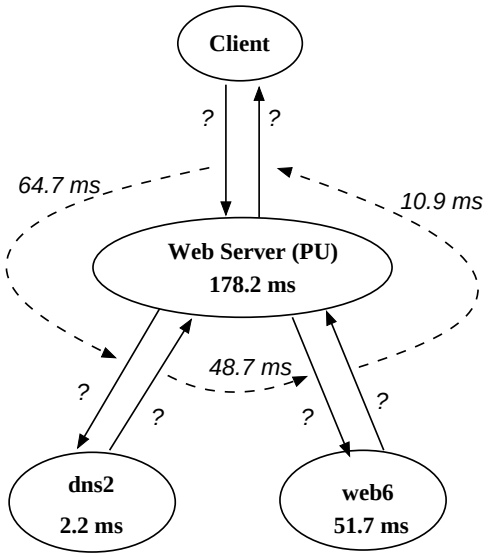
Table 4.2: Trace statistics.

nesting or linking, and is inherently unable to detect rare paths. When we applied convolution to Coral, it could not detect rare paths like DHT calls and could not separate node processing times of interest from network delays and clock offset.

The linking and nesting algorithms both have $O(n \lg n)$ running time, determined by the need to sort messages in the trace by timestamp, but both are dominated by an $O(n)$ component for the traces we have tried. The convolution algorithm requires $O((t/s) \lg(t/s))$ running time, where t is the duration of the trace and s is the size of the shortest delays of interest. In practice, convolution usually takes one to four hours to run, nesting rarely takes more than twenty seconds, and linking takes five to ten minutes but can take much less or much more given a non-default number of try-both decisions allowed per causal-link tree. Linking and nesting both require $O(n)$ memory because they load the entire trace into memory, while convolution requires an amount of memory proportional to the output size, often under 1 MB. Chapter 3 has more details for nesting and convolution, while Section 4.6 has more details for linking.

4.6 Results for PlanetLab applications

In this section, we present some results from our analyses of traces from the CoDeeN and Coral PlanetLab applications. Table 4.2 presents some overall statistics for these two traces.



Solid arrows show message paths, labeled one-way delays, when known. Dotted arcs show node-internal delays between message events. Nodes are labeled with name and total delay for node and its children.

Figure 4.9: Call-tree visualization.

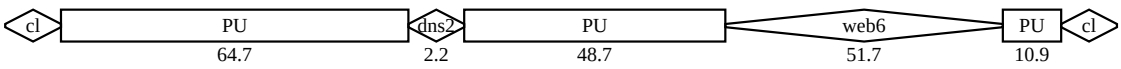


Figure 4.10: Linking algorithm output for a Coral miss path with DNS lookup, delays in ms.

4.6.1 Visualization of results

For a given causal path pattern, we use a timeline to represent both causality and time; for example, see Figures 4.10, 4.12, 4.13, and 4.14. Boxes represent nodes and lines represent communication links; each node or line is labeled with its mean delay in milliseconds. If we do not have traces from a node, we cannot distinguish its internal delay from network delays, so we represent the combination of such a node and its network links as a diamond, labeled with the total delay for that combination. Time and causality flow left to right, so if a node issues an RPC call, it appears twice in the timeline: once when it sends the call, and again when it receives the return.

These timelines differ from the call-tree pictures traditionally used to represent system structure (for example, Figure 1 in [65] and Figure 1 in [22], or the diagrams

Code name	Hostname
A&M	planetlab2.tamu.edu
A&T	CSPlanet2.ncat.edu
CMU	planetlab-2.cmcl.cs.cmu.edu
CT	planlab2.cs.caltech.edu
How	nodeb.howard.edu
MIT	planetlab6.csail.mit.edu
MU	plnode02.cs.mu.oz.au
ND	planetlab2.cse.nd.edu
PU	planetlab2.cs.purdue.edu
Pri	planetlab-1.cs.princeton.edu
Ro	planet2.cs.rochester.edu
UCL	planetlab2.info.ucl.ac.be
UVA	planetlab1.cs.virginia.edu
WaC	cloudburst.uwaterloo.ca (Coral DHT process)
WaP	cloudburst.uwaterloo.ca (Proxy process)
cl	Any client
dns N	some DNS server
lo	local loopback
web N	some Web origin server

Table 4.3: Abbreviated names for hosts used in WAP5 figures.

in our earlier work [1]) but we found it hard to represent both causality and delay in a call-tree, especially when communication does not follow a strict call-return model. Magpie [4] also uses timelines, although Magpie separates threads or nodes vertically, while we only do so when logically parallel behavior requires it.

It is possible to transform the timeline in Figure 4.10 to a call tree, as in the hand-constructed Figure 4.9, but this loses the visually helpful proportionality between different delays.

To avoid unreadably small fonts, we use short code names in the timelines instead of full hostnames. Table 4.3 provides a translation.

4.6.2 Characterizing causal paths

Our tools allow us to characterize and compare causal path patterns. For example, Figure 4.10 and Figure 4.12 show, for Coral and CoDeeN respectively, causal path patterns that include a cache miss and a DNS lookup. One can see that CoDeeN differs from Coral in its use of two proxy hops (described in [65] as a way to aggregate requests for a given URL on a single CoDeeN node).

A user of our tools can see how overall system delay is broken down into delays on individual hosts and network links. Further, the user can explore how application structure can affect performance. For example, does the extra proxy hop in CoDeeN contribute significantly to client latency?

Note that we ourselves are not able to compare the end-to-end performance of Coral and CoDeeN because we do not have traces made at clients. For example, one CDN might be able to optimize client network latencies at the cost of poorer server load balancing.

4.6.3 Characterizing node delays

When looking for a performance bug in a replicated distributed system, it can be helpful to look for large differences in delay between paths that should behave similarly. Although we do not believe there were any gross performance problems in either CoDeeN or Coral when our traces were captured, we can find paths with significantly different delays. For example, Figure 4.13 shows two different but isomorphic cache-miss paths for CoDeeN (these paths do not require DNS lookups). The origin server delay (a total including both network and server delay) is 321 ms in the top path but only 28 ms in the bottom path. Also, the proxies in the top path show larger delays when forwarding requests than those in the bottom path. In both cases, when a proxy forwards a response, it does so quite rapidly.

Node name	Mean delay	Number of samples
CoDeeN		
planet1.scs.cs.nyu.edu	0.29 ms	583
pl1.ece.toronto.edu	1.47 ms	266
planlab1.cs.caltech.edu	0.59 ms	247
nodeb.howard.edu	4.86 ms	238
planetlab-3.cmcl.cs.cmu.edu	0.20 ms	53
Coral		
planet1.scs.cs.nyu.edu	4.84 ms	6929
planetlab12.Millennium.Berkeley.EDU	6.16 ms	3745
planetlab2.csail.mit.edu	5.51 ms	1626
CSPlanet2.chen.ncat.edu	0.98 ms	987
planetlab14.Millennium.Berkeley.EDU	0.91 ms	595

Table 4.4: Examples of mean delays in proxy nodes.

Similarly, we can focus on just one role in a path and compare the delays at the different servers that fill this role. Table 4.4 shows mean delays in proxies for cache-hit operations (the causal path patterns in this case are trivial).

The message linking algorithm has enough information to generate the entire distribution of delays at a node or on a link rather than just the mean delay. Figure 4.11 shows the delay distributions for cache-hit operations on five nodes. The nodes in this figure are also listed in Table 4.4, which shows mean delay values for four of the nodes between 0 and 2 ms. The distribution for nodeb.howard.edu shows two peaks, including one at 18 ms that strongly implies a disk operation and corresponds with this node's higher mean delay in the table.

4.6.4 DHT paths in Coral

Coral uses a distributed hash table (DHT) to store information about which proxy nodes have a given URL and to store location information about clients.[1] Whenever

[1]The Coral authors call their structure a *distributed sloppy hash table* (DSHT) to emphasize design decisions they made to improve load balancing.

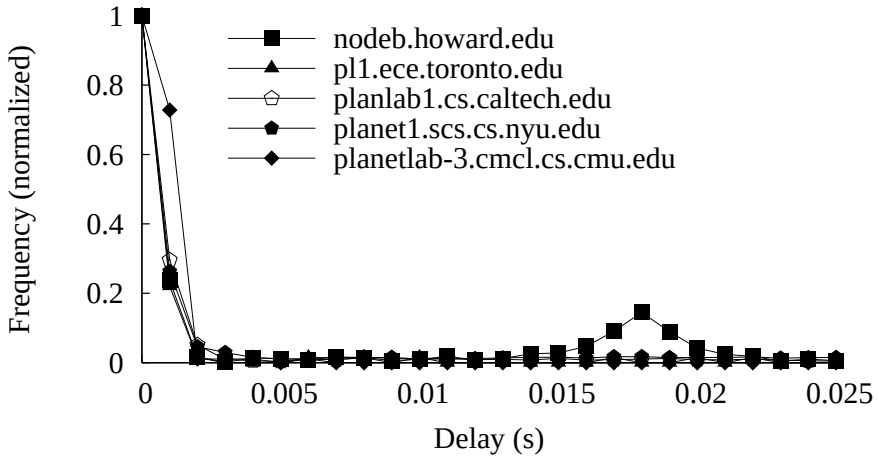


Figure 4.11: Node delay distributions in CoDeeN.

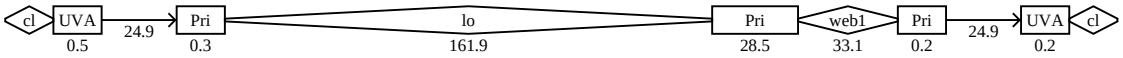


Figure 4.12: Linking algorithm output for a CoDeeN miss path with DNS lookup, delays in ms.

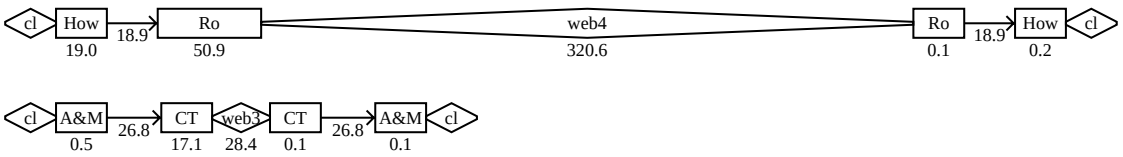


Figure 4.13: Linking algorithm output for two CoDeeN miss paths, delays in ms.

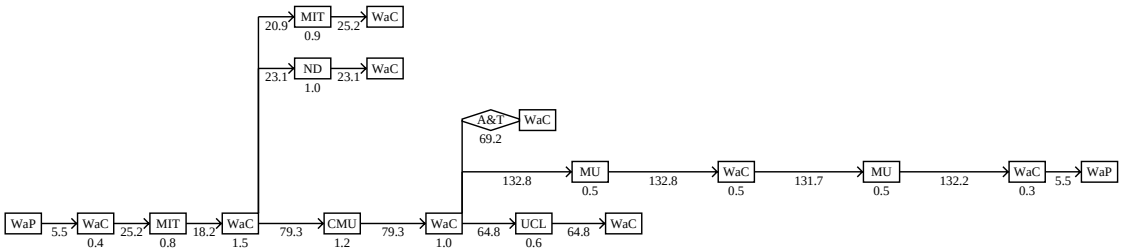


Figure 4.14: Linking algorithm output for a DHT call in Coral, delays in ms.

Trace	Number of messages	Trace duration	Reconciliation CPU secs	Reconciliation MBytes	Message Linking CPU secs	Message Linking MBytes
CoDeeN	4,702,865	1 hour	982	697	730	1354
Coral	4,246,882	1 hour	660	142	517	1148

Table 4.5: Runtime costs for analyzing the CoDeeN and Coral traces.

a proxy does not have a requested web object in its local cache, it searches the DHT to find other nodes that have the object, and it inserts a record into the DHT once it has retrieved the object. Figure 4.14 shows one such DHT call in Coral. The sets of three parallel calls in this figure reflect Coral's use of three overlapping DHTs at different levels of locality. From the figure, it is also clear that Coral's DHT is iterative: each hop in a DHT path responds directly to the requester rather than forwarding the query to the next hop.

4.6.5 Algorithm execution costs

We measured the CPU time and memory required to run the reconciliation and message linking algorithms on several traces. Table 4.5 shows that these costs are acceptable. The CPU time requirements are higher than for the nesting algorithm but lower than for the convolution algorithm [1]. The memory requirements reflect the need to keep the entire trace in memory for both reconciliation and linking. We expect the running time to be $O(n \lg n)$ for both reconciliation and linking because both require sorting. However, the $O(n)$ portions of each program dominate the running

time. The running time for linking is heavily dependent on the pruning parameters used, particularly the number of try-both bits allocated per link probability tree. Memory requirements are $O(n)$ for both programs.

4.6.6 Metrics for sorting path patterns

The linking algorithm produces two scores for each path pattern it identifies: a raw count of the number of instances and the expected number of instances believed to be real. The latter is the sum of the probabilities of all instances of the path pattern. Sorting by the expected number of instances is generally the most useful, in that the patterns at the top of the list appear many times, have high confidence, or both. Highlighting paths that appear many times is useful because they are where optimizations are likely to be useful. Highlighting paths with high confidence helps suppress false positives (i.e., patterns that are inferred but do not reflect actual program behavior).

Two additional, composite metrics are: (1) $expectation \div count$ and (2) $expectation \div \sqrt{count}$. The first is the average probability of instances of each path, and it favors high-confidence paths. The second captures the notion that seeing a path many times increases the confidence that it is not a false positive, but not linearly.

4.7 Enterprise applications

Although this chapter focuses on wide-area applications, previous work on black-box debugging using traces [1, 4, 12] focused on LAN applications. We had the opportunity to try our tools on traces from a moderately complex enterprise application, Slurpee[2]. Slurpee is the one system on which we have used linking, nesting, and convolution. We learned several things applying WAP5 to Slurpee that are applicable

[2]Slurpee is not its real name.

to wide-area applications.

