Improving Visibility of Distributed Systems through Execution Tracing

Rodrigo Fonseca

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-167
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-167.html

December 18, 2008

Improving Visibility of Distributed Systems through Execution Tracing

by

Rodrigo Lopes Cançado Fonseca

B.S. (Universidade Federal de Minas Gerais) 2000
M.S. (Universidade Federal de Minas Gerais) 2002

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Sciences

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Ion Stoica, Chair
Professor Randy H. Katz
Professor Scott Shenker
Professor John Chuang

Fall 2008

Improving Visibility of Distributed Systems through Execution Tracing

Rodrigo Lopes Cançado Fonseca

Abstract

Improving Visibility of Distributed Systems through Execution Tracing

by

Rodrigo Lopes Cançado Fonseca

Doctor of Philosophy in Computer Sciences

University of California, Berkeley

Professor Ion Stoica, Chair

We are faced today with distributed systems of unprecedented scale, built from hundreds of thousands of heterogeneous computers. These range from large datacenter installations to swarms of mobile devices embedded in and interacting with the physical world. They are pushing the limits of what problems we can solve, be it organizing all the world's information, making sense of the genome, or the world's climate. As our dependency on such systems increases, so does the importance of their availability, reliability, and efficiency, as well as the costs and impacts of failures. The problem, however, is that our ability to build and program these systems is progressing faster than our ability to understand how they work, and, especially, how they fail.

In this dissertation we argue that distributed systems should have *traceability*, or the ability to follow the execution of defined tasks or activities across the different components involved, as a first-class concept, creating the basis upon which to build tools to gain visibility into their execution and understand their behavior, performance, and failures.

To demonstrate that this is feasible and useful, we designed and implemented two instrumentation frameworks targeted at two widely different points in the distributed systems space. The first framework, X-Trace, tracks the execution and records the causal relationship among arbitrary programmer-defined events in large-scale, loosely-coupled distributed systems. X-Trace is general, lightweight, and is designed to span different layers, components, machines, and administrative domains. We instrumented several protocols and applications with X-Trace, including two wide-area deployed systems, the Coral CDN and the OASIS Anycast service. We used the instrumentation to find and solve several performance and correctness bugs. The second framework, Quanto, applies to wireless embedded sensor networks, and tracks execution of programmer-specified activities to understand energy and resource consumption. Quanto provides network-wide visibility of resource consumption for related events in extremely resource-constrained devices. We show how we instrumented TinyOS with Quanto and can use it to do a complete map of where and why energy is spent by a node and across the network.

Professor Ion Stoica
Dissertation Committee Chair

Dedicated to

Paula,

amor da minha vida.

Contents

List of Figures

List of Tables

Acknowledgments

Despite ultimately having one name as its author, this dissertation is, by any measure, the work of many who have helped, guided, encouraged, and stood by me all along the way. I can't expect to do justice to all in these few paragraphs, but hope to have left in these peoples' lives a small fraction of the mark they have left in mine.

I am extremely grateful to my research advisor and mentor, Ion Stoica, who took me under his wing early on. Ion provided constant encouragement and kept me focused, always asking sharp, challenging questions and helping me find the right research problems. He has so much passion for research, and I am lucky to have counted on his guidance, dedication, availability, and support.

I am also grateful to the other members of my dissertation committee. I have to thank Scott for his mentoring, for so many interesting discussions, and his impressive vision and insight into research. Randy's guidance, support, his eloquent comments, and his ability to put research questions in perspective have been instrumental in shaping my research and this dissertation. I also have to thank John Chuang kindly agreeing to be part of this committee, for his time and advice. I would like to thank all other professors with whom I interacted and who also provided me with guidance, advice, and collaboration, among which are David Culler, David Wagner, Eric Brewer, Anthony Joseph, Dave Patterson, Joe Hellerstein, Armando Fox, and Alan Smith.

I wouldn't have gotten to Berkeley were it not for the great education that I received at UFMG. I am ever so grateful to professor Virgilio Almeida, my undergraduate and Master's advisor and friend, who started me in research in my first semester as an undergrad, in this then 'new thing called the World Wide Web'. I would like to thank professors Wagner Meira Jr. and Nivio Ziviani, from UFMG, professor Mark Crovella, from Boston University, and professor Daniel Menascé,

from George Mason University for the advice, wisdom, and encouragement.

I was fortunate to do two very productive and gratifying internships during my studies. Sylvia Ratnasamy has been a great mentor, collaborator, and friend since the time I spent at Intel Research, in the Summer and Fall of 2003, when we worked on routing algorithms for Sensor Networks. I also have to thank Sujata Banerjee, Puneet Sharma, Sung-Ju Lee, and Sujoy Basu for a great and productive Summer in 2004, at HP Labs.

One of the most impressive aspects of my time in Berkeley was the quality of my fellow students, and how much I have learned from them. I remember my thinking as I got here, how could I possibly have ended up here among such great colleagues? My worries that I wasn't up to par were only alleviated with the realization that most of them felt the same way.

I would like to especially thank George Porter, for his friendship, for our great discussions, and for our so fruitful and productive collaboration in X-Trace. Prabal Dutta has been an amazing colleague and friend, and I want to thank him as well for our high-bandwidth collaboration that has resulted in the Quanto work. I keep learning so much that I may even get myself an oscilloscope now. I am sure much more will come from both pieces of work, and I am looking forward to it.

Phil Levis has been a constant source of inspiration and advice, and it has been a joy working with him over the years. Michael Freedman has been amazing, he was bold enough to let me use X-Trace to instrument his production systems, Coral and Oasis, for which I am truly indebted.

I am grateful to all my co-authors in so many collaborations through these years. It was a great joy working with Sukun Kim on Flush, and with Omprakash Gnawali, Kyle Jamieson, Phil Levis, at the TinyOS Networking Group, on CTP. Byung-Gon Chun has been a good friend and great collaborator in ideas, class projects, and papers, ever since we shared our first year office in Soda.

I must thank Rabin Patra and Sergiu Nedevschi for all the work together, the great friendship, and all of the fun times. I'd also like to thank Arsalan Tavakoli, Jorge Ortiz, Chris Baker, Daekyeong Moon, Jerry Zhao.

Joyojeet Pal has been a great collaborator, but above all, a great *bhai*. He is also much more eloquent than I am, and I have to steal his words in saying that his friendship is a greater gain than any degree I can take from here.

The times in Soda Hall were made so much more valuable because of all my fellow students. Thanks to Ana Ramirez Chang, Evan Chang, Blaine Nelson, Rusty Sears, Brighten Godfrey, Karthik Lakshminarayan, Lakshmi Subramanian, Dilip Joseph, Lucian Popa, Lisa Fowler, Jayanth Kannan, Gautam Altekar, Matthew Caesar, Peter Bodik, Jay Taneja, Stephen Dawson-Haggerty, Jaein Jong, Fred Jiang, and many others. Thanks to the TIER group and Eric Brewer for all the inspiring work that you guys do. I am also very grateful to Mike Howard, Jon Kuroda, and Keith Skowler for all the help.

I must also thank our dear friends with whom Paula and I shared so many unforgettable years here, Priscylle and Netinho, Flavio and Roberta, Peter, Claudia, Gabriel and Michael, Thelma and Michi, Alfredo and Zita. The Friday Brazilian lunches organized by Ram were such great relievers from the pressures of graduate school. Thanks to our so good friends Ram, Felipe, Mauricio, Ana Paula, Gregorio, Carol, Andres, Priscila and all others. I am grateful for all of our friends from back home, who are far but always so close, including Os Meninos, and my classmates from 2000email. Thanks for all who came to visit us in Berkeley and with whom we spent such great moments we will never forget.

It goes without saying that I have an unmeasurable gratitude towards my whole family. My

father, Franklin, and my mother, Beth, have no bounds for their love and dedication, and I simply have no words to thank them. Bernando, Kika, Lucas and Matheus, and Cristina, are the best siblings, friends, and beacons one can hope for. I also have to thank Sandra and Augusto, Fernanda and Claudão, Guilherme and Vitor for taking me as a son and as a brother, and showing the same love and support as they do for Paula. I love you all very much.

An finally, I can't hope to find words to express my gratitude for my wife, love of my life, Paula. It is no overstatement that none of this would be possible were it not for her relentless support, unconditional and unmeasurable love. We came together to Berkeley with not much more than each other, with big questions and great hopes, and we could not have dreamed of how well everything turned out. She has always been the brightest light by my side, in good and tough times, and I couldn't have been any more lucky to have her. Thank you, Linda, for all your sacrifices, support, help, love, encouragement, for putting up with the anxieties, long nights and crazy schedules, and, most of all, for simply being the most special person there is!

I have to thank God above all for having blessed me with the opportunity and privilege to have come to Berkeley, to have been surrounded by such an amazing set of people that continually inspires me to be a better person, and for having such a long list of so well deserved acknowledgments.

Bibliographic Notes

This dissertation is based on research I have performed between 2002 and 2008 with a number of outstanding colleagues. Part of the material on X-Trace in Chapters 3, 4, and 5 appears in a paper co-authored with George Porter, Randy Katz, Ion Stoica, and Scott Shenker [58], and some of this content also appears in George Porter's dissertation [134]. The work on tracking energy in embedded systems, in Chapters 7 and 8, is discussed on a paper on Quanto, co-authored with Prabal Dutta, Phillip Levis, and Ion Stoica [56].

Chapter 1

Introduction

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

We are faced today with distributed systems of unprecedented scale, built from hundreds of thousands of heterogeneous computers. These are pushing the limits of what problems we can solve, be it organizing all the world's information, making sense of the genome, or the world's climate. We are populating the world with wireless and mobile devices, connecting them, and embedding these networks into our own social networks and in the environment.

As our dependency on such systems increases, so does the importance of their availability, reliability, and efficiency, as well as the costs and impacts of failures. The problem, however, is that our ability to build and program these systems is progressing faster than our ability to understand how they work, and, especially, how they fail.

This dissertation looks at helping to close this gap, by providing frameworks and tools to give visibility into the end-to-end execution of these distributed systems. We aim to collect and pro-

duce relevant information to aid in checking correctness, troubleshooting, and analyzing and tuning performance and resource consumption.

In the remainder of this introductory chapter we present the motivation behind the dissertation, which can be summarized by the epigraph above, and how to improve on the situation. We then present our thesis, which argues that *traceability* should be a first-class concept in distributed systems. This is followed by a summary of our contributions, and by a description of the organization of the remaining chapters.

1.1 Motivation

Many of the distributed systems we deal with today are complex and heterogeneous. We increasingly depend on search engines, communication services, online auction and financial services, commerce applications, scientific computational grids, not to mention systems that run our telecommunication, health, transportation, and energy infrastructures.

A distributed system is a computational system composed of multiple independent computers, connected by a network and communicating by exchanging messages [43]. This places distributed systems in contrast with centralized systems, and also with parallel systems, that can have multiple tightly-coupled processing units connected by reliable, high bandwidth channels.