The Slurpee system aids in supporting customers of a computer vendor. It handles reports of incidents (failures or potential failures) and configuration changes. Reports arrive via the Internet, and are passed through several tiers of replicated servers. Between each tier there are firewalls, load balancers, and/or network switches as appropriate, which means that the component servers are connected to a variety of distinct LANs.

Since we could not install LibSockCap on the Slurpee servers, packet-sniffing was our only option for tracing Slurpee and had the advantage of non-invasiveness. However, we found the logistics significantly more daunting than we expected. Packet sniffing systems are expensive, and we could not allocate enough of them to cover all packet paths. They also require on-site staff support to set them up, configure switch ports, initiate traces, and collect the results. In the future, these tasks might be more automated.

We obtained simultaneous packet traces from five sniffers, one for each of the main LAN segments behind the main firewall. We treated each packet as a message, and applied a variant of our reconciliation algorithm (see Section 4.4) to generate a unified trace.

The five sniffers did not have synchronized clocks when the traces were made. Clock offsets were on the order of a few seconds. Since we had two copies of many packets (one sniffed near the sender, one sniffed near the receiver), we developed an algorithm to identify which sniffer's clock to use as the sender or receiver timestamp for each server (node) in the trace. If the sniffer is on the same switch as a node, then every packet to or from that node appears in that sniffer's traces. If we did not sniff the node's switch, then we chose the sniffer that contained the most packets to or from the node.

We applied all three of our causal-path analysis algorithms to the Slurpee trace. The Slurpee trace conforms to call-return semantics but does not contain the information needed to pair calls with returns. We tried several heuristics for pairing calls and returns, but the inaccuracy in this step limited the usefulness of the nesting algorithm. Convolution does not require call-return pairing, but it does require a large number of instances of any given path. Several of the Slurpee paths occurred infrequently and were not detected by convolution. Some of the Slurpee hosts were only visible in the trace for a few seconds and so did not send or receive enough messages to appear in any path detected by convolution. Linking was able to detect both common and rare paths and was not hampered by the lack of call-return pairing information.

4.7.1 Network address translation

In analyzing the Slurpee system, we found instances of network address translation, which did not appear in the Coral or CoDeeN traces but which might appear in other wide-area systems.

Network address translation (NAT) [18] allows network elements to change the addresses in the packets they handle. In Slurpee, a load balancer uses NAT to redirect requests to several server replicas. Wide-area systems often use NAT to reduce the pressure on IPv4 address space assignments. NAT presents a problem for message-based causality analysis, because the sender and receiver of a single message use different "names" (IP addresses) for one of the endpoints.

We developed a tool to detect NAT in packet traces and to rewrite trace records to canonicalize the translated addresses. This tool searches across a set of traces for pairs of packets that have identical bodies and header fields, except for IP addresses and headers that normally change as the result of routing or NAT. While small

numbers of matches might be accidental (especially for UDP packets, which lack TCP's pseudo-random sequence numbers), frequent matches imply the use of NAT. The tool can also infer the direction of packet flow using the IP header's Time-To-Live (TTL) field, and from packet timestamps if we can correct sufficiently for clock offsets.

We have not tested this tool on packet traces from a wide-area system, but we believe it would work correctly. However, because LibSockCap does not capture message contents and cannot capture packet headers, LibSockCap traces do not currently contain enough information to support this tool.

4.8 Summary

We have developed a set of tools collectively called Wide-Area Project 5 (WAP5) that helps expose causal structure and timing in wide-area distributed systems. Our tools include a tracing infrastructure, which includes a network interposition library called LibSockCap and algorithms to reconcile many traces into a unified list of messages; a message-linking algorithm for inferring causal relationships between messages; and visualization tools for generating timelines and causal trees. We applied WAP5 to two content-distribution networks in PlanetLab, Coral and CoDeeN, and to an enterprise-scale incident-monitoring system, Slurpee. We extracted a causal behavior model from each system that matched published descriptions (or, for Slurpee, our discussions with the maintainers). In addition, we were able to examine the performance of individual nodes and the hop-by-hop components of delay for each request.

Chapter 5

Pip: Checking Expectations

In this chapter, we introduce Pip, the last of the three debugging tools. Pip is designed to help isolate unexpected structural and performance behavior automatically by allowing programmers to write explicit, declarative expectations about how their programs should behave. Pip occupies the most intrusive, most accuracy point of the intrusiveness vs. accuracy trade-off: it normally requires annotations in program source code and in return constructs causal paths with arbitrarily fine granularity and perfect accuracy. This improvement in granularity and accuracy permits automatic checking of expectations.

5.1 Overview

Pip is a system for automatically checking the behavior of a distributed system against a programmer's stated expectations about the system. Pip classifies system behaviors as valid or invalid, groups behaviors into sets that can be reasoned about, and presents overall behavior in several forms suited to discovering or verifying the correctness and desired performance of system behavior. Unexpected behavior often indicates a bug. Pip can indicate the block of code responsible for the bug, as well as the conditions that led to it.

Bugs in distributed systems can affect structure, performance, or both. A structural bug results in processing or communication happening at the wrong place or in the wrong order. A performance bug results in processing taking too much or too little of any important resource. For example, a request that takes too long may indicate a bottleneck, while a request that finishes too quickly may indicate truncated

processing or some other error. Pip supports expressing expectations about both structure and performance and so can find a wide variety of bugs.

Our experience shows three major benefits of Pip. First, expectations are a simple and flexible way to express system behavior. Second, automatically checking expectations helps users find bugs that other approaches would not find or would not find as easily. Finally, the combination of expectations and visualization helps programmers explore and learn about unfamiliar systems.

5.1.1 Contributions and results

Pip makes the following contributions:

- An expectations language for writing concise, declarative descriptions of the expected behavior of large distributed systems. We present our language design, along with design principles for handling parallelism and for balancing over- and under-constraint of system behavior.

- A set of tools for gathering events, checking behavior, and visualizing valid and invalid behaviors.

- Tools to generate expectations automatically from system traces. These expectations are often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading them.

We applied Pip to several distributed systems, including FAB [58], SplitStream [10], Bullet [38, 40], and RanSub [39]. Pip automatically generated most of the instrumentation for all four applications. We wrote expectations to uncover unexpected behavior, starting in each case from automatically generated expectations. Pip found unexpected behavior in each application and helped to isolate the causes of poor performance and incorrect behavior.

The rest of this chapter is organized as follows. Section 5.2 contains an overview of the Pip architecture and tool chain. Sections 5.3 and 5.4 describe in detail the design and implementation of our expectation language and annotation system, respectively. Section 5.5 describes the GUI that is an integral part of finding bugs using Pip. Section 5.6 describes our results.

5.2 Architecture

Pip traces the behavior of a running application, checks that behavior against programmer expectations, and displays the resulting valid and invalid behavior in a GUI using several different visualizations.

5.2.1 Behavior model

We define a model of application behavior for use with Pip. This model does not cover every possible application, but we found it natural for the systems we analyzed.

The basic unit of application behavior in Pip is a path instance. Path instances are often causal and are often in response to an outside input such as a user request. A path instance includes events on one or more hosts and can include events that occur in parallel. In a distributed file system, a path instance might be a block read, a write, or a data migration. In a three-tier web service, path instances might occur in response to user requests. Pip allows the programmer to define paths in whatever way is appropriate for the system being debugged.

Each path instance is an ordered series of timestamped events. The Pip model defines three types of events: tasks, messages, and notices. A *task* is like a profiled procedure call: an interval of processing with a beginning and an end, and measurements of resources consumed. Tasks may nest inside other tasks but otherwise may not overlap other tasks on the same thread. Tasks may include asynchronous events

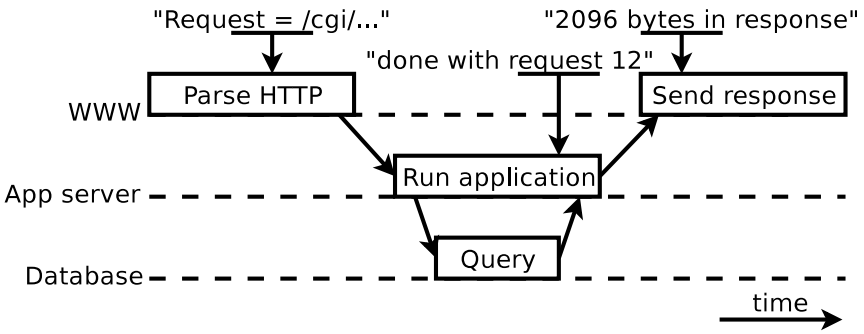


Figure 5.1: Sample causal path from a three-tier system.

like timer callbacks, which Pip normally associates with the path instances that scheduled them. A *message* is any communication event between hosts or threads, whether a network message, a lock, or a timer. Pip records messages when they are sent and again when they are received. Finally, a *notice* is an opaque string—like a log message, with a timestamp and a path identifier for context.

Figure 5.1 shows a sample path instance. Each dashed horizontal line indicates one host, with time proceeding to the right. The boxes are tasks, which run on a single host from a start time to an end time. The diagonal arrows are messages sent from one host to another. The labels in quotation marks are notices, which occur at one instant on a host.

Pip associates each recorded event with a thread. An event-handling system that dispatches related events to several different threads will be treated as having one logical thread. Thus, two path instances that differ only on which threads they are dispatched will appear to have identical behavior.

Our choice of tasks, messages, and notices is well suited to a wide range of distributed applications. Tasks correspond to subroutines that do significant processing. In an event-based system, tasks can correspond to event-handling routines. Messages correspond to network communication, locks, and timers. Notices capture many other types of decisions or events an application might wish to record.

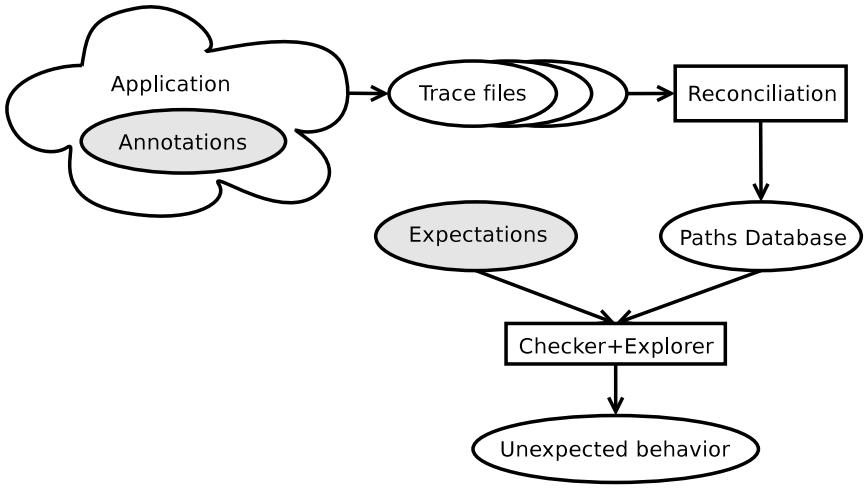


Figure 5.2: Pip workflow. Shaded ovals represent input that must be at least partially written by the programmer.

5.2.2 Tool chain

Pip is a suite of programs that work together to gather, check, and display the behavior of distributed systems. Figure 5.2 shows the workflow for a programmer using Pip. Each step is described in more detail below.

Annotated applications: Programs linked against Pip's annotation library generate events and resource measurements as they run. Pip logs these events into trace files, one per kernel-level thread on each host. We optimized the annotation library for efficiency and low memory overhead; it performs no analysis while the application is running.

We found that the required annotations are easiest to add when communication, event handling, and logging are handled by specialized components or by a supported middleware library. Such concentration is common in large-scale distributed systems. For applications linked against a supported middleware library, a modified version of the library can generate automatic annotations for every network message, remote procedure call, and network-event handler. Programmers can add more annotations

to anything not annotated automatically.

A separate program gathers traces from each host and reconciles them. Reconciliation includes pairing message send and receive events, pairing task start and end events, and performing a few sanity checks. Reconciliation writes events to a database as a series of path instances. Reconciliation is run offline, parsing log files from a short test run. Section 5.4 describes annotations and reconciliation in more detail.

Expectations: Programmers write an external description of expected program behavior. The expectations take two forms: *recognizers*, which validate or invalidate individual path instances, and *aggregates*, which assert properties of sets of path instances. Pip can generate initial recognizers automatically, based on recorded program behavior. These generated recognizers serve as a concise, readable description of actual program behavior. Section 5.3 describes expectations in more detail.

Formally, a set of recognizers in Pip is a grammar, defining valid and invalid sequences of events. In its current form, Pip allows users to define non-deterministic finite-state machines to check a regular grammar. We chose to define a domain-specific language for defining these grammars because our language more closely mirrors how programmers reason about behavior in their applications. We believe this choice simplifies writing and maintaining expectations.

Expectation checker: If the programmer provides any expectations, Pip checks all traced behavior against them. These checks can be done non-interactively, to generate a list of violations, or they can be incorporated into the behavior explorer (below). Section 5.3.5 describes the implementation and performance of expectation checking.

The expectation violations that Pip uncovers do not always indicate bugs in the

system being tested. Sometimes, the errors are in the expectations or in the annotations. Using Pip entails changing the application, the expectations, and the annotations until no further unexpected behavior is found. Unexpected paths due to incorrect expectations or annotations can loosely be called *false positives*, though they are not due to any incorrect inference by Pip.

Behavior explorer: Pip provides an interactive GUI environment that displays causal structure, communication structure, sets of validated and invalidated paths, and resource graphs for tasks or paths. Even without writing any expectations, programmers can visualize most aspects of application behavior. The GUI has three views for exploring the behavior of an individual path instance and a fourth view for exploring the performance behavior of many aggregated tasks or paths. These are described in more detail in Section 5.5. In addition to the GUI, Pip stores all of its path events in a SQL database so that users can write queries or scripts to explore and check application behavior in ways that Pip may not support directly.

5.3 Expectations

Both checking and visualization in Pip start with expectations. Using Pip's declarative expectations language, programmers can describe their intentions about a system's structure, timing, and resource consumption.

5.3.1 Design considerations

Our goal is to provide a declarative, domain-specific expectations language that is more expressive than general-purpose languages, resulting in expectations that are easier to write and maintain. Programmers using Pip should be able to find more complex bugs with less effort than programmers checking behavior with scripts or

programs written in general-purpose languages.

With expressiveness in mind, we present three goals for any expectations language:

1. Expectations written in the language must accept all valid paths. One recognizer should be able to accept a whole family of paths—e.g., all read operations in a distributed file system or all CGI page loads in a webserver—even if they vary slightly. In some systems, particularly event-driven systems, the order of events might vary from one path instance to the next.

2. Expectations written in the language must reject as many invalid paths as possible. The language should allow the programmer to be as specific as possible about task placement, event order, and communication patterns, so that any deviations can be categorized as unexpected behavior.

3. The language should make simple expectations easy to express.

We designed Pip with several real systems in mind: peer-to-peer systems, multicast protocols, distributed file systems, and three-tier web servers, among others. Pip also draws inspiration from two platforms for building distributed systems: Mace[1] [45] and SEDA [66]. The result is that Pip supports thread-oriented systems, event-handling systems, and hybrids. We gave special consideration to event-handling systems that dispatch events to multiple threads in a pool, i.e., for multiprocessors or to allow blocking code in event handlers.

5.3.2 Approaches to parallelism

The key difficulty in designing an expectations language is expressing parallelism. Parallelism in distributed systems originates from three main sources: hosts, threads, and event handlers. Processing happens in parallel on different hosts or on different

[1]Mace is an ongoing redesign of the MACEDON [54] language for building distributed systems.

threads within the same host, either with or without synchronization. Event-based systems may exhibit additional parallelism if events arrive in an unknown order.

Pip first reduces the parallelism apparent in an application by dividing behavior into paths. Although a path may or may not have internal parallelism, a person writing Pip expectations is shielded from the complexity of matching complex interleavings of many paths at once.

Pip organizes the parallelism within a path into threads. The threads primitive applies whether two threads are on the same host or on different hosts. Pip's expectation language exposes threading by allowing programmers to write *thread patterns*, which recognize the behavior of one or more threads in the same path instance.

Even within a thread, application behavior can be nondeterministic. Applications with multiple sources of events (e.g., timers or network sockets) might not always process events in the same order. Thus, Pip allows programmers to write *futures*, which are sequences of events that happen at any time after their declaration.

One early design for Pip's expectation language treated all events on all hosts as a single, logical thread. There were no thread patterns to match parallel behavior. This paradigm worked well for distributed hash tables (DHTs) and three-tier systems, in which paths are largely linear, with processing across threads or hosts serialized. It worked poorly, however, for multicast protocols, distributed file systems, and other systems where a single path might be active on two hosts or threads at the same time. We tried a `split` keyword to allow behavior to occur in parallel on multiple threads or hosts, but it was awkward and could not describe systems with varying degrees of parallelism. The current design, using thread patterns and futures, can naturally express a wider variety of distributed systems.

```
// Read3Others is a validating recognizer
validator Read3Others {
    // no voluntary context switches: never block
    limit(VOL_CS, 0);
    // one Client, issues a read request to Coordinator
    thread Client(*, 1) {
        send(Coordinator) limit(SIZE, {=44b}); // exactly 44 bytes
        recv(Coordinator);
    }
    // one Coordinator, requests blocks from three Peers
    thread Coordinator(*, 1) {
        recv(Client) limit(SIZE, {=44b});
        task("fabrpc::Read") {
            repeat 3 {
                send(Peer);
            }
            repeat 2 {
                recv(Peer);
                task("quorumrpc::ReadReply");
            }
            future { // these statements match events now or later
                recv(Peer);
                task("quorumrpc::ReadReply");
            }
        }
        send(Client);
    }
    // exactly three Peers, respond to Coordinator
    thread Peer(*, 3) {
        recv(Coordinator);
        task("quorumrpc::ReadReq") {
            send(Coordinator);
        }
    }
}
// "assert" indicates an aggregate expectation
assert(average(REAL_TIME, Read3Others) < 30ms);
```

Figure 5.3: Expectations for the FAB read protocol.

```
validator fab_109 {
    thread t_7(*, 1) {
        send(t_9); recv(t_9);
    }
    thread t_9(*, 1) {
        recv(t_7);
        task("fabrpc::Read") {
            send(t_1);
            send(t_1);
            send(t_1);
            recv(t_1);
            task("quorumrpc::ReadReply");
            recv(t_1);
            task("quorumrpc::ReadReply");
        }
        send(t_7);
        recv(t_1);
        task("quorumrpc::ReadReply");
    }
    thread t_1(*, 3) {
        recv(t_9);
        task("quorumrpc::ReadReq") { send(t_9); }
    }
}
```

Figure 5.4: Automatically generated expectation for the FAB read protocol, from which we derived the expectation in Figure 5.3.

5.3.3 Expectation language description

Pip defines two types of expectations: *recognizers* and *aggregates*. A recognizer is a description of structural and performance behavior. Each recognizer classifies a given path instance as matching, matching with performance violations, or non-matching. Aggregates are assertions about properties of sets of path instances. For example, an aggregate might state that a specific number of path instances must match a given recognizer, or that the average or 95th percentile CPU time consumed by a set of path instances must be below some threshold.

Figure 5.3 shows a recognizer and an aggregate expectation describing common read events in FAB [58], a distributed block-storage system. The `limit` statements are optional and are often omitted in real recognizers. They are included here for illustration.

FAB read events have five threads: one client, one I/O coordinator, and three peers storing replicas of the requested block. Because FAB reads follow a quorum protocol, the coordinator sends three read requests but only needs two replies before it can return the block to the client. The final read reply may happen before or after the coordinator sends the newly read block to the client. Figure 5.4 shows a recognizer generated automatically from a trace of FAB, from which we derived the recognizer in Figure 5.3.

The recognizer in Figure 5.3 matches only a 2-of-3 quorum, even though FAB can handle other degrees of replication. Recognizers for other quorum sizes differ only by constants. Similarly, recognizers for other systems might depend on deployment-specific parameters, such as the number of hosts, network latencies, or the desired depth of a multicast tree. In all cases, recognizers for different sizes or speeds vary only by one or a few constants. Pip could be extended to allow parameterized recognizers, which would simplify the maintenance of expectations for systems with multiple,

different deployments.

Pip currently provides no easy way to constrain similar behavior. For example, if two loops must execute the same number of times or if communication must go to and from the same host, Pip provides no means to say so. Variables would allow an expectations writer to define one section of behavior in terms of a previously observed section. Variables are also a natural way to implement parameterized recognizers, as described above.

The following sections describe the syntax of recognizers and aggregate expectations.

Recognizers

Each recognizer can be a *validator*, an *invalidator*, or a building block for other expectations. A path instance is considered valid behavior if it matches at least one validator and no invalidators. Ideally, the validators in an expectations file describe *all* expected behavior in a system, so any unmatched path instances imply invalid behavior. Invalidators may be used to indicate exceptions to validators, or as a simple way to check for specific bugs that the programmer knows about in advance.

Each recognizer can match either complete path instances or fragments. A *complete recognizer* must describe all behavior in a path instance, while a *fragment recognizer* can match any contiguous part of a path instance. Fragment recognizers are often, but not always, invalidators, recognizing short sequences of events that invalidate an entire path. The validator/invalidator and complete/fragment designations are orthogonal.

A recognizer matches path instances much the same way a regular expression matches character strings. A complete recognizer is similar to a regular expression that is constrained to match entire strings. Pip's recognizers define regular languages,

and the expectation checker approximates a finite state machine.

Each recognizer in Pip consists of expectation statements. Each statement can be a literal, matching exactly one event in a path instance; a variant, matching zero or more events in a path instance; a future, matching a block of events now or later; or a limit, constraining resource consumption. What follows is a description of the expectation statements used in Pip. Most of these statements are illustrated in Figure 5.3.

Thread patterns: Path instances in Pip consist of one or more threads or thread pools, depending on system organization. There must be at least one thread per host participating in the path. All complete (not fragment) recognizers consist of thread patterns, each of which matches threads. A whole path instance matches a recognizer if each thread matches a thread pattern. Pip's syntax for a thread pattern is:

thread(*where*, *count*) {*statements*}

Where is a hostname, or "*" to match any host. *Count* is the number of threads allowed to match, or an allowable range. *Statements* is a block of expectation statements.

Literal statements: Literal expectation statements correspond exactly to the types of path events described in Section 5.2. The four types of literal expectation statements are `task`, `notice`, `send`, and `recv`.

A `task` statement matches a single task event and any nested events in a path instance. The syntax is:

task(*name*) {*statements*}

Name is a string or regular expression to match the task event's name. The optional *statements* block contains zero or more statements to match recursively against the task event's subtasks, notices, and messages.

A `notice` statement matches a single notice event. `Notice` statements take a string or regular expression to match against the text of the notice event.

`Send` and `recv` statements match the endpoints of a single message event. Both statements take an identifier indicating which thread pattern or which node the message is going to or arriving from.

Variant statements: Variant expectation components specify a fragment that can match zero or more actual events in a path instance. The five types of variant statements are `repeat`, `maybe`, `xor`, `any`, and `include`.

A `repeat` statement indicates that a given block of code will be repeated n times, for n in a given range. The `maybe` statement is a shortcut for `repeat between 0 and 1`. The syntax of `repeat` and `maybe` is:

> **repeat between** *low* **and** *high* { *statements* }
> **maybe** { *statements* }

An `xor` statement indicates that exactly one of the stated branches will occur. The syntax of `xor` is:

> **xor** {
> **branch:** *statements*
> **branch:** *statements*
> ... *(any number of branch statements)*
> }

An `any` statement matches zero or more path events of any type. An `any` statement is equivalent to ".*" in a regular expression, allowing an expectation writer to avoid explicitly matching a sequence of uninteresting events.

An `include` statement includes a fragment expectation inline as a macro expansion. The `include` statement improves readability and reduces the need to copy and paste code.

Futures: Some systems, particularly event-handling systems, can allow the order and number of events to vary from one path instance to the next. Pip accommodates this fact using `future` statements and optional `done` statements. The syntax for `future` and `done` statements is:

> **future** [*name*] {*statements*}
> **done**(*name*);

A `future` statement indicates that the associated block of statements will match contiguously and in order at or after the current point in the path instance. Loosely, a future states that something will happen either now or later. Futures may be nested: when one future encloses another, it means that the outer one must match before the inner one. Futures may also be nested in (or may include) variant statements. Futures are useful for imposing partial ordering of events, including asynchronous events. Specifying several futures in a row indicates a set of events that may finish in any order. The recognizer in Figure 5.3 uses futures to recognize a 2-of-3 quorum in FAB: two peers must respond immediately, while the third may reply at any later time.

A `done` statement indicates that events described by a given future statement (identified by its name) must match prior to the point of the `done` statement. All futures must match by the end of the path instance, with or without a `done` statement, or else the recognizer does not match the path instance.

Limits: Programmers can express upper and lower limits on the resources that any task, message, or path can consume. Pip defines several metrics, including real

time, CPU time, number of context switches, and message size and latency (the only metrics that apply to messages). A limit on the CPU time of a path is evaluated against the sum of the CPU times of all the tasks on that path. A limit on the real time of a path is evaluated based on the time between the first and last events on the path.

Recognizer sets: One recognizer may be defined in terms of other recognizers. For example, recognizer C may be defined as matching any path instance that matches A and does not match B, or the set difference $A - B$.

Aggregates

Recognizers organize path instances into sets. Aggregate expectations allow programmers to reason about the properties of those sets. Pip defines functions that return properties of sets, including:

- `instances` returns the number of instances matched by a given recognizer.
- `min`, `max`, `avg`, and `stddev` return the minimum, maximum, average, and standard deviation of the path instances' consumption of any resource.

Aggregate expectations are assertions defined in terms of these functions. Pip supports common arithmetic and comparative operators, as well as simple functions like logarithms and exponents. For example:

assert(**average**(CPU_TIME, ReadOperation) $<$ 0.5s);

This statement is true if the average CPU time consumed by a path instance matching the ReadOperation recognizer is less than 0.5 seconds.

5.3.4 Avoiding over- and under-constraint

Expectations in Pip must avoid both over- and under-constraint. An over-constrained recognizer may be too strict and reject valid paths, while an under-constrained recognizer may accept invalid paths. Pip provides variant statements—repeats, xor, and futures—to allow the programmer to choose how specific to be in expressing expectations. Programmers should express how the system should behave rather than how it does behave, drawing upper and lower bounds and ordering constraints from actual program design.

Execution order is particularly prone to under- and over-constraint. For components that devote a thread to each request, asynchronous behavior is rare, and programmers will rarely, if ever, need to use futures. For event-based components, locks and communication order may impose constraints on event order, but there may be ambiguity. To deal with ambiguity, programmers should describe asynchronous tasks as futures. In particular, periodic background events (e.g., a timer callback) may require a future statement inside a repeat block, to allow many occurrences (perhaps an unknown number) at unknown times.

5.3.5 Implementation

The Pip trace checker operates as a nested loop: for each path instance in the trace, check it against each recognizer in the supplied expectations file.

Pip stores each recognizer as a list of thread patterns. Each thread pattern is a tree, with structure corresponding to the nested blocks in the expectations file. Figure 5.5 shows a sample expectation and one matching path. This example demonstrates why a greedy matching algorithm is insufficient to check expectations: the greedy algorithm would match Notice C too early and incorrectly return a match failure. Any correct matching algorithm must be able to check all possible sets of

```
task("A") {                          <task name="A">
    maybe { notice("B"); }               <notice name="B" />
    repeat between 1 and 2 {             <notice name="C" />
        notice(/.*/);                </task>
    }
    notice("C");
}
```

Figure 5.5: Sample fragment recognizer and a path that matches it.

events that variants such as `maybe` and `repeat` can match.

Pip represents each path instance as a list of threads. Each thread is a tree, with structure corresponding to the hierarchy of tasks and subtasks. When checking a recognizer against a given path instance, Pip tries each thread in the path instance against each thread pattern in the recognizer. The recognizer matches the path instance if each path thread matches at least one thread pattern and each thread pattern matches an appropriate number of path threads.