Our current distributed systems are *scaling up* in one extreme, in terms of the size and aggregated processing power of large-scale datacenters, and *scaling out* in the other, in terms the large number of mobile and embedded devices we are deploying. Two examples of the former are the computational Grid that leverages resources from scientific institutions around the world [60], and the massive distributed infrastructure of Internet services like Google [69, 29], Yahoo! [36], Ama-

zon [41], and Microsoft [85]. These companies are building large datacenter installations reaching almost 100,000 servers each [107]. A recent report by the United States Environmental Protection Agency [3] noted that the number of servers installed in the US was expected to almost triple from 2000 to 2010, reaching 15,775,000, with a compounded annual growth rate of 11%. Two examples of the growing use of mobile devices are proposals to use large numbers of GPS-equipped cell phones as wireless networked sensors for traffic [49], and for air pollution [80].

There are a number of reasons for distributing computation in this fashion. Large systems of loosely-coupled independent components are the most economically feasible way to cope with the scale of our data and processing demands. Systems are also distributed geographically to provide better latency, resiliency to catastrophic events, and for economic and political reasons. For example, large services like Google can move large batch jobs around the world to track off-peak electricity costs. Some countries, on the other hand, may require that data for services offered in the country be maintained in datacenters located domestically.

Users themselves are inherently 'distributed', even more so with increasingly smart mobile client devices. These form an integral part of complex systems, exemplified by peer-to-peer networks [104], voice over IP, distributed computational projects [9], and by current terms such as 'cloud computing' [7, 70] and 'software as a service' [145].

Lastly, embedded wireless devices in the form of cell phones, wireless sensor networks [101], and RFID tags promise to extend the reach of distributed systems to the environment around us, increasing the numbers of devices by orders of magnitude.

The systems in which we are interested in this dissertation are well characterized by Leslie Lamport in his quote in the epigraph [98]. They have several characteristics that make their program-

ming, deployment, troubleshooting, and operation harder than in their centralized or tightly-coupled counterparts. The individual components operate concurrently, can fail independently of each other, and do not share a common clock. Furthermore, they are composed of a large number of possibly heterogeneous components, and the set of components can vary over time. The communication links among the components are not necessarily reliable, and can also fail independently. Lastly, components may be under different administrative control and physically separated, making it hard or impossible for a single administrator to reach all of them.

These factors contribute to limit the visibility into the execution in these systems. When there is a failure in an operation, finding the cause may be a difficult task for the developer, user, or operator. Many failures will have the same symptoms, and even knowing in which components to look for faults is difficult. For example, a large number of faults in a Web server are presented to the user as an Internal Server Error [53]. The user will have a hard time explaining to the server administrator what happened, and the administrator will have to sift through logs at one of potentially many servers to find the user's session. If the system has several tiers, such as application, cache, and database servers, finding which one was the culprit can again require iteratively looking at logs, if it is at all possible. Many tools, like *traceroute*, do not cross network layer boundaries, thus only providing visibility up to the first application hop. Other tools offer health monitoring of individual machines or processes, and fail to get an end-to-end picture of specific tasks.

We address this problem by following the execution of tasks across the different components of distributed systems, gathering data that correlates the individual operations that happen at each component. Our design strives to allow this tracking to cross boundaries of system and application components, network layers, and administrative domains.

To make our presentation and our contributions concrete, we look at two specific classes of such systems: large scale Internet-based systems on the one hand, and embedded networked systems such as sensor networks, on the other.

1.2 Thesis

In this dissertation we argue that *distributed systems should have traceability as a first-class concept, creating the basis upon which to build tools to gain visibility into their execution and understand their behavior, performance, and failures.*

By traceability we mean being able to follow the execution of defined tasks or activities in a distributed system across the different components involved, and by first-class we mean that the components can be aware of the task or activity they are part of, and can participate in the instrumentation in a consistent way.

For this to be practical, given the fact that most systems have heterogeneous components that are developed, deployed, and administered by independent parties, the traceability must span boundaries of application and system components, network layers and protocols, and administrative domains. Also, given the broad class of systems we are targeting, there must be enough flexibility to allow different questions to be answered, depending on the system instrumented and on the person asking the question.

To demonstrate that this is feasible and useful, we designed and implemented two instrumentation frameworks that share the same basic principles, targeted at two widely different points in the distributed systems space. The underlying principle is to track the execution of high level tasks or activities that are meaningful to a programmer, user, or operator of the system, and collect in-

formation about the execution. We do this by adding metadata to messages, data, and runtime environments, and have this metadata propagate with the execution flow.

The first framework, X-Trace, tracks the execution and records the causal relationship among arbitrary programmer-defined events in large-scale loosely-coupled distributed systems. The challenges that X-Trace addresses are how to allow tracing of heterogeneous systems spanning different layers, machines, and administrative domains in a scalable and decentralized way. The second, Quanto, applies to embedded wireless sensor networks, and tracks execution of programmer-specified activities to understand energy and resource consumption. The challenges Quanto addresses are how to provide network-wide visibility of resource consumption for related events in extremely resource-constrained devices.

Next we summarize the main contributions from our work.

1.3 Contributions

Our contributions are centered around the two frameworks for instrumenting, respectively, large-scale loosely-coupled distributed system, and embedded networked systems. The two environments pose different requirements and constraints, and these shape the specific information we collect and the questions we can answer in either case. We argue that by providing simple execution tracking as a base we can provide targeted visibility into distributed systems and build useful applications for gaining understanding, isolating faults, and fixing correctness and performance problems.

The first framework, X-Trace, has the following contributions:

1. **Scalable Causality Tracing** X-Trace captures the happens-before relation [99] among arbitrary events using constant-sized metadata and no central coordination during runtime.

2. **Pervasive Tracing** X-Trace has a simple and uniform execution model that captures causality across different:

   - application and system components,
   - software and network layers,
   - machines,
   - administrative domains.

   X-Trace makes few assumptions about the instrumented system, presents low overhead, gives control of trace data to the instrumented system. It is also independently and incrementally deployable, as we defined metadata and report formats, and application programming interface such that others can instrument their own systems and integrate the instrumentation with already instrumented systems.

3. **Programming-model Agnostic Integration** X-Trace defines an API for instrumentation of systems that is uniform across thread- and event-based programming models, and is as simple as commong logging APIs in the common case.

We demonstrate X-Trace's feasibility and usefulness by integrating it into a number of applications and protocols, by building a data collection and analysis backend, and using it to aid development, performance analysis and tuning, and finding correctness and performance bugs in running systems. We integrated X-Trace, among others, into HTTP, DNS, Java, C++, `libasync`, Coral, Oasis, Chord, and I3. Notably, my colleagues also have integrated X-Trace into an 802.1X authentication system [134] and into the Hadoop Map-Reduce system [177].

The second framework, Quanto, applies tracing to embedded network systems, specifically sen-

sor network nodes running the TinyOS operating system [78]. While exhibiting common characteristics of other distributed systems such as independent failure of multiple components and unreliable and relatively expensive communication, the defining characteristic here is the extreme resource constraints in terms of memory, processing power, bandwidth, and, primarily, energy.

The contributions of Quanto are the following:

1. **Low overhead causality tracking** A very low-overhead framework to track the execution of distributed activities in embedded networked systems across software layers, hardware components and network nodes.

2. **Detailed and scalable energy breakdown** Quanto combines OS instrumentation of devices' power state with scalable, aggregate energy measurement at each node to provide a runtime breakdown of energy usage per energy sink.

3. **Integrated ernergy profiling** Quanto integrates the energy breakdown with distributed activities to provide profiles of time and energy usage by activity at the granularity of each peripheral of a node. The approach is also extensible to other performance counters.

We demonstrate Quanto's feasibility and usefulness by instrumenting the TinyOS operating system and producing time and energy profiles of simple applications. These profiles tell where and why energy and time were spent on behalf of high-level activities that are meaningful to programmers.

1.4 Organization

The dissertation is organized around the X-Trace and Quanto frameworks, and how they leverage distributed execution tracking, respectively, in large scale distributed systems and in embedded networked systems. The following paragraphs describe each of the remaining chapters in turn.

Chapter 2 provides a background for the problem of improving the visibility in distributed systems, and different techniques that have been developed in the literature and in industry for debugging, profiling, and tracking execution in such systems. The chapter describes general techniques, and then describes challenges and previous solutions specific to distributed systems.

Chapter 3 introduces X-Trace. We first describe its requirements, specifically how it is intended to capture the deterministic causality between arbitrary events in the execution of a distributed system with minimal impact on the execution, and generate data that should survive failures in the execution. We then describe X-Trace's design and architecture in light of the requirements, including its formal model of the distributed execution of tasks, which captures the full happens-before relation, and how it implements this model by propagating metadata with the execution. We discuss how the design meets the requirements, and also some of the limitations of X-Trace.

Chapter 4 describes the implementation of X-Trace, including details of X-Trace's metadata, its propagation, and the collection of information. X-Trace requires modification to existing systems, and we discuss how to integrate X-Trace into application and system components. X-Trace provides a runtime library that greatly eases the burden of instrumentation. The library automates metadata propagation with a unified interface for both thread- and event-based programming models, and makes recording X-Trace information as easy as issuing a log statement. The output of X-Trace is a richly annotated causality graph representing the execution, and the chapter ends by describing how

to process and visualize this output.

Chapters 5 and **6** provide a practical evaluation of X-Trace. In Chapter 5 we integrate X-Trace into three test scenarios that illustrate important X-Trace characteristics. The first consists of a web client, DNS infrastructure, and a simple web server, and the second consists of database-backed web server. These two scenarios illustrate integration with recursive applications that use different existing protocols, and how we can find the root cause of some failures. The third scenario comprises an overlay network, and exercises tracing across multiple layers. We provide benchmarks showing that X-Trace can have manageable overhead.

Chapter **6** has our main X-Trace results, and describes the use of X-Trace in two real-world production systems, the Coral Content Distribution Network [63], and Oasis [64]. We integrated X-Trace into these two systems and into their common runtime system based on the asynchronous execution library `libasync` [111]. We show situations in which X-Trace can distinguish the causes of different problems with the same symptoms, such as a cross-machine breakdown of Coral's response time. Despite being in production for over three years, we also found correctness and performance bugs in both systems, and describe two cases in detail.

Chapters 7 and **8** move to embedded networking systems, specifically sensor networks. Chapter 7 presents the design, implementation, and evaluation of Quanto. Quanto is a distributed energy profiler that attributes energy usage in several peripherals of a node to high level activities. Quanto tracks activities much like X-Trace tracks tasks, but does not record traces of events. Instead, it uses activities as *principals* to which resource usage is attributed. We describe the implementation and integration into the TinyOS operating system, combined with a lightweight and precise energy monitoring hardware.

In Chapter 8 we use the data gathered from simple applications to disambiguate the energy usage per activity and per hardware peripheral, and also produce detailed visualizations of the breakdown. In our evaluation also show how we can do profiling of activities that span nodes, and present three case studies of detailed node behavior we uncovered with Quanto.