Each type of expectation statement has a corresponding `check` function that matches path instance events. Each `check` function returns each possible number of events it could match. Literal statements (`task`, `notice`, `send`, and `recv`) match a single event, while variant statements (`repeat`, `xor`, and `any`) can match different numbers of events. For example, if two different branches of an `xor` statement could match, consuming either two or three events, `check` returns the set $[2, 3]$. If a literal statement matches the current path event, `check` returns $[1]$, otherwise $\emptyset$. When a `check` function for a variant statement returns $[0]$, it can be satisfied by matching zero events. A failure is indicated by the empty set, $\emptyset$.

The possible-match sets returned by each expectation statement form a search tree, with height equal to the number of expectation statements and width dependent on how many variant statements are present in the expectation. Pip uses a depth-first search to explore this search tree, looking for a leaf node that reaches the end of the

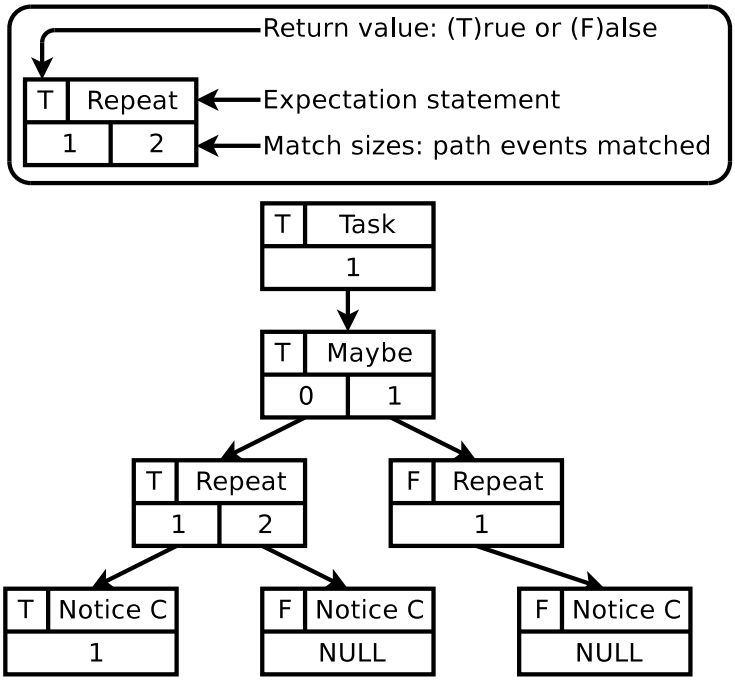


Figure 5.6: Search tree formed when matching the recognizer and the path events in Figure 5.5.

expectation tree and the path tree at the same time. That is, the match succeeds if, in any leaf of the search tree, the expectation matches all of the path events. Node expansion is lazy; thus, the tree is constructed from the left only until a matching leaf is found.

Figure 5.6 shows the possibilities searched when matching the expectations and the path events in Figure 5.5. Each node represents a `check` function call. Each node shows the return value (true or false) of the recursive search call, the expectation statement being matched, and the number(s) of events it can match. Leaves with no possible matches are shown with a possible-match set of `NULL` and a return value of false. A leaf with one or more possible matches might still return false, if any path events were left unmatched.

```
future F1 { notice("C"); }          <task name="A">
task("A") {                              <notice name="B" />
    maybe { notice("B"); }               <notice name="C" />
    repeat between 1 and 2 {        </task>
        notice(/.*/);
    }
}
```

Figure 5.7: The same path as in Figure 5.5, with a slightly modified recognizer to match it. Note that the `notice("C")` statement has been moved into a future block.

Futures

Pip checks futures within the same framework. Each `check` function takes an additional parameter containing a table of all currently available futures. Possible-match sets contain <events matched, futures table> tuples rather than just numbers of events that could be matched. Most `check` calls do not affect the table of active futures, simply returning the same value passed as a parameter. `Future.check` inserts a new entry into the futures table but does not attempt to match any events; it returns a single tuple: <0 events, updated futures table>. `Done.check` forces the named future to match immediately and removes it from the futures table.

Each node in the search tree must try all futures in the table as well as the next expectation statement. If a future matches, then that branch of the tree uses a new futures table with that one future removed. A leaf of the tree matches only if each expectation statement returns success, all path events are consumed, and the futures table is empty.

Figure 5.7 shows the same path instance as in Figure 5.5, with a different expectation to match it: the `notice("C")` statement is now a future. Figure 5.8 shows the possibilities searched, in a depth-first traversal, when matching the expectations and the path events in Figure 5.7. Lazy evaluation again means that nodes are expanded from the left side of the tree depicted in Figure 5.8 only until a matching leaf is found.

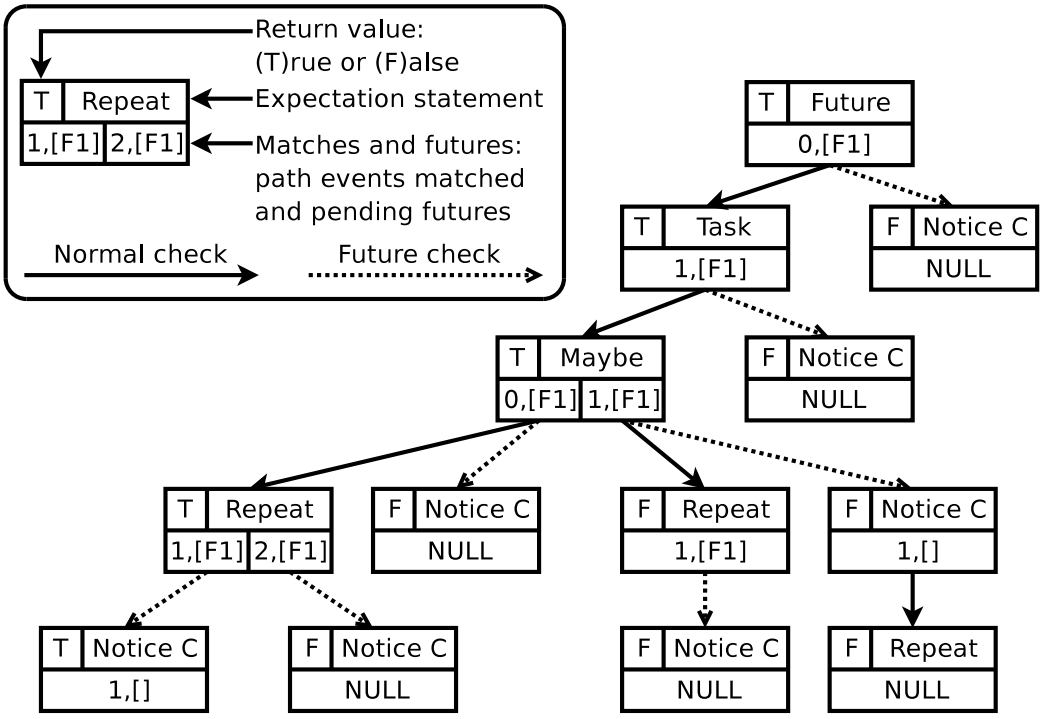


Figure 5.8: The search tree formed when matching the expectation and the path events in Figure 5.7.

Performance

The time to load and check a path instance depends, of course, on the complexity of the path instance and the complexity of the recognizers Pip checks it against. On a 1.6 GHz laptop running Linux 2.6.15 and MySQL 4.1, a complex path instance containing 100 hosts and 1700 events takes about 12 ms to load and another 12 ms to check against seven recognizers, two of which contain futures. Thus, Pip can load and check about 40 complex path instances, or as many as 3400 simple path instances, per second on this hardware.

5.4 Annotations

Pip represents behavior as a list of path instances that contain tasks, notices, and messages, as described in Section 5.2. These events are generated by source-code

annotations. We chose annotations over other event and tracing approaches for two reasons. First, it was expedient. Our focus is expectations and how to generate, check, and visualize them automatically. Second, most other sources of events do not provide a path ID, making them less detailed and less accurate than annotations. Pip could easily be extended to incorporate any event source that provides path IDs.

Pip provides a library, libannotate, that programmers link into their applications. Programmers insert a modest number of source code annotations indicating which path is being handled at any given time, the beginning and end of interesting tasks, the transmission and receipt of messages, and any logging events relevant to path structure.

The six main annotation calls are:

- **annotate_set_path_id**(*id*): Indicate which path all subsequent events belong to. An application must set a path identifier before recording any other events. Path identifiers must be unique across all hosts and all time. Often, identifiers consist of the host address where the path began, plus a local sequence number.

- **annotate_start_task**(*name*): Begin some processing task, event handler, or subroutine. Annotation overhead for a task is around 10 μs, and the granularity for most resource measurements is a scheduler time slice. Thus, annotations are most useful for tasks that run for the length of a time slice or longer.

- **annotate_end_task**(*name*): End the given processing task.

- **annotate_send**(*id*, *size*): Send a message with the given identifier and size. Identifiers must be unique across all hosts and all time. Often, identifiers consist of the address of the sender, an indication of the type of message, and a local sequence number. Send events do not indicate the recipient address, allowing logical messages, anycast messages, forwarding, etc.

- **annotate_receive**(*id*, *size*): Receive a message with the given identifier and size. The identifier must be the same as when the message was sent, usually meaning that at least the sequence number must be sent in the message.

- **annotate_notice**(*string*): Record a log message.

Programs developed using a supported middleware layer may require only a few annotations. For example, we modified Mace [45], a high-level language for building distributed systems, to insert five of the six types of annotations automatically. Our modified Mace adds begin- and end-task annotations for each transition (i.e., event handler), message-send and message-receive annotations for each network message and each timer, and set-path-id annotations before beginning a task or delivering a message. Only notices, which are optional and are the simplest of the six annotations, are left to the programmer. The programmer may choose to add further message, task, and path annotations beyond what our modified Mace generates.

Other middleware layers that handle event handling and network communication could automate annotations similarly. For example, we believe that SEDA [66] and RPC platforms like CORBA could generate message and task events and could propagate path IDs. Pinpoint [12] shows that J2EE can generate network and task events.

5.4.1 Reconciliation

The Pip annotation library records events in local trace files as the application runs. After the application terminates, the Pip *reconciler* gathers the files to a central location and loads them into a single database. The reconciler must pair start- and end-task events to make unified task events, and it must pair message-send and message-receive events to make unified message events.

The reconciler detects two types of errors. First, it detects incomplete (i.e., unpaired) tasks and messages. Second, it detects reused message IDs. Both types of errors can stem from annotation mistakes or from application bugs. In our experience, these errors usually indicate an annotation mistake, and they disappear entirely if annotations are added automatically.

5.4.2 Performance

Pip's useful granularity is limited by three factors: the size of traces that can be efficiently reconciled and checked, the resolution of timers and performance counters, and the overhead of each annotation. Table 5.3, in the next section, shows the time required to reconcile and check several traces. Practically speaking, we avoid producing traces with more than five million events.

Most of Pip's statistics come from the `getrusage` system call, while all timestamps are retrieved using `gettimeofday`. Under recent Linux kernels, `getrusage` has a resolution of 1 ms, and `gettimeofday` has a resolution of 1 μs. However, `getrusage` is insufficient for debugging multithreaded programs, because it returns performance counters for an entire process (per POSIX semantics) while Pip needs information on only the current thread. Thus, Pip reads per-thread performance counters from the `/proc` file system instead, with a resolution of 10 ms. We developed a Linux kernel patch that allows the `getrusage` call to return per-thread information when Pip passes it the proper parameters, which restores the finer 1 ms granularity.

Finally, Pip incurs an absolute run-time overhead per annotation call. On a 1.6 GHz laptop running Linux 2.6.15, the overhead is 0.9 μs for notices and messages and 1.5 μs to change the path ID or begin or end a task. For the `/proc`-based code (i.e., debugging multithreaded programs without the `getrusage` kernel patch), the overheads are much higher: 1.2 μs for notices and 21.1 μs to change path ID or begin

or end a task.

Pip's relative, real-world overhead—i.e., lower throughput or higher end-to-end latency—depends on where the annotations are placed and on how much CPU time the unmodified code consumes. In FAB tests, we found throughput reductions of 6% for read operations on 256 KB blocks, throughput reductions of 9% for a read/write workload on 4 KB blocks, and reductions of 10% for read-only operations on 4 KB blocks. Because read operations on small blocks are the fastest to begin with, they suffer the most relative overhead from a fixed number of annotation calls.

5.5 Behavior explorer GUI

Pip provides a graphical user interface (GUI) that is an integral part of finding structure and performance bugs. The GUI shows both expected and unexpected behavior to help the user figure out the causes and effects of the unexpected behavior. However, the GUI is also useful for exploring systems for which the user has not yet written expectations.

The GUI provides four visualizations: three for visualizing the structure and timing of path instances and a fourth for plotting the aggregate performance of many tasks or path instances. These visualizations are described in detail below.

The first view in the Pip behavior explorer is the causal tree view, shown in Figure 5.9. Here, the behavior of a path instance is shown as a tree of causal steps. Each step contains all of the events, always in a single process on a single host, directly caused by a message sent during the preceding causal step. Thus, an individual host or thread might appear in several steps if it receives and reacts to several messages. Clicking on a causal step shows the list of events the step represents. Clicking on a task or message event in that list shows more detailed information about the individual event.