Finally, **Chapter 9** presents a summary of our contributions and results, along with a discussion of exciting research directions that this work opens up.

We now move on to discuss important concepts and background for improving visibility into distributed systems, exploring the design space and previous related work.

Chapter 2

Background and Related Work

In this dissertation we develop and describe two frameworks that improve the visibility of running distributed systems. The first, X-Trace, focuses on large-scale, loosely-coupled and heterogeneous distributed systems, and captures the causal relations among events in the execution. The output of X-Trace can be used for example for debugging, troubleshooting, verifying assumptions, performance tuning, and visualization. The second framework, Quanto, focuses on wireless embedded sensor networks, and has as its main goal profiling resource usage of distributed activities. The same technique underlies both frameworks, which is being able to follow the distributed execution of a running system.

Our work builds upon and borrows from several techniques that have been developed for providing visibility into centralized, parallel, and distributed systems. We are primarily interested in techniques that record information about specific executions of distributed systems, spanning multiple levels of granularity and multiple nodes.

In this chapter we examine part of the design space for these techniques, putting some of the

previous work into context. We leave more specific work related to Quanto in terms of profiling energy to Chapter 7. Section 2.1 begins by describing common techniques to provide visibility into systems in general, namely formal methods, monitoring, logging, tracing, and profiling. We look at different issues that influence these techniques, including granularity of instrumentation and data aggregation, how to instrument a system, and how to capture concurrency. Section 2.2 looks specifically at distributed systems, and at previous works that have improved the visibility in the domain. Section 2.3 briefly looks at some related techniques from the field of computer security. Lastly, Section 2.4 explores important tradeoffs in this space and presents an informal taxonomy of previous work related to X-Trace and Quanto.

2.1 Providing Visibility

There is a wide range of techniques that can be said to improve visibility into a system, including profiling, tracing, monitoring, and debugging. They can operate at different granularities, from machine instructions to requests received by a multi-tier web service, and with different scopes, from a single program in a single-processor machine, to, say, a large set of loosely-coupled federated systems running a distributed physics simulation on the Grid [60]. Given this diversity, the choice of techniques depends on a number of factors. It depends on who is using them, whether a developer, an integrator, an operator, or an end user. It depends on the objective, which can be identifying faults, verifying correctness, prioritizing optimization targets, collecting data for simulations, auditing, identifying dependencies, visualizing the execution, monitoring resource usage, among many other uses. Finally, the choice also depends on the extent to which the system under observation can be instrumented, if at all; on the amount of information that can be collected; and on the overhead that

can be tolerated in the running system.

2.1.1 Techniques

Formal Methods

Our focus in this dissertation is on techniques that directly collect data about executions of live systems. Our runtime techniques are complementary to static methods of verification of software correctness, such as Model Checking [34], or Symbolic Execution [91]. Model Checking systematically explores the state space of all possible executions, and uses temporal logic to check for violation of liveness and safety properties. It has enjoyed great success in proving the correctness of hardware designs [112], network protocols [68], and even partially checking the Linux TCP implementation [119]. However, in large distributed systems in which multiple components can fail independently and in which communication latencies can often violate design-time assumptions, it is sometimes important to record and examine faulty executions, or to understand the cause of unexpected behavior in production systems. Furthermore, for performance tuning and bottleneck identification, one needs to observe systems running at scale, as different bottlenecks may appear as the system and workload scale to production levels.

Monitoring

A number of tools focuses on monitoring network status, aggregating data from many devices and layers. Ganglia [108] collects time series of system data from a set of nodes, allowing operators to identify hotspots, and high-level bottlenecks such as disk, CPU, or network. Nagios [121] is an open source monitoring tool, used to collect health data from hosts, services, network components.

Nagios periodically checks the status of monitored entities and maintains a global view of the system. SNMP [26] is a protocol that lets operators inspect instrumentation data from network devices such as packet counts and error conditions. HP Openview is an example of an enterprise-wide network management tool that makes use of SNMP data. Openview can coordinate views at different granularities, as well as coordinate network policy changes. Cisco Systems' Netflows [122] also provides device instrumentation, although at a finer granularity than SNMP. These tools take an orthogonal view to our view of following the execution, for they measure the effect of continued execution on individual components. Kompella et al. [93] present a service for collecting "cross-layer information" in a network. Their focus of is on collecting control path state at different layers and identifying dependencies. Using the information their system collects, one could identify how failures in one layer impact other layers. Traceroute is a popular tool to identify paths in an IP network. It sends a set of its own probes to determine the sequence of nodes in a path. This is also different from our approach in that we do not use separate probes to collect data; rather, we collect data on the actual application execution.

Logging

Perhaps the simplest technique to provide visibility into a running system is the use of *print* statements to a console or file. This can range from ad-hoc and unstructured one-time messages to more efficient and disciplined *logging* infrastructures such as syslog [103], log4j [102] and its many derivatives. Logging frameworks are designed to selectively enable logging statements of a running program, record the output to a variety of destinations, and to minimize the cost of disabled statements. Statements are output in linear, temporal order, and fail to represent the actual causality when there are more than one concurrent thread of execution. For the same reason it is also difficult

to correlate logs from different machines without resorting to special identifiers in the logs, especially if the clocks are not synchronized. Despite their ad-hoc nature, logs are pervasive throughout systems and can be used for troubleshooting and debugging. Splunk [150] is a commercial system that aggregates logs from different sources in a network, and uses information retrieval techniques to index and provide interactive searches on multiple logs.

Tracing

Tracing can be viewed as a more structured form of logging that records sequences of executions of basic blocks of a program. Tracing is extremely valuable for correctness and performance debugging, trace-driven simulation, creating and validating models and tuning parameters, performance tuning, auditing and forensics, among other uses. It can be done at different levels of granularity, such as machine instructions, system calls (e.g. `strace`, `ktrace`, `truss`), file-system activity [11], network packets [19, 83, 161], or function calls. Some systems, like liblog [67] and Flashback [152] collect traces with enough information to replay the execution of a program, allowing for example the inspection of the replayed instance with a symbolic debugger such as `gdb`.

The line between tracing and logging is not crisply defined. For our purposes, we consider a trace to be a sequence of events that is well structured and comprehensive, meaning that it captures, for some period of time, all relevant events, allowing the reconstruction of the sequence of actions at the desired level of granularity. In this sense, a trace can be a subset of a log. For example, we can define a trace to be the sequence of all requests made to a web server, which can be extracted from the server logs. The logs can, however, contain additional information that is not part of the trace, such as initialization or error information. Tracing can generate a large amount of data. Depending on the application, this data may not all be needed, or may impose too much of an overhead. In

these cases, profiling can be an alternative.

Profiling

Profiling is a set of related techniques that aggregate information about resource usage instead of producing sequences of events as tracing does. Profiling is useful for performance tuning, as it can determine, for instance, which components of a system are using the most time. Understanding which activities of a program consume the most resources (including time), allows one to prioritize optimization efforts [76]. `gprof` [72] is a commonly used program profiler that uses compile-time instrumentation of function calls to present how much time was spent on each function call path. Anderson et al. [10] describe DCPI, an efficient, system-wide profiler that uses high-frequency sampling of processor counters to collect performance data about executables in a multi-processor system, down to the level of pipeline stalls by individual instructions, while maintaining an overhead between 1 and 3%. As with tracing, profilers operate on different granularities, and with different scopes. Profiling can also aggregate resources other than time spent [74], such as energy. ECOSystem[180], and Quanto, which we describe in Chapter 7, are two example systems that treat energy as a first-class resource.

2.1.2 Granularity

Both profiling and tracing rely on a set of events triggered during execution, with tracing registering the events for later reconstruction, and profiling aggregating the information. One question that arises is at what granularity to capture the events. The answer depends on the question we are trying to answer with the instrumentation, the extent to which we can instrument the system, and the amount of information we want or need. It is easier, for instance, to monitor system calls made

by a process than to trace the function calls inside of an operating system kernel.

Capturing events at too fine a granularity, such as machine instructions, may be useful for understanding the performance of a processor cache, but will constitute too much detail and too large a volume of information if one wants to capture requests made to a file system. Because of the number of events, also, this may impose too high an overhead both for capturing and processing the information. On the other hand, capturing events at too coarse a granularity may help identify a problem, for example a slow response time, but will generally not have the necessary detail to understand the cause.

One way to have fine granularity and still bound the amount of information and processing is to use sampling techniques (e.g. [10, 72]), which will generate approximations of the actual resource usage in the case of profiling. In [114], the authors mention that in a first incarnation of the IPS system, instrumentation at then level of procedures was done with periodic sampling, to make the overhead manageable. In the second version of the system they were able to use exact counting, due to improvements in the instrumentation efficiency.

Another way to have fine granularity and yet control the amount of information and processing is to use dynamic instrumentation. DTrace [25] is a recent framework by Sun Microsystems for dynamic instrumentation of systems running in a single machine. It allows very detailed runtime addition of instrumentation to several points in the system (called probes), such as function call boundaries and system calls, both in user processes and inside the kernel, and has zero cost for non-active probes. D-Trace tackles the granularity problem by allowing user scripts to dynamically activate specific probes and execute arbitrary read-only actions at these points. Actions include accumulating profile information, recording traces, or inspecting internal state for debug-

ging purposes. This way it combines very detailed information with manageable amounts of data and overhead only where needed. Paradyn [113] is a comprehensive tool for performance analysis of large scale parallel programs which relies heavily on dynamic instrumentation to provide detailed performance information while keeping the cost of instrumentation under control.

X-Trace allows capturing events at different granularities by defining a uniform event model that can span different levels.

2.1.3 Aggregation

A related question is how to aggregate the events. Aggregation allows a system to create hierarchical abstraction levels of events, summarize the information, do filtering, and accounting. Common ways of aggregating events include processes and threads in the case of operating systems, requests, in the case of web services, or flows, in the case of network routers. IPS-2 [114], a performance measurement system for parallel and distributed programs, had a hierarchical model for collecting data, from basic events, to procedures, to processes, to different machines, to whole distributed programs, allowing users to explore the information grouped at these different levels.

Operating systems perform resource accounting, for scheduling decisions, for example, based on processes and threads. Banga et al. [18] use the term *resource principals* for the entities to which resource usage should be attributed. OSs normally combine the concepts of protection domain (processes) with that of resource principals. This is less than ideal in many situations, high performance Internet servers being one example [18]. Common patterns to build these systems [125] can have one thread serve several independent requests over time, in the case of pooled-thread servers; one thread per core multiplexing several requests at once, in event-based systems; or even several threads working on behalf of one request, if there is parallelism. In all these cases the resource principals

should be the requests, and not the threads. In the first two cases, a single thread would be spending resources on behalf of different principals over time, while in the last case several threads would be doing so on behalf of the same principal.

As systems get larger and more distributed, the notion of a process again fails as an entity for accounting. Modern Internet services are composed of thousands of computers [29, 69, 41, 36], and a single user request might cause processes on many machines to perform some work on its behalf. On the other extreme, embedded networked devices may have operating systems that even lack the notion of threads or processes [78], but still perform coordinated actions on behalf of different activities.

Both X-Trace and Quanto allow a flexible definition of resource principals that are meaningful to the system programmer or operator, independent of other entities in the system like module, functions, or processes. We borrow from the Rialto operating system [88] the concept of *activities* as the abstraction to which resources are allocated and charged. X-Trace groups events by *tasks*, while Quanto does accounting of resource usage to *activities*. Within an X-Trace task, events may retain a hierarchical structure that allows further filtering and summarization.

2.1.4 Instrumentation

Instrumentation for capturing the relevant events for either tracing or profiling depends on the granularity desired. It can be done by the programmer by adding special calls to an application [13, 139], automatically by the compiler [72], by the runtime system [31], at runtime, by dynamic library [95] or system call interposition [67], by binary transformation and instrumentation [15, 79]. Events can also be captured passively, without any changes to the running system, if they are externally visible, for example by monitoring network messages [4, 14]. Which points of the program

are instrumented determines the completeness of the obtained data. In the case of instrumenting program blocks, like function calls, Ball and Larus [17] describe algorithms to insert an optimally minimal set of instrumentation points while unambiguously capturing the control flow of a program. As we discuss in Section 2.4 below, a combination of the instrumentation with assumptions about the instrumented system determines what information can be reconstructed and with what certainty.

D-Trace and Paradyn [113] use dynamic instrumentation by binary transformation as a way of balancing the cost of instrumentation and the level of detail required for an observation. Causeway [28] and SDI [138] provide mechanisms for automating metadata propagation within operating system and application structures, and can facilitate the building of tracing and profiling applications. In both X-Trace and Quanto we instrumented runtime systems, libraries, and components to automatically perform tracing, and expose to the programmer APIs for adding application-specific events to the traces.

2.1.5 Concurrency

Several works have focused on the additional challenges introduced by concurrency and parallelism. One fundamental problem is to identify which events precede, or can influence, other events. Using wallclock time is insufficient. Even if clocks on different machines are perfectly synchronized (which is commonly not the case), the time of occurrence imposes a total ordering that does not necessarily express dependency.

Lamport's seminal paper [99] introduced the happens-before relation, and formalized the notions of concurrent and sequential computation. Vector Clocks [52, 109] are an approach to capture the happens-before relation using timestamps for events in a distributed computation. One problem with Vector Clocks is that they require the size of the timestamps to be as large as the number of

processes in the computation, which can be quite large for some cases. There are several approaches to reduce the size of the vector clocks, but they still incur some sizeable overhead when the number of processes is large, and are hard to use when the individual processes are not known a priori. Chain Clocks [2] is a technique that reduces the size of timestamps, and in Chapter 4 we present an algorithm to derive chain clocks from the X-Trace output. Ward [171] presents a set of dimensions to characterize logical timestamp systems that capture deterministic causality among events.

The happens-before relation imposes a partial order among events in a system, and several algorithms and tools can use the partial order to identify common patterns (e.g. race conditions), global predicates, execution differences, performance analysis (e.g., Critical Path analysis [19, 114]), among others. Again we refer the reader to [171] for a survey of these techniques.

X-Trace captures the happens-before relation of events within a task by propagating constant-sized metadata along with the execution and embedded in messages among components.

2.2 Visibility in Distributed Systems

The distributed systems we consider here have important differences to many parallel systems studied previously [90, 113]. These generally comprised a single program running on a fairly homogeneous parallel machine or cluster. We focus on distributed systems comprised of many different applications running on much more heterogeneous sets of machines. It may be hard to instrument all components, and in some cases to even know which components are involved in an execution. Communication latencies are much higher and more unpredictable, and components can fail independently.

Given the difficulty of instrumenting some systems, Aguilera et al., in Project5 [4], and Reynolds

et al., in Wap5 [140], find anomalous behavior in distributed systems by treating each component as a black box, and inferring the operation paths by only looking at network messages like remote procedure calls and HTTP transactions. They passively trace all internode messages, and then use heuristics to find causality between messages, by looking at the timings of successive messages. The main goal is to find performance problems and be able to identify nodes which are more likely to be causing the problems. The technique is vulnerable to excessive parallelism, clock skew among machines, and applications that delay or reorder outputs in a non-obvious way.

Sherlock [14], and more recently, Orion [32] use correlation in the timings of messages to infer dependencies among services in large enterprise networks. Sherlock creates a graph of probabilistic dependencies that guides inference algorithms to find the cause of detected performance problems.

BorderPatrol [95] increases the precision of the information is by incorporating knowledge and assumptions about the traced system. It produces causal traces of requests through systems such as three-tiered web services with much more precision than in pure inference systems like the ones mentioned above. It does this by adding knowledge about the *protocols* that the black boxes implement, and assumptions about the behavior of applications. It assumes that applications follow the protocol correctly, that outputs immediately follow the causing inputs, and that individual requests are internally independent. While BorderPatrol demonstrates that these assumptions are valid for a number of applications, the approach is not completely general, and still relies on inference in some cases. Some applications violate the assumptions and may reorder, batch, and treat individual requests in a way that requires application knowledge to disambiguate the causal relationships.

Magpie [20] is another framework that adds information about the observed system in order to avoid inference. Magpie doesn't modify the applications. It produces exact traces of requests

through a distributed system by using extensive logging in the involved components and a detailed schema that relates specific identifiers in the traces. The approach requires expert knowledge about all components involved, which may not be feasible, for example, for system integrators that combine components from different vendors. Also, because there is no a priori way of classifying logged statements, Magpie cannot sample logs from different components in a correlated way. Like X-Trace, they correlate lower level events with a higher level task, but focus mostly on a single system or on distributed systems that are highly instrumented in a compatible way.

Another common approach to increase precision and avoid the need to do inference is to *add* metadata to the execution. By using well defined metadata that leaves "breadcrumbs" as the execution proceeds, several frameworks [13, 28, 31, 138, 139, 162] can precisely reconstruct causality in the execution paths. This is also the approach we take in X-Trace and Quanto.

The Application Response Measurement (ARM) [13] project annotates transactional protocols in corporate enterprises with identifiers. Devices in that system record start and end times for transactions, which can be reconciled offline. ARM targets the application layer, and its focus is to diagnose performance problems in nested transactions.

Pinpoint [31] aims at detecting faults in large distributed systems. The authors modified J2EE middleware to capture the paths that component-based Java systems took through that middleware. They use anomaly detection techniques on the traces to infer which components are responsible for causing faults. Their data mining techniques can be used on X-Trace traces, provided that the concept of faulty executions be well defined for a set of traces.

Pip [139] is an infrastructure for comparing actual and expected behavior of distributed systems by reasoning about paths through the application. They record paths by propagating path identifiers

between components, and can specify recognizers for paths that deal with system communication structure, timing, resource consumption. Pip is targeted at a single distributed application, under the same administrative domain, and works at the application layer only. X-Trace is complementary to Pip in this sense. We believe that some of Pip's analysis can also be performed on X-Trace's task graphs.

Stardust [162] is a system for tracking activity in distributed systems, with a focus on storage systems. It tracks activities by propagating "breadcrumbs" with the execution, and recording activity records at strategic points. A user can then issue queries about resource usage and latency of operations to the database that stores the activity records. The breadcrumbs in Stardust are similar to X-Trace's tasks and Quanto's activities.

Whodunit [27] is a distributed profiler, developed simultaneously with X-Trace, that provides transactional profiling of multi-tier applications by tracking transactions through shared memory, events, stages [173], and interprocess communications. They extend call graph profiling [74] to distributed applications by propagating a transaction context with the execution. Their propagation is similar to what we use for X-Trace and Quanto, but the nature of the information is different. While they propagate cumulative call paths, X-Trace propagates constant-sized metadata that allows capturing of complete execution partial orders. In that sense, the execution model captured by X-Trace is more general. Quanto also does profiling, but uses flat activity identifiers as the aggregation entities, due to the resource constraints in wireless sensor networks.

Event Tracing for Windows (ETW) [127] is an API that shipped with Windows Vista that allows efficient logging of events with OS support. ETW has a flexible event model and support for end-to-end tracing by means of unique request identifiers. It is not clear, however, how to integrate

applications instrumented with ETW with other components running in other operating systems.

2.3 Other Related Techniques

The techniques we use for following computation have parallels in the field of security. Dynamic Taint Analysis [123] keeps track of data from untrusted sources – *tainted data* as it enters a program, such that when the data itself, or some data arithmetically derived from the data, is used in an unsafe way, some action can be taken. Perl offers a similar feature in its so called taint mode, which treats values from incoming untrusted sources as tainted.

Distributed information flow control [96, 120, 178] is an approach to security that allows tracking of data as it flows through a system and gives applications fine-grained, explicit control of security policies. DIFC works by propagating labels with data, and assigning those labels to components dealing with the data. DStar [179] extends DIFC to work across node boundaries, by encoding and propagating labels with messages among nodes. There is some commonality in the way DIFC systems track data flow and the way we track execution of tasks in X-Trace, and activities in Quanto, and we plan to explore this connection in the future.

2.4 Tradeoffs

In the design space of distributed visibility tools, there are certain tradeoffs that we informally discuss in light of existing systems.

2.4.1 Exact versus Approximate Visibility

The first tradeoff relates the amount of knowledge about the observed system, the amount of recorded information, and the certainty about causality and dependencies that a tool can uncover.

Many systems provide approximate visibility due to infeasibility of directly instrumenting some components. Project 5 and Wap 5 don't modify the observed systems, but can only find correlations with some uncertainty. Likewise, Sherlock and Orion only look at network traffic, and try to infer dependencies among services. These systems have to deal with false positives and negatives in their findings. BorderPatrol adds knowledge about protocols and assumptions about application behavior, and substantially increases the precision of the causality between messages. When their assumptions are violated, however, they still have to resort to heuristics and inference. Magpie [20] resorts to extensive logging from applications and expert knowledge to combine information from the logs, and is able to uncover exact causality.

Other systems achieve precision by adding metadata to the execution, such as ARM, Pinpoint, Pip, Stardust, as well as X-Trace and Quanto.

2.4.2 In-path Metadata versus Logged Information

Another dimension in the design space refers to the amount of information that an instrumentation framework adds to the datapath, which we call metadata, versus the amount of information that is logged by each component as they execute. If we constrain ourselves to solutions that produce deterministic results, there is a tradeoff here too. On the one extreme, we have applications that add all information to the datapath as the execution progresses, and log no intermediate information. Conversely, there are systems that log extensively and carry no extra information on the path.