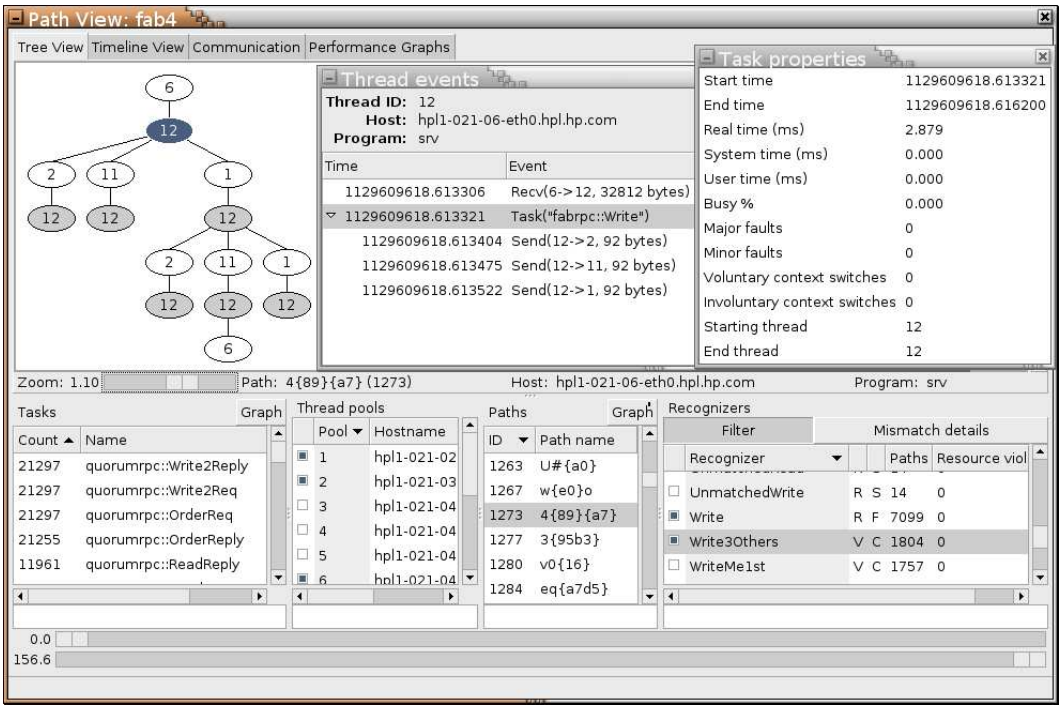


Figure 5.9: Pip path explorer: tree view.

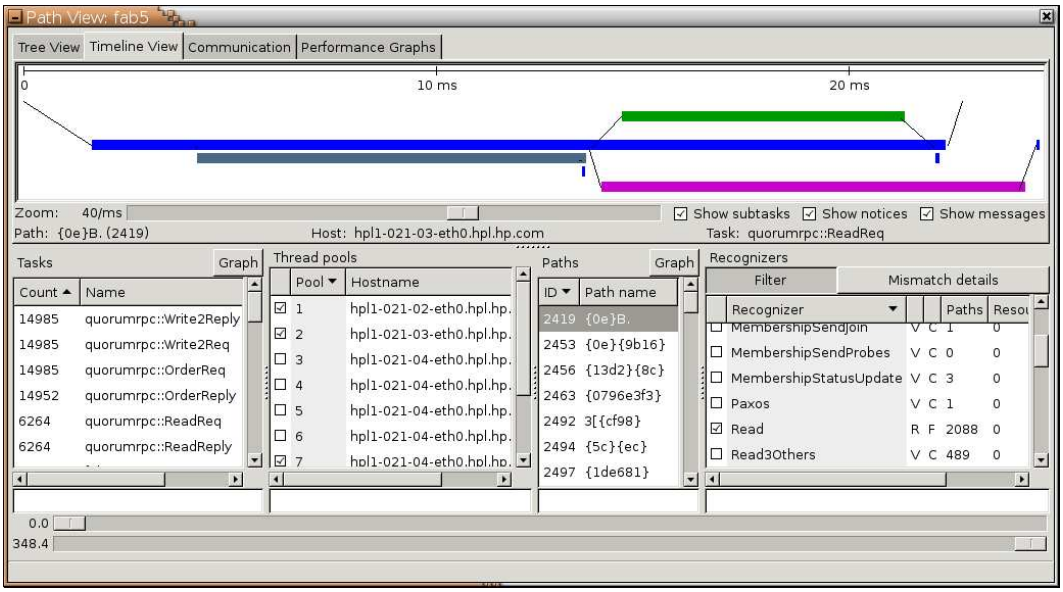


Figure 5.10: Pip path explorer: timeline view.

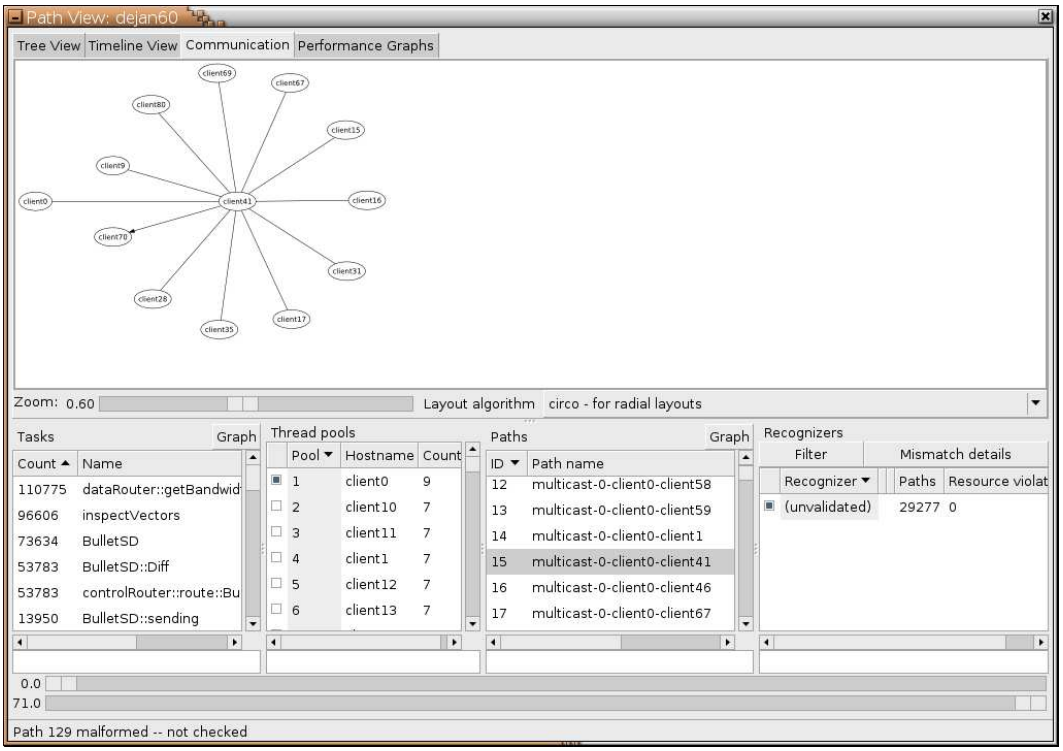


Figure 5.11: Pip path explorer: communication graph view.

The second view is a timeline, as shown in Figure 5.10. One again, only the events from a single path instance are shown at one time. Each horizontal bar corresponds to the events caused by a message, as with the tree nodes described above. The horizontal axis represents time, with the length of each bar showing how long the events took to complete. The vertical axis separates the activity of multiple hosts, each designated by its own color. Adjacent bars of the same color indicate nested subtasks on a single node. Clicking on any node shows the same event-list and event-detail pop-up dialogs described above.

Third, the Pip behavior explorer can display a communication graph, as shown in Figure 5.11. Here, each node corresponds to a host, and each edge indicates that at least one message was sent between a pair of hosts during execution of the selected path instance. Directed edges indicate one-way communication, while undirected edges indicate reciprocal communication.

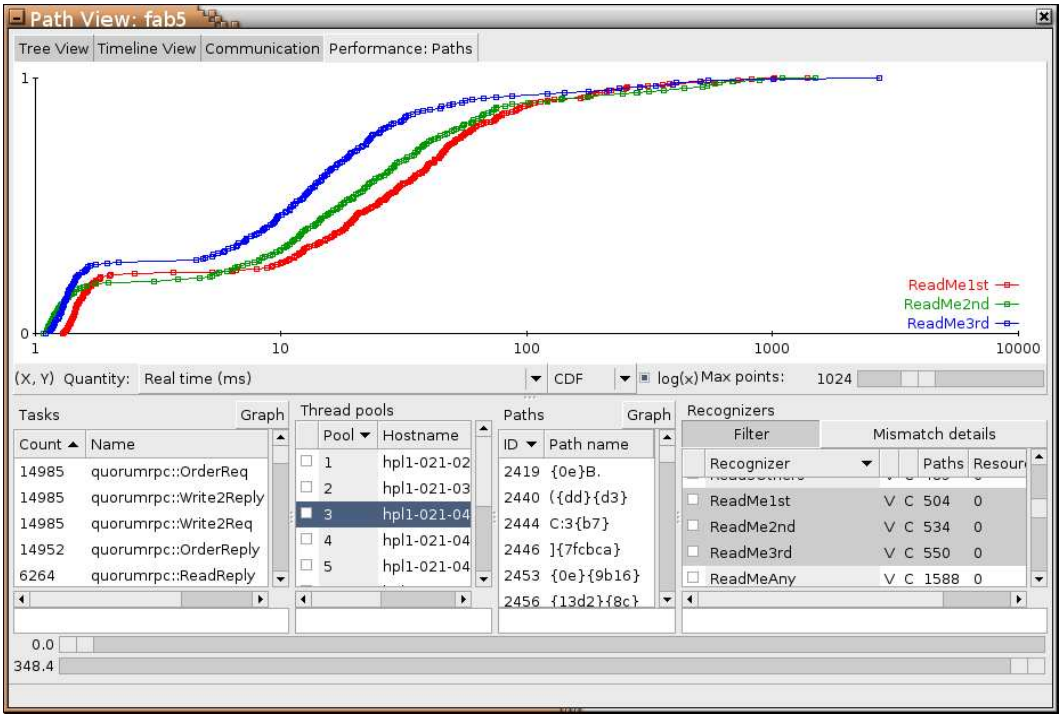


Figure 5.12: Pip path explorer: performance graph view.

The final visualization is a plot of the performance of sets of tasks or path instances. The user can select one or more tasks by name (e.g., all instances of `quorumrpc::ReadReply`) or can plot all path instances. Path instances are aggregated according to which recognizer they match; for example, the user could compare the performance of read and write paths. The metric to plot can be any metric Pip records, including real time, CPU time, network messages, or context switches. The plot can be a cumulative distribution function (CDF), a probability density function (PDF), or recorded values as a function of time. Clicking on a point in the plot shows its exact value, along with the task or path that produced the point. This feature allows the user to explore directly the cause of performance outliers.

The lists in the lower half of the GUI (seen in Figures 5.9–5.12) show task names, host names, path instances, and recognizers. The user can select tasks or paths to show in the four visualizations described above. The user can also select hosts

and recognizers to filter the list of path instances shown. Changing the list of path instances shown also changes the performance graphs. For example, users can plot all path instances that traverse a specific host, or those that match (or do not match) a given recognizer. The text entry lines below the lists allow the user to restrict the tasks, hosts, path instances, or recognizers visible using a substring search.

Finally, the two sliders at the bottom of the GUI restrict the list of path instances in time, relative to the beginning of the trace. The upper slider is the start time for the time window to examine, and the lower slider is the end time. Only path instances with at least one event in the selected time window will be displayed and plotted. By default, the time window is the full duration of the trace.

5.6 Results

We applied Pip to several distributed systems, including FAB [58], SplitStream [10], Bullet [38, 40], and RanSub [39]. We found 18 bugs and fixed most of them. Some of the bugs we found affected correctness—for example, some bugs would result in SplitStream nodes not receiving data. Other bugs were pure performance improvements—we found places to improve read latency in FAB by 15% to 50%. Finally, we found correctness errors in SplitStream and RanSub that were masked at the expense of performance. That is, mechanisms intended to recover from node failures were instead recovering from avoidable programming errors. Using Pip, we discovered the underlying errors and eliminated the unnecessary time the protocols were spending in recovery code.

The bugs we found with Pip share two important characteristics. First, they occurred in actual executions of the systems under test. Pip can only check paths that are used in a given execution. Thus, path coverage is an important, though orthogonal, consideration. Second, the bugs manifested themselves through traced

System	Lines of code	Number of recognizers	Lines of recognizers	Lines of annotations
FAB	124,025	17	590	28
SplitStream	2,436	19	983	8
Bullet	2,447	1	38	23
RanSub	1,699	7	283	32

Table 5.1: Statistics for Pip target systems.

System	Number of hosts	Number of events	Trace duration (sec)
FAB	4	88,054	4 sec
SplitStream	100	3,952,592	104 sec
Bullet	100	863,197	71 sec
RanSub	100	312,994	602 sec

Table 5.2: Statistics for Pip traces.

events. Program annotations must be comprehensive enough and expectations must be specific enough to isolate unexpected behavior. However, the bugs we found were not limited to bugs we knew about. That is, most of the bugs we found were not visible when just running the applications or casually examining their log files.

Table 5.1 shows the size of each system we tested and the effort required to apply Pip to each. Bullet has fewer expectations because we did not write validators for all types of Bullet paths. SplitStream has many expectations because it is inherently complex and because in some cases we wrote both a validator and an overly general recognizer for the same class of behavior (see Section 5.6.2). Table 5.2 shows statistics for the traces we gathered for the target systems. Table 5.3 shows how long Pip took to analyze each trace and how many bugs we found. Over 90% of the running time of reconciliation and checking is spent in MySQL queries; a more lightweight solution for storing paths could yield dramatic speed improvements. In addition to the manual annotations indicated in the table, we added 55 annotation calls to the Mace compiler and library, and we added 19 annotation calls to the FAB IDL compiler.

System	Reconciliation time (sec)	Checking time (sec)	Bugs found	Bugs fixed
FAB	6	7	2	1
SplitStream	1184	837	13	12
Bullet	140	81	2	0
RanSub	47	9	2	1

Table 5.3: Pip run times and results.

Reconciliation time is $O(E \lg p)$ for E events and p path instances, as each event is stored in a database, indexed by path ID. The number of high-level recognizer checking operations is exactly rp for p path instances and r recognizers. Neither stage's running time is dependent on the number of hosts or on the concurrency between paths. The checking time for a path instance against a recognizer is worst-case exponential in the length of the recognizer, e.g., when a recognizer with pathologically nested future and variant statements almost matches a given path instance. In practice, we did not encounter any recognizers that took more than linear time to check.

In the remainder of this section, we will describe our experiences with each system, some sample bugs we found, and lessons we learned.