Take for example a multihop routing protocol, like BVR [59], and let us ask two questions about a given path: how many hops were traversed, and which were these hops. One way to answer the first question is to have a counter in the packet header, and have each router along the way increment the counter. In the last hop we will have the answer, and no information was externally logged along the way by the routers. Another way to answer the question is for each router to log the fact that it has forwarded a packet with id X, and then later find the routers which logged the given packet. The first option is more attractive for many reasons, except for the fact that the logged information in this case shares fate with the packet: if we don't receive the packet, we will not know where it stopped. The same two options exist for the second question, and in fact the IPv4 option Record Route [135] does just that. An additional problem with the approach that adds information to the packet is that in this case the information grows with each hop, and we may have to limit the amount recorded.

We can also combine the two options. Without any information in the packet, we would need to log all packets at all routers to know about a route later, if using the second option. We can add a single bit to a packet stating whether we should log it, and as long as the bit is propagated, routers would only need to log packets for interesting flows.

Looking at existing systems, Magpie again sits in one extreme, adding no information to the datapath, but requiring very detailed logs at all components to allow for later correlations, which they call temporal joins. X-Trace, on the other hand, adds constant-sized metadata to messages and propagates it with the execution, and requires intermediate nodes to record bindings between successive events when changing the metadata.

Frameworks based on Vector Clocks [52, 109] transmit potentially large timestamps with messages (vector clocks), to be able to compare any two events in the system in constant time, deter-

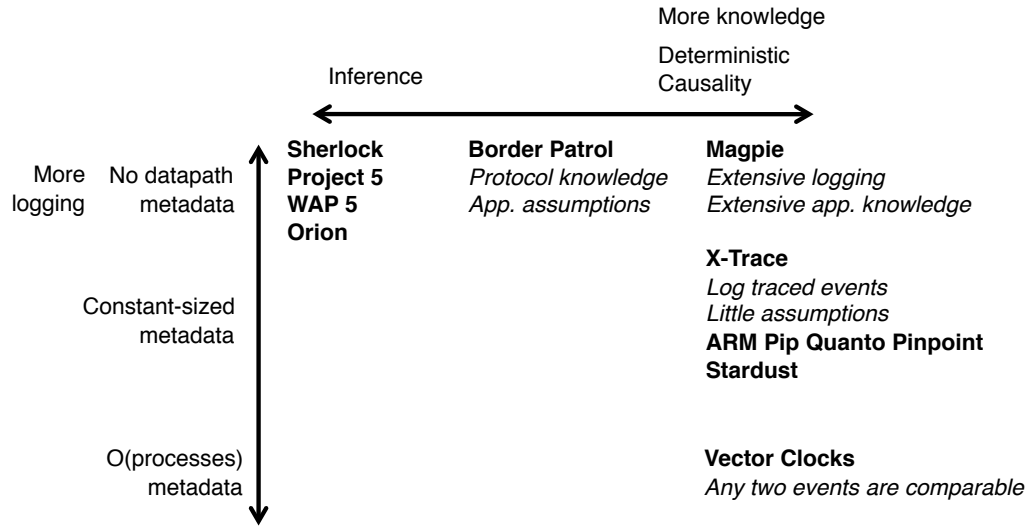


Figure 2.1: Tradeoffs in the design space of distributed visibility tools. Both X-Trace and Quanto keep track of deterministic causality, and carry constant-sized metadata with the execution.

mining whether they are concurrent or sequential. In the original version the vectors are as large as the number of processes, and even though there are approaches to reduce the size of the vectors, they still have to be as large as the *width* of the partial order of events (*cf.* Section 4.4.2).

X-Trace transmits less information as metadata, but with post-processing we can obtain Chain Clock timestamps [2] that have the same expressive power as Vector Clocks. The tradeoff here is that any two events with vector clock timestamps can be directly compared for precedence, while the precedence of two X-Trace events can only be determined if all events between the two are also examined.

Figure 2.1 shows some of these works arranged in two axes according to these two tradeoffs. The horizontal axis represents the tradeoff between exactness and a priori knowledge about the application and protocols, while the vertical axis represents the tradeoff between the amount of

information carried in the datapath versus the amount that needs to be logged for reconstructing the deterministic causality.

2.4.3 Intrusiveness versus Scope

Finally, another dimension of the design space refers to what components are modified with instrumentation, versus the extent to which we can reconstruct the causality among events.

Many systems refrain from modifying the topmost level of the software stack, the application, for these are the most diverse components, and the ones developed by the largest number of independent developers. In the general case, when a layer in the software stack is instrumented, events in that layer and in layers below can be correlated, but it is not certain that events in layers above it can. BorderPatrol [95] doesn't instrument applications and uses assumptions about internal application structure to causally relate these messages. It fails to disambiguate causality when these assumptions are violated.

Dapper is a framework widely used at Google to monitor the performance of the RPC layer of distributed applications. Details of how it works are not yet widely available [154]. Dapper adds identifiers to RPC requests and responses, and is able to follow a distributed execution across multiple components. It has a transparent mode that requires no changes in applications, but may fail to disambiguate precise causality at the application layer. For these cases, it offers a set of APIs so that application developers can add instrumentation to capture precise causality.

If end-to-end tracing is the objective, under some circumstances, only keeping track of causality at the application layer can unambiguously track causality. We can cast this as a version of the end-to-end argument [146], stating that tracing at lower levels of the system is not always sufficient to determine causality at a higher level. *Both X-Trace and Quanto allow instrumentation at the*

application layer for correlating lower layer events, in the general case.

2.5 Summary

To conclude this chapter, we summarize how X-Trace and Quanto compare against some of the related work, in light of the tradeoffs we discussed. In short, in both cases, we choose exact visibility, minimal in-path metadata, and allow modifications of systems to achieve end-to-end tracing.

As we describe in detail in Chapter 3, X-Trace's requirements (*cf.* Section 3.1.2) dictate our design choices. The requirement that we deterministically record the causality between events differentiates us from projects like Project 5, Sherlock, and Border Patrol. Two other requirements, that the recorded data not share fate with messages in the datapath, and that we minimize the performance overhead on the datapath, dictate that we keep constant size metadata with the datapath, and record any extra metadata separately. Also because we want to capture end-to-end traces in the general case, we opt for modifying applications and runtime systems, when necessary, to run X-Trace. Section 3.3.1, in the next chapter, examines in detail how X-Trace's features combine to fulfill the requirements. X-Trace is unique among related works in that it aims at working across layers, applications, machines, and administrative domains, and in the general case. This aim for generality precludes approaches such as that in Magpie, that require extensive knowledge of the instrumented system, and also places a focus on standard metadata and reporting formats.

Quanto shares some of the same requirements of deterministic causal tracking with minimal intrusion in the datapath. Thus, it propagates fixed size metadata and logs events based on the metadata it receives. Because of the very stringent requirements of the environment, Quanto's metadata has only two bytes of information. Quanto is unique in that it combines very precise

profiling of energy with activities that comprise causally connected events spanning software layers, hardware components, and network nodes.

We have now set the state to present both X-Trace and Quanto, their design, implementation, and evaluation. We begin in the next four chapters with the presentation of X-Trace, followed by Quanto in Chapters 7 and 8.

Chapter 3

X-Trace: A Tracing Framework for Distributed Systems

3.1 Introduction

In this chapter we present the design and architecture of X-Trace, a framework to provide visibility into loosely coupled distributed systems, by tracing the execution of well-defined tasks as they exercise different parts of the system. X-Trace can be used by developers, operators, or end-users to gather data about specific executions of the system. This data can be used for troubleshooting, debugging, monitoring, and performance tuning, among other purposes.

A key aspect of the systems X-Trace targets lies not only in their scale and complexity, but in their heterogeneity and loosely-coupled nature. They generally comprise multiple application, middleware, and protocol components that are developed and deployed by independent parties. They span several layers of abstraction, including multiple software and networking layers. Furthermore,

these systems have components under diverse administrative control, both within single organizations and across separate organizations.

X-Trace, as we discuss in Chapter 2, builds upon a vast body of previous work on debugging, profiling, and troubleshooting sequential, parallel, and distributed systems. Its main focus, however, is in addressing the difficulties imposed by this heterogeneity. X-Trace provides a task-centric view of the execution in such systems, spanning the barriers of multiple independent components, layers, and administrative domains.

Before we describe in more detail how X-Trace works, we make our discussion more concrete using Wikipedia, a popular online encyclopedia, as a motivating example. In Section 3.1.2 we describe the requirements we have for X-Trace, derived from the environments where we expect it to be useful. Section 3.2 gives a detailed architectural description of X-Trace, including its formal execution model and the design that implements the model. We describe how we follow the distributed execution of a system by propagating well-defined metadata, and how we collect information. Section 3.3 then looks at how the different architecture components fulfill the requirements, and examines other issues such as how X-Trace copes with lack of clock synchronization, report volume, and security. The chapter ends with a discussion of some limitations of the architecture.

3.1.1 A Motivating Example

Wikipedia is a a collaborative, open-content encyclopedia, that was, according to Alexa, the seventh most popular Web site in the world as of July 2008 [6]. Figure 3.1 shows a simplified snapshot of the infrastructure as of the same time. They had servers spread across 3 worldwide sites, comprising 90 web caches chosen by DNS assignments, 7 load balancers, 161 web servers, and 17 database servers [175].

A request from a user starts with a DNS request, to a site based on geographic location. A subsequent HTTP request then hits a load balancer, which selects a cache server. The requests may optionally transit another load balancer, a web server, and eventually a database. Caching is done at each of these levels. Even though Wikipedia's infrastructure in 2008 is at least 3 orders of magnitude smaller than Google's [107], for example, the number of possible paths for a user request through the system is already very large.

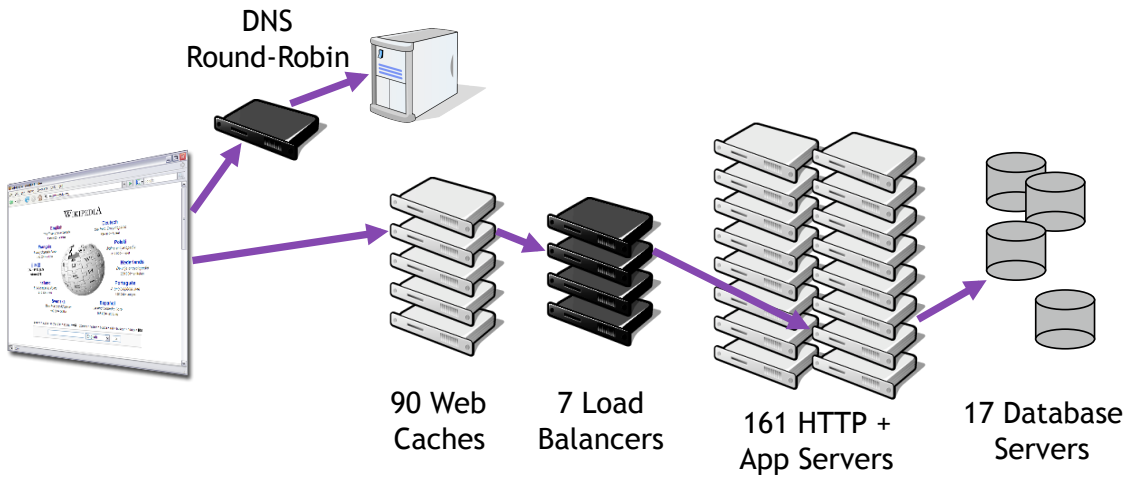


Figure 3.1: Schematic Wikipedia infrastructure as of July 2008 [175].

Suppose a user updates a page on Wikipedia, and fails to see her updates upon reloading the page. Of course, it may be the case that by design it takes some time for changes to propagate, but let us assume, for the sake of the argument, that this behavior is considered an error. It is very difficult to detect the cause of this error and to correct it, for a number of reasons.

First, there may be no alarm of the error to the administrator, because the error was not the hard failure of a component, but a semantic error that is detected by the user of the site. Other

similarly user-detectable errors include failure to add an item to an online shopping-cart, or a post to a message board being lost. If the user is able to report that there was an error, the best that she can do is to tell an administrator the time, the page she was accessing, and the IP address of the computer she was using.

Second, the administrator may not be able to reproduce the execution path that lead to the error, because she will re-submitting a request from a different network location. The user might be more sophisticated, and be able to run traceroute to tell the administrator which server might have received her HTTP request. While this is already more that can be expected from an average end-user, it will still leave the administrator with a difficult path to find the error. Because traceroute will issue a different DNS request, it might return a different server altogether.

Third, it will be at best laborious, and at worse impossible, for the administrator to piece together logs from different components to reconstruct the user's request path. Such a search would start from logs from the front-end caches, looking for occurrences of the user's IP address, URL, around the correct time, and proceed down the chain of calls to other components. Wikipedia has front-end caches in the US, in Amsterdam, and in Korea. It may be that a single administrator does not have immediate access to the logs in all locations. For this approach to succeed, individual log files must have common tokens that link recursive requests together, or the chain will be broken. There is no guarantee that this chain exists, because system components are written independently, and there is no standard or specification on how to create these links among different logs.

X-Trace allows unified tracing across the boundaries of application components, software and network layers, and administrative domains, enabling the reconstruction of execution paths through a complex system. The approach is to use well-defined metadata, carried with the execution by

applications and protocols, and well defined event reports that contain enough information to place these events into a causal graph of the execution.

3.1.2 Requirements

The example above highlights some of the problems in the scenarios in which we want to apply X-Trace. In this section we derive some functional and feasibility requirements for X-Trace. We come back to these after we describe X-Trace's design, to see how our design decisions map to the requirements.

Functional Requirements

The first group of requirements are functional requirements, representing what X-Trace must provide as an output.

- **Deterministic Causality** We want X-Trace to provide a deterministic representation of the execution in the system, in the face of arbitrary concurrency. X-Trace should capture the *happened before* relation among events in the execution across application components, layers of abstraction, and administrative boundaries.

- **Trace the execution path** We want to trace the actual execution path of the system, rather than a decoupled reenactment or approximation of the path. In the Wikipedia case, it was difficult or impossible to reproduce the same path through the system twice. Traceroute, for example, sends independent probes to a given IP address, to discover the routers in the path. There is no guarantee that separate probe packets take the same path as application traffic, and in systems with more complex execution paths than IP routing, it may not even be possible to

exercise these paths by other means than real execution.

- **No trace data fate sharing** In contrast to the need to trace the actual execution path, we want the resulting trace data to not *share fate* with that execution path. This is desirable for two reasons. First, we can still recover (partial) data about the execution even if the path is interrupted. This can greatly narrow down the search for possible causes of such a failure. Second, by not carrying cumulative trace data in the execution path, we avoid a significant source of overhead.

Non-functional Requirements

The requirements in the second group are non-functional, but have a direct influence X-Trace's applicability in the real systems we target.

- **Minimal assumptions on the environment** To maximize applicability of X-Trace to diverse distributed systems, we make as little assumptions about the system and the environment as possible. For example, we cannot assume that we know in advance all the components or machines involved in an execution, the programming model with which components are written, or the events or variables that are of interest. The latter two, for example, can vary within a single system across its several components.

- **Low overhead** A very important requirement is that X-Trace have a low overhead in the instrumented system, ideally to the point of it being feasible to use it as an always-on monitoring solution. This has a number of dimensions. The instrumentation should minimize the delay imposed on the execution path, the extra data (if any) that is sent along communication paths, and the state required at each computational node in the system to perform the tracing.

- **Policy Control** Given that systems often have components under different administrative domains, pervasive tracing will only be feasible when individual parties have control over the information that is released to other parties in the system. In particular, X-Trace should allow each party to control, based on the type of execution and on who is requesting trace information, the granularity of the information provided, if any. For example, given a request for what happened to a specific execution of their system, a hosting company could provide very detailed information to a direct client, but a very coarse report stating that the operation did not complete, in case the requester is not a client.

- **Incremental Deployment** Given the multiple components and independent parties that characterize the systems we target, it is fundamental that X-Trace be amenable to incremental deployment. This applies to cases in which a single party instruments different components of a system in stages, and also when different parties independently instrument separate components of the system. The implications are that instrumented components must remain compatible with uninstrumented ones, that the captured execution model support increasing levels of granularity and partial instrumentation, and that there be standard ways of generating, transmitting, and sharing trace data.

Other issues

Finally, it is also worth mentioning two significant aspects that we decided to not consider as requirements for X-Trace. In Section 3.3.3 we discuss how these contribute to some of the limitations of X-Trace.

- **Change to legacy systems** To achieve the full visibility into the execution of systems, we

allow for, and sometimes require, the modification of existing systems for X-Trace instrumentation.

Not changing legacy systems is a requirement in other tracing and monitoring systems [95, 4, 14, 20], and makes it easier to deploy tracing. However, there are instances in which one can only determine the true causality by tracking the execution at the highest level. For example, if a system receives a number of quasi-simultaneous requests as inputs, and subsequently issues a number of recursive calls, it may be impossible to correctly attribute the recursive calls to specific input requests unless the internals of the application are known. This will happen if the externally visible requests themselves do not have enough information to allow disambiguation, and if it is not possible to infer the causality from the timings of the requests. X-Trace allows the instrumenter to define which parts of the system are black-boxes in a flexible way.

- **Off-line analysis**

  Lastly, we do not require that the data produced by X-Trace be analyzed in real time, or in an on-line fashion. This allows trace data to be aggregated, filtered, compressed, and the decoupling between the execution and the analysis contributes to decrease the overhead imposed on the execution.

With these requirements in mind, next we describe the model with which X-Trace represents an execution, and the details of its design and implementation.

3.2 Architecture

3.2.1 Execution Model

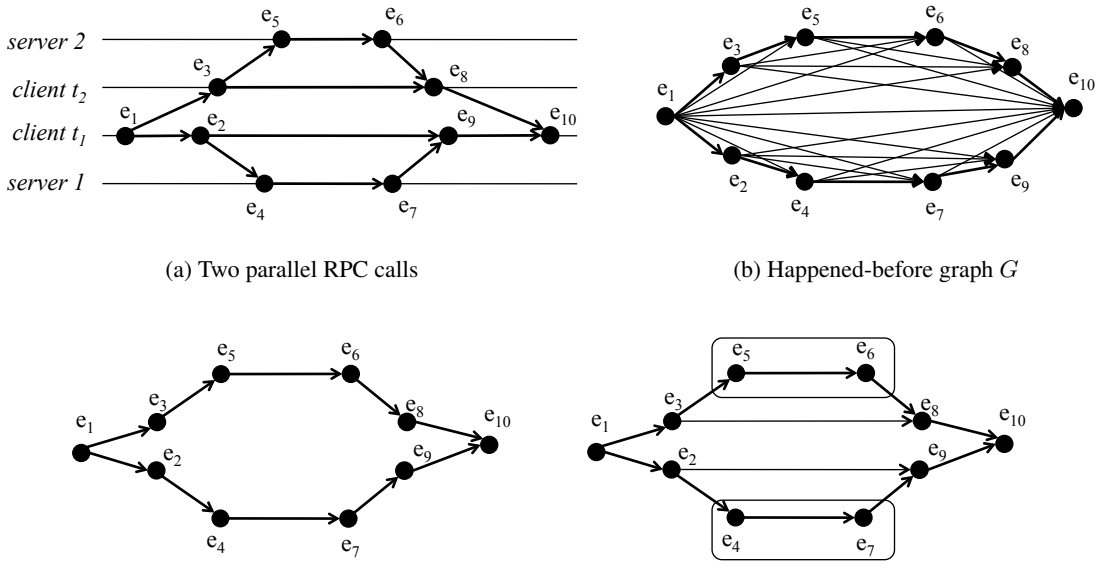
Following from the first requirement of deterministic causality capture, X-Trace models the execution in a distributed system using causally related events, recording the happened-before relationship introduced by Lamport [99]. Rather than being process-centric, however, X-Trace groups events into tasks. A task consists of a starting event, and the set of events causally related to it, and generally maps to a high level activity meaningful to a user, programmer, or operator.

We assume that there are a set of logical threads of computation that can be executing in the same or different processors in the system, in the same or different machines. Threads execute a sequence of events. These threads may communicate with each other by means of messages or through shared memory. Events can be internal events, read/write events to shared variables, or send/receive events for messages between threads. We call read and write events access events.

More formally, X-Trace represents a computation as an irreflexive partial order $(E, \rightarrow)$ on a set E of relevant events of the execution. The partial order is determined by Lamport's happened before relation ($\rightarrow$), defined here as the smallest *transitive* relation that satisfies the following properties:[1]

1. If e and f are events in the same thread, and e comes before f, then $e \rightarrow f$.
2. If e is the sending of a message by a thread, and f is the receipt of the same message by another thread, $e \rightarrow f$.
3. If e is an access to a shared variable, and f is a subsequent access to the same variable, $e \rightarrow f$.

[1] We combine here traditional definitions for message-passing (e.g. [99]) and shared memory systems (e.g., [2]). The important aspect is how we can establish causality in both.



(a) Two parallel RPC calls

(b) Happened-before graph G

(c) Transitive reduction G' of G

(d) Typical X-Trace representation

Figure 3.2: Event-based representation of two parallel RPC calls between a client and two servers (a), with the corresponding happened-before transitive closure graph (b). The transitive reduction in (c) removes all redundant edges and clearly shows the parallelism. In (d), X-Trace may choose to leave some redundant edges. Edge e_3e_8 abstracts the marked subgraph (e_5, e_6), and edge e_2e_9 abstracts the marked subgraph e_4, e_7).

Two events e and f are concurrent, written $e \parallel f$, if $e \nrightarrow f$ and $f \nrightarrow e$.

The partial order defines a directed acyclic graph G among the events, where there is an edge between events e and f iff $e \rightarrow f$. This graph is the transitive closure of the partial order relations from the definition. Figure 3.2(a) shows an example computation with two parallel RPC calls, with relevant events in each thread marked as circles. Figure 3.2(b) shows the complete partial order relation among these events.