5.6.1 FAB

A Federated Array of Bricks (FAB) [58] is a distributed block storage system built from commodity Linux PCs. FAB replicates data using simple replication or erasure coding and uses majority voting protocols to protect against node failures and network partitions. FAB contains about 125,000 lines of C++ code and a few thousand lines of Python. All of FAB's network code is automatically generated from IDL descriptions written in Python. The C++ portions of FAB combine user-level threading and event-handling techniques. A typical FAB configuration includes four

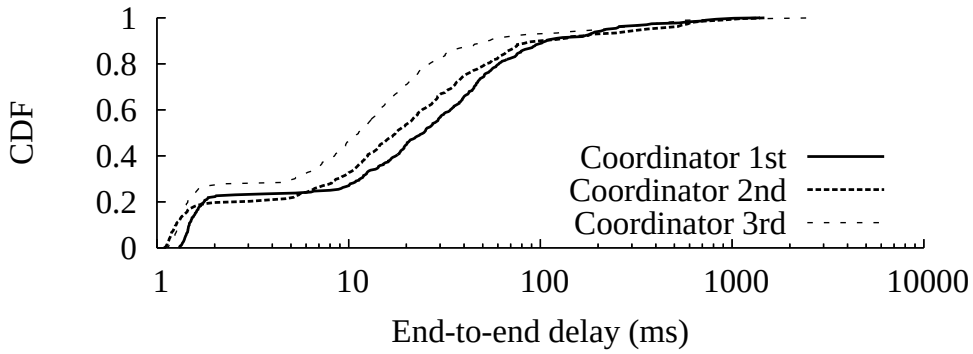


Figure 5.13: CDF of end-to-end latency in milliseconds for FAB read operations. The left-most line shows the case where the coordinator calls itself last. Note that the x axis is log-scaled to show detail.

or more hosts, background membership and consensus communication, and a mix of concurrent read and write requests from one or more clients.

We were not initially familiar with FAB, but we had access to its source code, and one of its authors offered to help us understand it. With just a few hours of effort, we annotated FAB's IDL compiler, and was able to get the tasks and messages necessary to examine every protocol.

Figure 5.3 in Section 5.3.3 showed an expectation for the FAB read protocol when the node coordinating the access (the I/O coordinator) does not contain a replica of the block requested. In this section, we focus on the case where the coordinator does contain a replica. In addition to the read and write protocols, we annotated and wrote expectations for FAB's implementation of Paxos [43] and the Cristian-Schmuck membership protocol [14] but did not find any bugs in either.

Bugs: When the FAB I/O coordinator contains a replica of the block requested, the order of RPCs issued affects performance. In FAB, an RPC issued by a node to itself is handled synchronously. Originally, FAB issued read or write RPCs to all replicas in an arbitrary order. A recent optimization changed this code so that the coordinator always issues the RPC to itself (if any) last, allowing greater overlap of

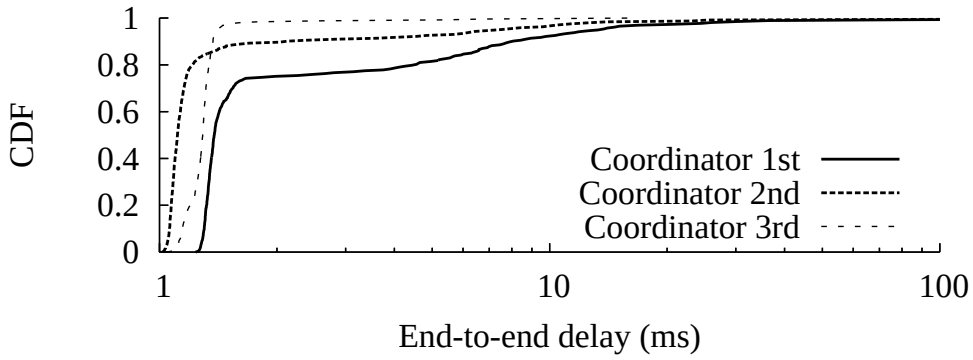


Figure 5.14: CDF of end-to-end latency in milliseconds for FAB read operations in a system with a high cache hit rate. The left-most line shows the case where the coordinator calls itself second. Note that the x axis is log-scaled to show detail.

computation.

FAB's author sent us the unoptimized code without describing the optimization to us, with the intention that we use Pip to rediscover the same optimization. Figure 5.13 shows the performance of read operations when the coordinator calls itself first, second, or last. When the block is not in cache (all delay values about 10 ms), having the coordinator issue an RPC to itself last is up to twice as fast as either other order. Write performance shows a similar, though less pronounced, difference.

We discovered this optimization using expectations and the visualization GUI together. We wrote recognizers for the cases where the coordinator called itself first, second, and third and then graphed several properties of the three path sets against each other. The graph for end-to-end delay showed a significant discrepancy between the coordinator-last case and the other two cases.

Figure 5.14 shows the same measurements as Figure 5.13, in a system with a higher cache hit rate. We noticed that letting the coordinator call itself second resulted in a 15% decrease in latency for reads of cached data by performing the usually unneeded third call after achieving a 2-of-3 quorum and sending a response to the client. The FAB authors were not aware of this difference.

Lessons: Bugs are best noticed by someone who knows the system under test. We wrote expectations for FAB that classified read and write operations as valid regardless of the order of computation. We found it easy to write recognizers for the actual behavior a system exhibits, or even to generate them automatically, but only someone familiar with the system can say whether such recognizers constitute real expectations.

5.6.2 SplitStream

SplitStream [10] is a high-bandwidth content-streaming system built upon the Scribe multicast protocol [56] and the Pastry DHT [55]. SplitStream sends content in parallel over a "forest" of 16 Scribe trees. At any given time, SplitStream can accommodate nodes joining or leaving, plus 16 concurrent multicast trees. We chose to study SplitStream because it is a complex protocol, we have an implementation in Mace, and our implementation was exhibiting both performance problems and structural bugs. Our SplitStream tests included 100 hosts running under ModelNet [64] for between two and five minutes.

Bugs: We found 13 bugs in SplitStream and fixed most of them. We found two of the bugs using the GUI and 11 of the bugs by either using or writing expectations. Seven bugs had gone unnoticed or uncorrected for ten months or more, while the other six had been introduced recently along with new features or as a side effect of porting SplitStream from MACEDON to Mace. Four of the bugs we found were due to an incorrect or incomplete understanding of the SplitStream protocol, and the other nine were implementation errors. At least four of the bugs resulted in inefficient (rather than incorrect) behavior. In these cases, Pip enabled performance improvements by uncovering bugs that might have gone undetected in a simple check of correctness.

One bug in SplitStream occurred twice, with similar symptoms but two different causes. SplitStream allows each node to have up to 18 children, but in some cases was accepting as many as 25. We first discovered this bug using the GUI: visualizations of multicast paths' causal structure sometimes showed nodes with too many children. The cause the first time was the use of global and local variables with the same name; SplitStream was passing the wrong variable to a call intended to offload excess children. After fixing this bug, we wrote a validator to check the number of children, and it soon caught more violations. The second cause was an unregistered callback. SplitStream contains a function to accept or reject new children, but the function was never called.

Lessons: Some bugs that look like structural bugs affect only performance, not correctness. For example, when a SplitStream node has too many children, the tree still delivers data, but at lower speeds. The line between structural bugs and performance bugs is not always clear.

The expectations checker can help find bugs in several ways. First, if we have an expectation we know to be correct, the checker can flag paths that contain incorrect behavior. Second, we can generate recognizers automatically to match existing paths. In this case, the recognizer is an external description of actual behavior rather than expected behavior. The recognizer is often more concise and readable than any other summary of system behavior, and bugs can be obvious just from reading it. Finally, we can write an overly general recognizer that matches all multicast paths and a stricter, validating recognizer that matches only correct multicast paths. Then we can study incorrect multicast paths—those matched by the first but not the second—without attempting to write validators for other types of paths in the system.

5.6.3 Bullet

Bullet [38, 40] is a content-distribution mesh. Unlike overlay multicast protocols, Bullet forms a mesh by letting each downloading node choose several peers, which it will send data to and receive data from. Peers send each other lists of which blocks they have already received. One node can decide to send (push) a list of available blocks to its peers, or the second can request (pull) the list. Lists are transmitted as deltas containing only changes since the last transmission between the given pair of nodes.

Bugs: We found two bugs in Bullet, both of which are inefficiencies rather than correctness problems. First, a given node A sometimes notifies node B of an available block N several times. These extra notifications are unexpected behavior. We found these extra notifications using the reconciler rather than the expectations checker. We set each message ID as <sender, recipient, block number> instead of using sequence numbers. Thus, whenever a block notification is re-sent, the reconciler generates a "reused message ID" error.

The second bug is that each node tells each of its peers about every available block, even blocks that the peers have already retrieved. This bug is actually expected behavior, but in writing expectations for Pip we realized it was inefficient.

Lessons: We were interested in how notifications about each block propagate through the mesh. Because some notifications are pulls caused by timers, the propagation path is not causal. Thus, we had to write additional annotations for *virtual* paths in addition to the causal paths that Mace annotated automatically.

Pip can find application errors using the reconciler, not just using the path checker or the GUI. It would have been easy to write expectations asserting that no node

learns about the same block from the same peer twice, but it was not necessary because the reconciler flagged such repeated notifications as reused message IDs.

5.6.4 RanSub

RanSub [39] is a building block for higher-level protocols. It constructs a tree and tells each node in the tree about a uniformly random subset of the other nodes in the tree. RanSub periodically performs two phases of communication: *distribute* and *collect*. In the distribute phase, each node starting with the root sends a random subset to each of its children. In the collect phase, each node starting with the leaves sends a summary of its state to its parent. Interior nodes send a summary message only after receiving a message from all children. Our RanSub tests involved 100 hosts and ran for 5 minutes.

Because RanSub is written in Mace, we were able to generate all needed annotations automatically.

Bugs: We found two bugs in RanSub and fixed one of them. First, each interior node should only send a summary message to its parent after hearing from all of its children. Instead, the first time the collect phase ran, each interior node sent a summary message after hearing from one child. We found this bug by writing an expectation for the collect-and-distribute path; the first round of communication did not match. The root cause was that interior nodes had some state variables that did not get initialized until after the first communication round. We fixed this bug.

The second bug we found in RanSub is a performance bug. The end-to-end latency for collect-and-distribute paths starts out at about 40 ms and degrades gradually to about 50 ms after running for three minutes. We traced the bottleneck to a task called `deliverGossip` that initially takes 0 ms to run and degrades gradually to

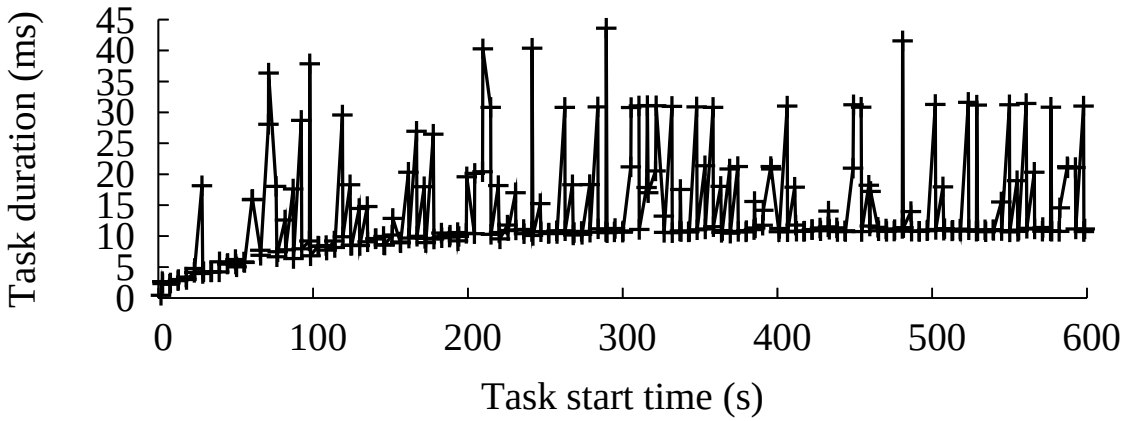


Figure 5.15: Duration for the `deliverGossip` task as a function of time.

about 11 ms. We found this bug using the GUI. First, we examined the end-to-end latency as a function of time. Seeing an error there, we checked each class of tasks in turn until we found the gossip task responsible for the degradation. Figure 5.15 shows the time consumed by the gossip task as a function of time. The reason for `deliverGossip` degrading over time is unclear but might be that `deliverGossip` logs a list of all gossip previously received.

5.7 Summary

Pip helps programmers find bugs in distributed systems by comparing actual system behavior to the programmer's expectations about that behavior. Pip provides visualization of expected and actual behavior, allowing programmers to examine behavior that violates their expressed expectations, and to search interactively for additional unexpected behavior. The same techniques can help programmers learn about an unfamiliar system or monitor a deployed system.

Pip can often generate any needed annotations automatically, for applications constructed using a supported middleware layer. Pip can also generate initial expectations automatically. These generated expectations are often the most readable description of system behavior, and bugs can be obvious just from reading them.

We applied Pip to a variety of distributed systems, large and small, and found bugs in each system.

Chapter 6

Related Work

We have presented three tools that use causal paths for debugging distributed systems. These tools draw inspiration from many earlier tools. Below, we describe other systems that employ causal paths and other systems that infer causal relationships from black-box communication traces. Project 5 is the first to combine these techniques. We also describe systems for automatically checking expectations. Pip is the first tool to combine causal paths with expectations. Finally, we describe model-checking languages, domain-specific languages for building distributed systems, and a few representative systems that use interposition.

6.1 Path analysis tools

Several recent tools use causal paths as the basis for debugging distributed or single-node systems. These tools vary greatly in how they obtain events, how they combine those events into paths, how they aggregate paths, and how they present the paths to a user. None of these tools supports automatically checking programmer-specified or path-related expectations.