An edge in G is said to be redundant if it can be removed without changing the reachability of

the graph. If we remove any such edge from G, we obtain a graph *equivalent* to G, or a graph with the same reachability among nodes. If we remove all of the superfluous edges from the graph, we obtain a graph G', called the minimum equivalent graph (MEG) [81] of G. For acyclic directed graphs, the MEG is unique, and is also the transitive reduction of G [5]. G' can be computed in polynomial time from G or from any graph equivalent to G. The transitive reduction is interesting because it is a compact representation of the causality, and it clearly shows the parallel structure of the computation. In Section 4.4.1 we show an algorithm for determining the transitive reduction of X-Trace graphs. Figure 3.2(c) shows the transitive reduction for our simple example.

X-Trace allows relevant *events* to be recorded in any of the threads of the execution, and to later reconstruct the partial order on these events. X-Trace records the partial order by directly representing a reduced graph equivalent to G, and we call these graphs *task graphs*. It does this by recording an report for each event in the graph. An event report has a unique identifier for the event, and a representation of each incoming edge to the event in the graph. The assumption here is that when registering an event, the X-Trace instrumentation has access to the event identifiers of the immediately preceding events.

A task graph is not necessarily the transitive reduction G', and may include some of the redundant edges. These edges are explicitly added to the information recorded by X-Trace by the instrumentation, and may be interesting for two purposes. First, some redundant edges may represent meaningful abstractions, such as a sub part of a task, or a set of events that happen in another layer. These can be used for example to summarize the graph both for visualization and analysis. Second, recording these edges adds redundancy to the instrumentation. In the example in Figure 3.2, if Server 1 were not instrumented with X-Trace, we would still be able to recover the flow of the

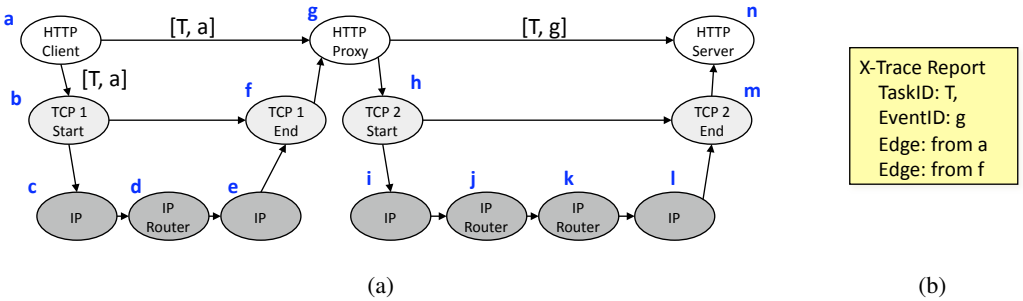


Figure 3.3: X-Trace annotation of an HTTP request through a proxy. (a) shows the task graph. All events in this task share the same TaskID*, and each one gets a unique* EventID*. Each recorded event generates a report with at least the* TaskID*,* EventID*, and the* EventIDs *of the immediately preceding events. (b) shows the report generated by the recording of event g.*

computation because the edge e_2e_9 would be recorded at the client. This redundancy gives X-Trace flexibility to define different granularities of tracing, and to cope with partial and incremental deployment.

3.2.2 Design

Differently from timestamp-based approaches [99, 52, 109], X-Trace directly records and reconstructs a task graph as described in Section 3.2.1. Each task in X-Trace receives a unique *TaskID*, and each event within the task receives a unique *EventID*. There are only two requirements for X-Trace to record an event:

1. The event recording code must have access to the *EventID* (s) of the immediately preceding, causally related event(s); and

2. The causal edge(s) between the current event and the preceding events must be recorded.

X-Trace satisfies (1) by propagating *metadata* with the computation that encodes the *TaskID* and the *EventID* of the last event in a thread. It satisfies (2) by creating *reports* for each recorded event. These reports contain references to the *EventID* (s) of the preceding event(s), the *TaskID* of the current task, and optional information. They are sent to a reporting infrastructure that is not part of the traced system's datapath, for later analysis.

As an example, Figure 3.3(a) shows an HTTP request through a proxy. It is simplified, as it does not take into account the control packets for TCP, retransmissions, or fragmentation. X-Trace gives each event in the graph a unique *EventID* (the lower case letters $a, b, c, \ldots$), and they all share the same *TaskID* T. The tuple [*TaskID*,last *EventID*] must be carried along each edge in the graph, and used to record the event reports. In (b), we see the contents of the report from event g, indicating the *TaskID*, and the edges from a and f.

Figure 3.4 shows an overview of an X-Trace-instrumented system. In this case, the user starts tracing by creating a *TaskID* and adding it to the protocol messages her machine sends out. X-Trace metadata is propagated among components of the system through protocol messages, and within single components through the runtime environment. Components can then record relevant events to the reporting infrastructure, which X-Trace will then correctly place in the task graph.

This design follows from the requirements in Section 3.1.2. By propagating the *TaskID* and *EventID* with the execution, we trace the execution path, and by sending reports to a reporting infrastructure, we decouple the fate of the reports from that of the execution path. This model doesn't make any assumptions about the environment, other than that the metadata can be propagated and events recorded.

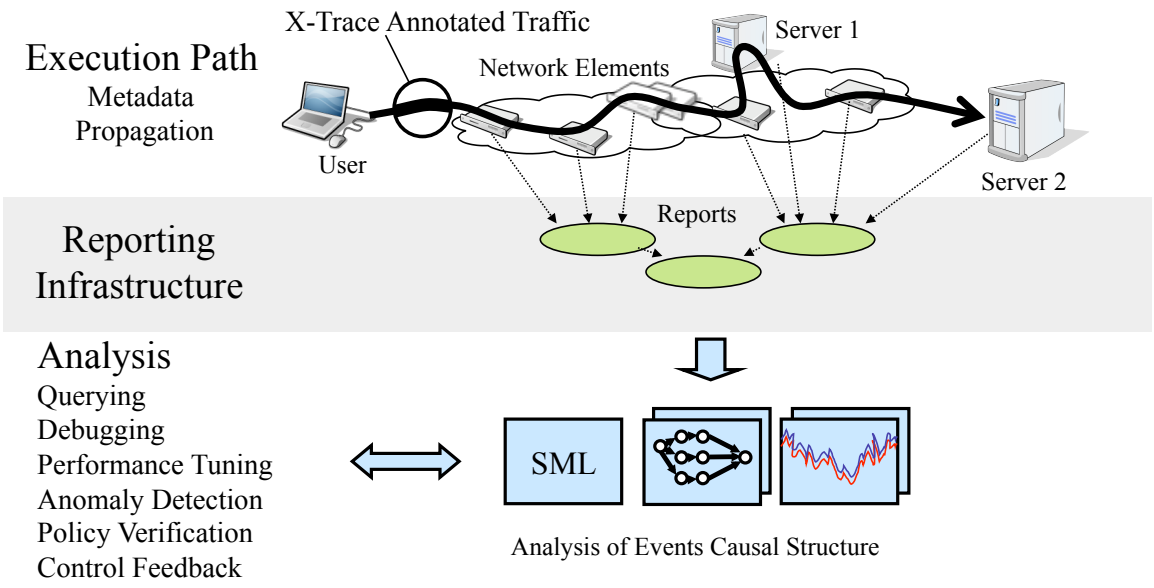


Figure 3.4: X-Trace system overview. The traffic annotated with X-Trace metadata triggers different elements along the execution path to send reports, which are later aggregated and can be analyzed by different techniques.

We now describe in more detail the three main elements of X-Trace, the metadata contents, metadata propagation, and event reporting.

3.2.3 Metadata Format and Semantics

The X-Trace metadata is what allows tracing information to be carried with the computation. A component adds X-Trace metadata to the execution path to start a trace. This can be, for example, a user browser, or the ingress router on a service provider network. It consists of a *TaskID*, an *EventID*, and optional extension fields, and is carried in the execution path by the X-Trace framework.

The *TaskID* uniquely identifies a task. This identifier should be unique among all tasks reported to the same reporting infrastructure over a reasonable window of time. The *EventID* uniquely identifies an event *within a task*.

The component that starts tracing a task is responsible for choosing a *TaskID*, and each component that logs an even is responsible for choosing that event's *EventID*. In our implementation both of these identifiers are opaque integer identifiers, and are chosen by a random number generator. This approach provides probabilistic uniqueness, controlled by the size of the identifier space, and has two advantages. Choosing an identifier is a constant-time operation, and it doesn't require any external coordination. Both aspects are important to minimize the time overhead of tracing.

Options are self-describing extension fields which may or may not be present. Currently there are two types of options defined: *destination* and *severity*. Both options affect reporting of events. Destination is a hint for where reports should be sent, and severity indicates the importance of the current task, for run-time determination of reporting granularity.

The metadata has constant size throughout the execution, as it always contains one *EventID*, the last one in the current execution thread. This contributes to minimize the overhead. For events in

the graph which depend on more than one previous event, as is the case of *barriers*, this dependency is recorded in the report for that event, and each incoming edge only carries one *EventID*. For the case of an event that initiates parallel branches in the computation, the same causing *EventID* is sent along each branch.

The metadata contains all information needed to trace a task, and does not require that components keep state about X-Trace propagation or reporting. It has a compact binary representation that minimizes the overhead when adding it to messages and to run-time environments, and follows a well-defined specification, reproduced in Appendix A. Its standard format allows independent implementations in different components to work together to trace the same task.

3.2.4 Metadata Propagation

Components must propagate metadata along the execution path being traced. The instrumentation needs to maintain the invariant that the event being recorded have access to the *EventID* (s) of the preceding event(s). Propagation needs to happen between software components and inside components. The first case comprises protocol messages and API calls, whereas the second case involves instrumenting the runtime environments.

Propagation in protocols requires that a suitable representation of the X-Trace metadata be added as metadata to protocol messages, generally as extension fields to protocols. Examples of such fields are IP options, HTTP extension headers, and TCP options. In Appendix A we define binary and text representations of X-Trace metadata for this purpose.

As X-Trace can represent events that happen across different software layers, in some cases the propagation of X-Trace metadata can happen by augmented APIs between successive layers. In one example, we added a `setsockopt` call to Linux to set the metadata options for outgoing IP

packets. An alternative to changing APIs is to use system-wide solutions like Causeway [28] or SDI [138].

Propagation within software components is the more interesting case. Applications must support two aspects of X-Trace metadata propagation. The first is to extract the X-Trace metadata from incoming messages and calls, and correctly add metadata to the causally related outgoing messages. This involves following the execution paths, and in Section 4.3 detail strategies to do this in applications with different programming models.