Magpie: Probably the work most similar to ours mine is Magpie [4], which also aids performance debugging and anomaly detection based on event traces from single-node and distributed systems. Magpie relies on Event Tracing for Windows (ETW) as its source for trace data. Many existing Windows applications generate ETW events, and the Magpie authors have added ETW hooks to additional components, including the WinPcap packet-capture library, as needed. ETW events include a timestamp, a

name, and zero or more typed attributes, such as thread identifier, CPU identifier, a socket handle, or an application-level path identifier. Magpie depends upon an *event schema* to perform a *temporal join* of these events to generate causal paths. That is, Magpie can connect events in different components, on different hosts, or at different times by finding attributes that they have in common. This grouping is transitive, meaning that events can be grouped even if some pairs have no attributes in common. The result of this temporal join is that each event gets assigned to exactly one causal path, without necessarily requiring modifications to application source code.

Magpie groups all reconstructed causal paths into *request clusters*. Two causal paths have a similarity defined in terms of their edit distance. Different events, different ordering, or different resource measurements will all contribute to a higher edit distance. Each path detected is grouped with the most similar existing request cluster, unless no cluster is similar enough, in which case a new cluster is started. The result of this clustering is that every path is added to exactly one request cluster. Small request clusters indicate anomalous paths that are similar to few or no other paths. Those anomalies are assumed to indicate possible bugs.

Pinpoint: Chen et al. [12] describe Pinpoint, a system for detecting faults in distributed systems and then locating the component most likely to be the cause of each fault. Pinpoint's focus on faults is different from our work, in that our tools help programmers detect and locate performance or correctness problems. Pinpoint [12] constructs causal paths by annotating applications or platforms such as J2EE to generate events and maintain a unique path identifier per incoming request. Like Pip, Pinpoint can construct paths with a high degree of confidence because it does not rely on inference. Like Magpie, Pinpoint assumes that anomalies indicate bugs.

Pinpoint detects faults in three ways. First, it detects path collisions, which are cases when one path effectively aborts another. For example, when a user stops and

restarts a web request, a path collision occurs. Second, Pinpoint detects structural anomalies. Unlike Pip, Pinpoint uses a probabilistic, context-free grammar (PCFG) to detect anomalies on a per-event basis rather than considering whole paths. For example, Pinpoint learns over time that event A always causes exactly one of B or C. If Pinpoint ever sees $A \rightarrow D$ in a path, it will know it is incorrect. Using a PCFG significantly underconstrains path checking, however. A path may be incorrect even though each individual hop is present in the PCFG. As Pinpoint's authors point out, the PCFG can cause Pinpoint to validate some paths with bugs. Finally, Pinpoint assumes that latency changes indicate faults. An increase in the number of outliers, a change in the mean latency, or even latency decreases can all indicate different kinds of failures.

Statistical approaches like Magpie and Pinpoint have three weaknesses. First, they require large amounts of data before anomalies can be detected. Pip, in contrast, can detect errors in a trace that contains even just one path. Second, statistical approaches assume that all bugs are anomalies. Some bugs are present in common paths and would be miscategorized as valid behavior. Finally, these statistical approaches do not cope well with legitimately changing behavior. An increase or decrease in load can change step-by-step latency. A shift in request patterns can make a previously rare path pattern become common or vice versa. An online statistical approach might naively classify new behaviors as invalid.

Other causal path analysis tools: A much earlier project, the Distributed Programs Monitor (DPM) [47], also reports paths of causality through distributed systems. It uses kernel instrumentation to track the causal information between pairs of messages, rather than trying to infer causality from message timestamps. DPM reports an edge between a pair of nodes if any causal path includes that edge. Therefore, the existence of a path in the resulting graph does not necessarily mean that

any real causal path followed all of those edges in that sequence.

Tierney et al. [60] describe NetLogger, a system for real-time diagnosis of performance problems in distributed systems. Their approach requires programmers to add event logging to carefully-chosen points in the application, and generates *lifelines* that correspond to our causal paths. NetLogger provides tools for managing and visualizing logs, but the tools appear unable to aggregate information from many executions of the same causal path.

Hellerstein et al. [29] describe ETE, an approach for measuring both end-to-end response times and the contributing component latencies. Their approach also requires programmers to instrument applications to reveal significant events and to describe interesting transactions ahead of time.

Causeway [11] is a toolkit for adding path-based functionality to applications. Causeway allows programmers to set and retrieve metadata in an application. Causeway propagates the metadata any time the application communicates via a pipe, socket, file, or shared memory. The metadata can contain path identifiers, priority flags, etc. Thus, Causeway could simplify the development of path-based debugging and profiling tools by linking together all events in each path instance. Causeway does not include any higher-level tools for analyzing, aggregating, or checking paths.

6.2 Causality inference tools

Brown et al. [8] also describe a technique aimed at problem (fault) determination based on characterizing dynamic dependencies between components. However, rather than using traces (as in Pinpoint), they perturb system components (for example, by temporarily locking a database table to prevent a component from making progress). Bagchi et al. [3] describe a similar approach based on fault injection. Note that the resulting pair-wise dependencies are less specific than end-to-end causal paths

would be, and the perturbation approach, which is definitely not passive, requires considerable knowledge of the system design.

In Chapter 3 we described an algorithm for discovering causality from traces based on statistical correlation. Zhang and Paxson [67] also use statistical techniques, correlating traffic to detect intruders who subvert hosts for use as "stepping stones" (i.e., intruders who telnet into a host and then out of it, intending to cover their tracks). Huang et al. use analysis of passively-obtained network traces to detect performance problems in wide-area networks [32]. However, they are interested in network-scale phenomena (delay or congestion) rather than causality.

6.3 Expectation checkers

Several existing systems support expressing and checking expectations about structure or performance. Some of the systems operate on traces, others on specifications, and still others on source code. Some support checking performance, others structure, and others both. Some, but not all, support distributed systems.

PSpec [51] is the work that most directly influenced the expectations in Pip. PSpec allows programmers to write assertions about the performance of systems. It uses information from application event logs to power three tools: a checker, a solver, and an evaluator. The checker is most analogous to Pip: it uses timing values from the log file to check performance assertions specified by the programmer. Unlike Pip, PSpec does not support structural expectations. PSpec allows the programmer to write assertions with symbolic constants; the solver finds an appropriate value for each constant, for a given trace. Finally, the evaluator is an interactive read-eval-print loop that the programmer can use to evaluate expressions defined in PSpec's expectation language.

The PSpec expectation language allows the programmer to define intervals and

write assertions about properties of those intervals. For example, the programmer can specify a *read* operation as occurring between a read-start event and a read-end event. Alternatively, the read *inter-arrival time* is the time between one read-start event and the next read-start event. The programmer can then specify bounds for the minimum, mean, or maximum duration of these intervals. The language also supports arithmetic and set logic, to allow more advanced assertions based on properties of intervals.

Meta-level compilation (MC) [19] checks source code for static bugs using a compiler extended with system-specific rules. MC checks all code paths exhaustively but is limited to single-node bugs that do not depend on dynamic state. MC works well for finding the root causes of bugs directly, while Pip detects symptoms and highlights code components that might be at fault. MC focuses on individual incorrect statements, while Pip focuses on the correctness of causal paths, often spanning multiple nodes. MC finds many false positives and bugs with no visible symptoms, while Pip is limited to actual bugs present in a given execution of the application.

Partiqle [27] checks structural expectations against running Java programs without requiring source-code modifications. Programmers write queries in a Program Trace Query Language (PTQL) that are injected into a program's Java bytecode and executed as the program runs. Partiqle also supports some limited performance-related queries by exposing the start and end times for functions and the allocation and collection times for objects. PTQL operates efficiently at a fine granularity, exposing every function in a program. However, it does not support multi-node systems or path-level expectations, and it is limited to Java.

Paradyn [48] is a performance measurement tool for complex parallel and distributed software. The Paradyn Configuration Language (PCL) allows programmers to describe expected characteristics of applications and platforms, and in particular

to describe metrics; PCL seems somewhat analogous to Pip's expectation language. However, PCL cannot express the causal path structure of threads, tasks and messages in a program, nor does Paradyn reveal the program's structure.

6.4 Model checkers

Programmers may find some bugs using model checking [26, 42]. Model checking is exhaustive, covering all possible behaviors, while Pip and all the other techniques described here check only the behaviors exhibited in actual runs of the system. However, model checking is expensive and is practically limited to small systems and short runs—often just tens of events. Model checking is often applied to specifications, leaving a system like Pip to check the correctness of the implementation. Finally, unlike model checking, Pip can check performance characteristics.

MaceMC [36] is an approach to implementation-based model checking that is specific to the Mace programming language. MaceMC takes advantage of properties of Mace to provide faster, deeper exploration of a program's state space. MaceMC uses model checking to systematically explore the state space of a distributed system, finding safety or liveness bugs that may not correspond to any actual run. MaceMC does not use an external description of expectations, but instead uses traditional assertions to express the presence of potential safety or liveness bugs.

6.5 Domain-specific languages

Developers of distributed systems have a wide variety of specification and implementation languages to choose from. Languages like Estelle [33], π-calculus [49], join-calculus [20], and P2 [44, 59] embrace a formal, declarative approach. Mace [45], Erlang [9], and Mlog [30] use an imperative approach, with libraries and language

constructs specialized for concurrency and communication. Finally, many programmers still use traditional, general-purpose languages like Java and C++.

Declarative domain-specific languages lend themselves to static analysis of structure, namely, to proving invariants about correct execution of protocols. Imperative domain-specific languages lend themselves to language-specific debuggers. All of these languages aid annotation by having high-level constructs that correspond to the path model we use in Pip. While programmers using declarative languages can verify the correctness of their programs through static analysis, Pip remains valuable for monitoring and checking dynamic properties of a program, such as latency, throughput, concurrency, and node failure.

6.6 Interposition

Many existing systems use interposition. Trickle [61] uses library interposition to provide user-level bandwidth limiting. ModelNet [64] rewrites network traffic using library interposition to multiplex emulated addresses on a single physical host. Systems such as Transparent Result Caching [63] and Interposition Agents [34] use debugging interfaces such as `ptrace` or `/proc` to intercept system calls instead of library interposition. Library interposition is simpler and more efficient, but it requires either dynamically linked binaries or explicit relinking of traced applications. WAP5 employs an interposition library similar to the ones used by Trickle and ModelNet.

6.7 Summary

Many existing systems support debugging on one or several nodes. Several earlier systems have employed some of the same techniques we employed in this thesis: causal paths, black-box causality inference, expectation checking, and interposition. Project 5 was the first system to support inferring causal paths from black-box net-

work traces. WAP5 was the first system to apply black-box causal path inference to wide-area systems. Pip was the first system to apply automatic expectation checking to causal paths.

Chapter 7

Conclusions and Future Work

We conclude this dissertation by restating the hypothesis, summarizing the contributions herein, and describing possible future work.

7.1 Contributions

Our hypothesis in this dissertation is that expressing distributed system behavior as a set of causal paths and helping programmers separate those paths into expected and unexpected behavior is a powerful technique for improving the performance and correctness of distributed systems. To support this thesis, we introduce three debugging tools and an expectation language, and we describe how we applied them to understand and debug real systems.

Our research and experiences have led to four main contributions:

1. We introduce three debugging tools: Project 5, Wide-Area Project 5 (WAP5), and Pip. Project 5 infers causal paths through a distributed system using traces of network events. The system itself is treated as a black box and does not need to be modified. WAP5 infers causal paths in wide-area systems, which exhibit significant network delays, unsynchronized clocks, and network architectures that do not support sniffers. Finally, Pip obtains causal paths directly via modifications to system source code and checks those paths against a set of programmer-specified expectations.

2. We describe three algorithms for inferring causality from communication events. First, Project 5's nesting algorithm attempts to find causal relationships be-

tween "nested" remote procedure calls (RPCs) made into and out of each node in a trace. Second, WAP5's linking algorithm seeks a direct cause for every message sent in a trace. Finally, Pip forms causal paths out of associated sets of events based on timing and message ordering.

3. We describe a language for expressing expectations about the behavior of a distributed system. Pip's expectation language allows programmers to describe the behavior of a wide variety of applications. Applications may be parallel or serial, asynchronous or synchronous, threaded or event-based, or a combination of the above. Pip's expectation language aims to balance over- and under-constraint: specifying path behavior in enough detail to accept all valid paths and reject most or all invalid ones.

4. We develop an understanding of the trade-off between disruptiveness and accuracy in distributed debugging tools. Project 5 is the least disruptive to apply and produces the least accurate and least detailed results. Pip is more disruptive, normally requiring changes to application source code, but it enables fine-grained debugging with enough accuracy to allow automatic expectation checking.

7.2 Future work

Our experiences with debugging distributed systems have left interesting open questions and avenues for future research.

7.2.1 Refinements

Pip language improvements: The Pip expectations language could benefit from a few further improvements: parameterized recognizers, variables, extensible anno-

tations, and additional performance metrics.

The recognizers we wrote for FAB (see Figure 5.3) match only a 2-of-3 quorum, even though FAB can handle other degrees of replication. Recognizers for other quorum sizes differ only by constants. Similarly, recognizers for other systems might depend on deployment-specific parameters, such as the number of hosts, network latencies, or the desired depth of a multicast tree. In all cases, recognizers for different sizes or speeds vary only by one or a few constants. Pip could be extended to allow parameterized recognizers, which would simplify the maintenance of expectations for systems with multiple, different deployments.

Pip currently provides no easy way to constrain similar behavior. For example, if two loops must execute the same number of times or if communication must go to and from the same host, Pip provides no means to say so. Variables would allow an expectations writer to define one section of behavior in terms of a previously observed section. Variables are also a natural way to implement parameterized recognizers, as described above.

Pip currently records task names and notices as opaque strings, which do not allow applications to record or check any other data types. We plan to add an annotation to record structured tuples of data containing numbers, addresses, timestamps, and strings, among possibly other types. We will record these events in the paths database and will provide expectations to check and aggregate the values recorded.

Pip's current performance metrics cover real time, CPU usage, and network message characteristics. They do not measure energy consumption, memory allocation, or disk (or other I/O) requests. Attributing energy consumption to causal paths could be useful for debugging power efficiency in portable devices or waste-heat generation in data centers. Tracking memory allocation could help locate memory leaks or components at fault for swap thrashing behavior. Tracking disk requests would be

particularly useful in disk-bound (rather than CPU-bound) systems, including FAB.

WAP5 service time distributions: WAP5 currently assumes that service times at all nodes are exponentially distributed. WAP5 can accommodate other service time distributions; for example, we have tried a gamma distribution. Other distributions might result in higher inference accuracy if they more closely fit the actual distribution of the traced delays. However, other distributions might result in lower accuracy if they do not fit the actual delay distribution or if they have incorrect parameters, such as the mean service time. The exponential distribution is tolerant of incorrect parameter values, because it is monotonically decreasing and has no sharp peak.

Ideally, WAP5 could choose the appropriate distribution independently and dynamically for each node in a system. This choice should be made based on actual observed delays. The choice of appropriate distributions and scaling parameters is likely to be iterative. That is, an initial choice would let WAP5 associate incoming and outgoing messages, which improves the quality of the information available for choosing the distribution and scaling parameters.

7.2.2 Online analysis and monitoring

All three of our debugging tools currently operate only in an offline mode. That is, each tool requires the user to run the targeted system, gather a trace, and then analyze the trace for bugs or other unexpected behavior. With small modifications, all three systems could be used online, either to enable quicker test-and-debug cycles or to enable monitoring for sudden unexpected behavior. Pip in particular could monitor system execution silently until it detects unexpected behavior and then send alerts to a programmer or an administrator.

The debugging tools have CPU and bandwidth requirements based on the rate of event generation in the targeted system. To ensure scalability, any online analysis

would have to provide efficient data-gathering techniques and address load balancing. Data gathering techniques like overlay networks, filtering, and aggregation could improve scalability. In the case of Pip, paths can be constructed and checked in parallel at many hosts.

7.2.3 Trace-based simulation

Pip captures an enormous amount of data when an application runs. So far, we have used it only to generate the behavior model that Pip checks against the programmer's expectations. However, these traces have other potential uses. One such use is reasoning about the performance effects of small changes to a distributed application. We could write a tool to automate this reasoning. Using this tool, a programmer could pose questions about the effect of changing or consolidating the hosts in the application, changing routing decisions, changing the application's processing requirements or available resources, or increasing the application's offered load. The tool would then simulate the modified application based on the behavior of the original. The programmer would be able to test performance improvements without the trouble of rewriting, testing, and redeploying the application. We would like to explore the types of questions and modifications such a tool could accommodate and how accurately it could predict the resulting performance.

7.2.4 Unifying logging and tracing

Pip's notices allow programmers to specify arbitrary events of interest, much like logging statements. One logical extension would be to intercept non-Pip logging statements and have them generate Pip notices. These new notices would benefit from the timestamps and context (path, host, process, and thread) that Pip records for every event. For applications written in Mace, we could redirect all logging

statements to Pip notices. For other applications, we could intercept calls to `syslog` or writes to standard error.

7.3 Summary

Distributed systems are academically interesting and commercially important. They underlie every major site and service on the internet. However, they are more complex to develop and maintain than traditional, single-node systems. Complexity leads to performance and correctness bugs, which leads to slow and unreliable application performance for end users.

Traditional debugging and profiling tools are not well suited for distributed systems, which have large amounts of state, high parallelism, and complex administrative boundaries. In this thesis, we presented a debugging methodology—offline debugging of causal paths—and three tools to enable it in distributed systems.

The first debugging tool, Project 5, treats the targeted system as a set of black boxes. Each node is a closed device that communicates with other nodes. Project 5 infers causality and timing information from traces of network messages, normally obtained using a network sniffer. Project 5 is the least disruptive and, at least in theory, the easiest to apply of the three debugging tools. However, it is also the coarsest and the least accurate. Its granularity is limited to what happens between pairs of messages. Each communicating node is an entire host, and each processing step encompasses everything between an incoming message and an outgoing message. Its accuracy is limited to what can be inferred through statistical analysis. Accuracy suffers in traces with high parallelism or highly variable processing delays.

The second debugging tool we presented, WAP5, assumes the ability to interpose a new library beneath each application component. Doing so requires that the application components run on a supported platform—in the case of WAP5, Linux—and

requires restarting each component to start or stop tracing. The expected benefit is greater granularity and accuracy. WAP5 can distinguish between threads or processes on each host and can capture intra-host communication. It also traces network events at a level closer to application semantics, capturing each network call rather than each network packet. Finally, WAP5 captures separate timestamps when each packet is sent and received, allowing WAP5 to tolerate arbitrarily high levels of clock offset and clock drift.

The final debugging tool we presented, Pip, normally requires programmers to modify and recompile their applications. The required modifications are small, marking whatever communication, processing tasks, and event handlers make up the paths to be debugged. In some cases, these annotations can be inserted automatically by a compiler or middleware layer. Pip also requires the programmer to write expectations about program behavior in a declarative language, though initial expectations can be generated automatically from traces of actual program behavior.

In exchange for this high level of disruptiveness, Pip promises the best granularity and accuracy of the three tools. Pip can trace arbitrarily small communication or processing tasks, depending on where the programmer added annotations. Pip detects paths with perfect accuracy by relying on annotations rather than statistical inference. If the programmer writes expectations about system behavior, Pip can check actual behavior against expected behavior and indicate any differences.

We applied our three debugging tools to a variety of real and synthetic application traces. All three proved useful for analyzing and debugging distributed systems.

References

[1] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, Bolton Landing, NY, October 2003.

[2] Akamai Technologies, Inc. http://www.akamai.com/.

[3] Saurabh Bagchi, Gautam Kar, and Joseph L. Hellerstein. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In *Proc. Intl. Workshop on Distributed Systems: Operations & Management*, Nancy, France, October 2001.

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modeling. In *Proc. OSDI*, pages 259–272, San Francisco, CA, December 2004.

[5] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating System Support for Planetary-Scale Services. In *Proc. NSDI*, pages 253–266, March 2004.

[6] Simon P. Booth and Simon B. Jones. Walk backwards to happiness—debugging by time travel. In *Automated and Algorithmic Debugging*, Linköping, Sweden, May 1997.

[7] Dan Bourque. Time travel made possible with Eclipse. In *EclipseCon*, Santa Clara, CA, March 2006.

[8] Aaron Brown, Gautam Kar, and Alexander Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proc. IFIP/IEEE Intl. Symp. on Integrated Network Management*, Seattle, WA, May 2001.

[9] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. Core Erlang 1.0 language specification. Technical Report 030, Uppsala University, November 2000.

[10] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proc. SOSP*, Bolton Landing, NY, October 2003.

[11] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: Operating system support for controlling and analyzing the execution of distributed programs. In *Proc. HotOS*, 2005.

[12] Mike Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Proc. NSDI*, pages 309–322, San Francisco, CA, March 2004.

[13] Bram Cohen. Incentives build robustness in BitTorrent. In *Proc. Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, June 2003.

[14] Flaviu Cristian and Frank Schmuck. Agreeing on processor group membership in timed asynchronous distributed systems. Report CSE95-428, UC San Diego, 1995.

[15] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for Low Latency and High Throughput. In *Proc. NSDI*, pages 85–98, March 2004.

[16] DDD: The data display debugger. http://www.gnu.org/software/ddd/, 2006.

[17] eDonkey. http://www.edonkey2000.com/index.html.

[18] Kjeld Borch Egevang and Paul Francis. The IP Network Address Translator (NAT). RFC 1631, IETF, May 1994.

[19] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proc. OSDI*, December 2000.

[20] Cédric Fournet and Georges Gonthier. The join calculus: a language for distributed mobile programming. In *Proc. APPSEM*, Caminha, Portugal, 2000.

[21] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-based scalable network services. In *Proc. SOSP*, Saint-Malo, France, October 1997.

[22] Michael J. Freedman, Eric Freudenthal, and David Maziéres. Democratizing content publication with Coral. In *Proc. NSDI*, pages 239–252, San Francisco, CA, March 2004.

[23] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, September 2000.

[24] GDB: The GNU project debugger. http://www.gnu.org/software/gdb/gdb.html, 2005.

[25] Gnutella. http://www.gnutella.com/.

[26] Patrice Godefroid. Software model checking: the VeriSoft approach. *Formal Methods in System Design*, 26(2):77–101, March 2005.

[27] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *Proc. OOPSLA*, October 2005.

[28] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: A call graph execution profiler. In *Proc. SIGPLAN Symp. on Compiler Construction*, pages 120–126, Boston, MA, June 1982.

[29] Joseph L. Hellerstein, Mark Maccabee, W. Nathaniel Mills, and John J. Turek. ETE: A customizable approach to measuring end-to-end response times and their components in distributed systems. In *Proc. ICDCS*, pages 152–162, Austin, TX, May 1999.

[30] C. Hrischuk, J. Rolia, and C.M. Woodside. Automatic generation of a software performance model using an object-oriented prototype. In *Proc. MASCOTS*, pages 399–409, Durham, NC, January 1995.

[31] Yang hua Chu, Sanjay G. Rao, Srinivasan Seshan, and Hui Zhang. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast*, 20(8), Oct 2002.

[32] Polly Huang, Anja Feldmann, and Walter Willinger. A non-intrusive, wavelet-based approach to detecting network performance problems. In *Proc. Internet Measurement Workshop*, San Francisco, CA, November 2001.

[33] ISO 9074. Estelle: A formal description technique based on an extended state transition model. 1987.

[34] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proc. SOSP*, pages 80–93, Asheville, NC, December 1993.

[35] Kazaa. http://www.kazaa.com/.

[36] Charles Killian. MaceMC: Model checking for Mace, 2005.

[37] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. USENIX*, Anaheim, CA, April 2005.

[38] Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX*, April 2005.

[39] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using random subsets to build scalable network services. In *Proc. USITS*, March 2003.

[40] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. SOSP*, Bolton Landing, NY, October 2003.

[41] John Kubiatowicz, Davic Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, Cambridge, MA, November 2000.

[42] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.

[43] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.

[44] Boon Thau Loo, Tyson Condie, Joseph Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Proc. SOSP*, Brighton, UK, October 2005.

[45] Mace. http://mace.ucsd.edu/, 2005.

[46] Petros Maniatis, Mema Roussopoulos, TJ Giuli, David S. H. Rosenthal, Mary Baker, and Yanto Muliadi. LOCKSS: A peer-to-peer digital preservation system. *ACM TOCS*, 23(1):2–50, February 2005.

[47] B. P. Miller. DPM: A measurement system for distributed programs. *IEEE Trans. on Computers*, 37(2):243–248, Feb 1988.

[48] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[49] Robin Milner. The polyadic π-calculus: A tutorial. Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[50] Vern Paxson. On calibrating measurements of packet transit times. In *Proc. SIGMETRICS*, Madison, WI, March 1998.

[51] Sharon E. Perl and William E. Weihl. Performance assertion checking. In *Proc. SOSP*, pages 134–145, December 1993.

[52] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. NSDI*, San Jose, CA, May 2006.

[53] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: black-box performance debugging for wide-area systems. In *Proc. WWW*, Edinburgh, Scotland, May 2006.

[54] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proc. NSDI*, pages 267–280, San Francisco, CA, March 2004.

[55] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[56] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-scale Event Notification Infrastructure. In *3rd Intl. Workshop on Networked Group Communication*, Nov 2001.

[57] Yasushi Saito, Brian N. Bershad, and Henry M. Levy. Manageability, availability and performance in Porcupine: A highly scalable, cluster-based mail service. In *Proc. SOSP*, Kiawah, SC, December 1999.

[58] Yasushi Saito, Svend Frolund, Alistair Veitch, Arif Merchant, and Susan Spence. FAB: Building distributed enterprise disk arrays from commodity components. In *Proc. ASPLOS*, pages 48–58, Boston, MA, 2004.

[59] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using queries for distributed monitoring and forensics. In *Proc. EuroSys*, Leuven, Belgium, April 2006.

[60] Brian Tierney, William E. Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *Proc. HPDC*, July 1998.

[61] Trickle lightweight userspace bandwidth shaper. http://monkey.org/˜marius/trickle/.

[62] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. John Wiley and Sons, New York, 2001.

[63] Amin Vahdat and Thomas Anderson. Transparent result caching. In *Proc. USENIX*, June 1998.

[64] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI*, 2002.

[65] Limin Wang, KyoungSoo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proc. USENIX*, pages 171–184, Boston, MA, June 2004.

[66] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. SOSP*, pages 230–243, 2001.

[67] Yin Zhang and Vern Paxson. Detecting stepping stones. In *Proc. USENIX Security Symp.*, Denver, CO, August 2000.

Biography

Born

Austin, Texas, January 24th, 1978

Colleges and universities

1999-2006 Duke University Durham, NC
M.S. in Computer Science, December 2003.
Ph.D. in Computer Science, May 2006.

1996-1999 University of Virginia Charlottesville, VA
B.S. in Computer Science, May 1999.

1995-1996 Virginia Tech Blacksburg, VA

Publications

"Pip: detecting the unexpected in distributed systems," Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, May 2006.

"WAP5: black-box performance debugging for wide-area systems," Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. In *Proceedings of the 15th International World Wide Web Conference*, May 2006.

"Mace: language support for building distributed systems," Charles Killian, James W. Anderson, Ryan Braud, Patrick Reynolds, Ranjit Jhala, and Amin Vahdat. In submission.

"Performance debugging for distributed systems of black boxes," Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

"Efficient peer-to-peer keyword searching," Patrick Reynolds and Amin Vahdat. In *Proceedings of Middleware*, June 2003.

"Self-organizing subsets: from each according to his abilities, to each according to his needs," Amin Vahdat, Jeffrey Chase, Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, and Patrick Reynolds. In *Proceedings of the First International Peer to Peer Symposium (IPTPS)*, March 2002.

"Passport: a scalable, self-organizing peer-to-peer system," Patrick Reynolds and Amin Vahdat. Duke University Technical Report, May 2001.

Awards

Duke Computer Science Service Award: 2003.

National Science Foundation Graduate Research Fellowship, 2001–2004.

James B. Duke Graduate Fellowship, Duke University, 1999–2003.