The second requirement regards correctly logging relevant events in arbitrary points in the execution. To log an event, an application has to do the following sequence of operations:

1. Gather the *EventID* (s) of the immediately preceding event(s)
2. Generate a new *EventID* for the current event
3. Generate a report that records the binding between the incoming *EventID* (s) and the new *EventID*
4. Make X-Trace metadata with the new *EventID* available to the next event(s) that immediately follow.

Figure 3.5 shows in detail how the propagation of the metadata happens in the HTTP proxy example from Figure 3.3.

3.2.5 Reporting

When a component sees traffic that carries X-Trace metadata, it generates reports for relevant events, which we use in post-processing to reconstruct task graphs. Reports are fundamental to

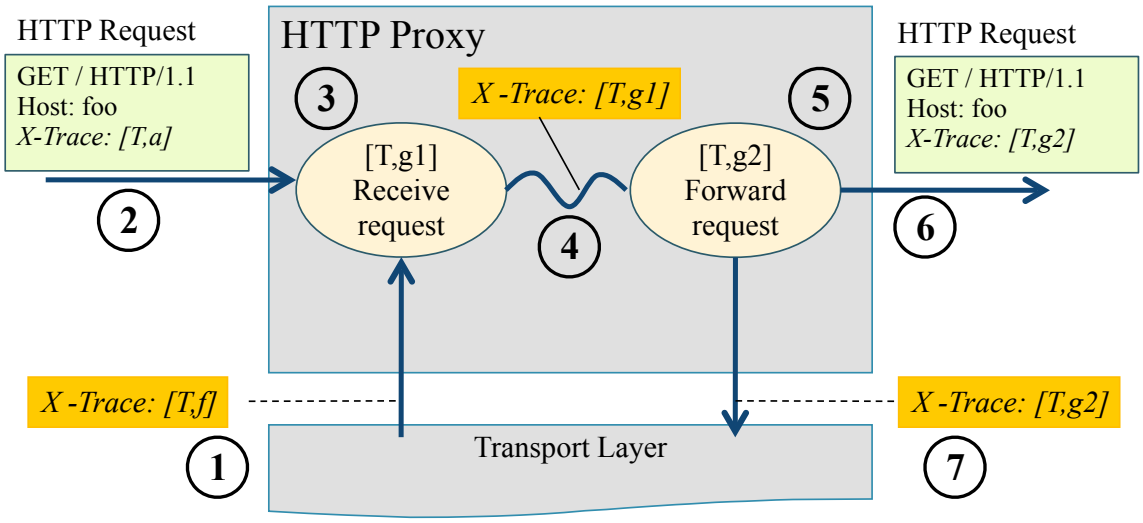


Figure 3.5: Detail of metadata propagation in the HTTP proxy example. This Figure shows how the edges in the graph correspond to different types of metadata propagation. The proxy is logging two relevant events $g1$ and $g2$. It receives data from the transport layer, and metadata through the API (1). It can then decode the HTTP header, which also has metadata encoded (2). In (3) it sends a report, and then propagates the new EventID *in the runtime environment (4). The reverse happens in (5), (6), and (7).*

record not only information about events, but the binding between successive events. Each new event records in a report all of the incoming edges, and needs to propagate only its own *EventID*, assuring we can represent the task graph with constant-sized metadata.

Each event in X-Trace generates a *report*. The mandatory information in an X-Trace report consists of the *TaskID* of the task to which the event belongs, the *EventID*, and the *EventID* (s) of all immediately preceding events. Components can also record redundant edges, not in the transitive reduction of the partial order (*c.f.* Section 3.2.1). In the HTTP example from Figure 3.3, the edges ag, bf, gn, and hm are redundant, but are meaningful to abstract specific subgraphs and add robustness to partial deployment.

It is also very useful to include additional information in the reports, like a label, a local timestamp, machine name, and other information relevant to the specific event. The exact fields included will vary with the system, application component, and specific details of the event itself. For example, an HTTP cache may report on the URI and cookie of the request, and the action taken upon receiving the request. It can also add systems information such as the server load at the time. IP routers, on the other hand, report information contained in the IP headers of packets, such as source and destination addresses, and can add other relevant performance information such as current queue lengths.

X-Trace reports are text-based, composed of key-value pairs of information, similar to the headers in RFC 822 [38]. This allows for a simple and flexible schema. We define the format of the report, but not all of the fields. The common format allows different implementations of report collection and processing to work together in the same infrastructure. An example report from the Apache HTTP web server is shown in Figure 3.6.

```
X-Trace Report ver 1.0
X-Trace: 104E2271D8EA71C723
Timestamp: 1216237571.364
Edge: 6581A5CE
Agent: Apache
Label: Received Request /ctest
Host: <hidden>.cs.berkeley.edu
```

Figure 3.6: A sample X-Trace report from the Apache http server.

X-Trace is flexible in how reports are aggregated, and gives complete control over the report data to the components generating the data. It is useful to group components in Administrative Domains, or ADs.

The reports generated by devices within one AD are kept under the control of that AD, according to its policy. That policy could be to store all the reports in local storage, such as a database. The AD can use this store of reports to diagnose and analyze flows transiting its network.

For cases with multiple ADs involved, we use a *destination* option in the X-Trace metadata. If present, this option signals that a user (located at that destination) is interested in receiving the trace data as well. This user might be the originator of the task, or it could be any delegated report server. This indirection is useful for users behind NATs, since they are not addressable from the Internet. The AD uses its policy to respond to this request.

The simplest policy is for each device to just send reports directly to the indicated destination, which would collect them and reconstruct the task graph. This may not be desirable, though, because AD's in general will want to control who has access to different granularities of data. One possible mechanism that uses indirection works as follows. The AD still collects all reports locally in a private database. It then sends a special report to the destination in the metadata, containing a pointer to the report data. The pointer could be the URL of a page containing the trace data. This

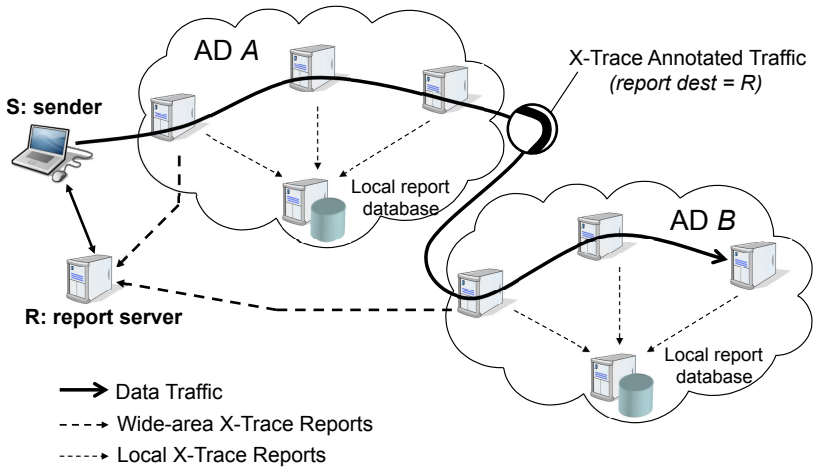


Figure 3.7: An example of wide-area reporting. The client embeds X-Trace metadata with a message, setting the report destination to R. Different ISPs collect reports locally, and send pointers to R so that the client can later request the detailed reports.

gives each AD control of the visibility of the trace information, by requiring users authenticate themselves when they fetch the data. The AD can make use of this authentication information when choosing the level of detail of the report information returned to the user. We describe this usage in more detail in Section 4.1.2. Note that all the information needed to get a report to a user is kept in the X-Trace metadata, meaning that nodes in the network do not need to keep any per-flow state to issue reports.

Figure 3.7 shows a sender S who sets the destination for reports as being the report server R. ADs A and B send pointer reports to R, and either the client or R itself fetches these reports later. A special case is when the *user* of X-Trace is in the same AD as the devices generating reports, such as network operators performing internal troubleshooting. X-Trace metadata gets added at the AD ingress points. The network operators go directly to the local report databases, and there is no need to use the destination field in the metadata.

Once the interested user collects and aggregates reports for tasks of interest, she can reconstruct the task graph, or the part of the task graph she has access to. Different ADs may store different sub-graphs related to the same task. The fact that all of the events in a task refer to the same *TaskID* means that these ADs may choose to combine or share these subgraphs as needed or desired, to solve specific problems. The *TaskID* serves as a common handle that different parties can use to refer to the same execution.

The reconstructed graph is the end product of the tracing process, and can be stored, associated with trouble ticket systems, or used by operators as a record of individual failure events for reliability engineering programs. For transitory errors, the graph serves as a permanent record of the conditions that existed at the time of the connection. Additionally, any performance data included by the devices in the reports can be used to correlate failures in the datapath with devices that may be under-performing due to overload.

3.3 Discussion

Before moving on to describing the X-Trace's implementation, in Chapter 4, and evaluating it in constructed scenarios and in two deployed wide-area applications (Chapters 5 and 6, respectively), we summarize the architecture below, relating X-Trace's features to its requirements. We then look at some other important issues and limitations of X-Trace.

3.3.1 Revisiting the Requirements

Having presented the basic architecture, we now revisit the requirements from Section 3.1.2. Table 3.1 maps the features of X-Trace we described above to how they contribute to fulfill the

requirements.

The X-Trace execution model captures the full happened-before relation among events with its explicit event-graph representation. The model is general, and makes no assumptions on the programming model or system structure, other than the possibility of carrying X-Trace metadata.

The in-band X-Trace metadata allows capturing of the actual execution path, rather than a separate path that would be captured by independent tracing traffic. There can be alternatives to adding explicit metadata. Magpie [20], for example, does tracing without adding metadata, but assumes very comprehensive logging and complex schemas to be able to recover bindings between event. We adopted the metadata solution in part to minimize the assumptions on the traced system.

Also because of minimal assumptions on the instrumented systems, the event reports are a set of text-based, key-value pairs. While not as efficient as a codified binary representation, we opted for the flexibility afforded by the text representation. The reports in X-Trace do not share fate with the execution after they are generated, as they are sent to a separate reporting infrastructure.

These out-of-band reports also also contribute to minimize the overhead. Once generated, reports do not have to be sent synchronously with the execution, thereby decreasing the imposed delay. Overhead is also minimized by constant-sized metadata and constant time metadata propagation. Further, because the metadata has all information needed for reporting, components do not have to keep long-lasting state for tracing.

The standard metadata and report formats allow different administrative domains to independently instrument their systems. The *TaskID* acts as a common handle for the exchange of information when necessary, and the decoupling of trace data request and generation allows the different ADs to apply appropriate policies to share information.

	X-Trace Features									
Requirements	Explicit event graph representation	In-band metadata	Out-of-band reports	Flexible Key-value pair reports	Constant size metadata	Constant time propagation	No state on nodes	Decoupled trace request and reports	Standard metadata & report formats	Explicit redundant-edges
Deterministic Causality	●									
Execution path tracing		●								
No trace data fate sharing			●							
Minimal assumptions on environment	●	●		●						
Low overhead			●		●	●	●			
Policy Control								●	●	
Incremental Deployment									●	●

Table 3.1: How different features of the X-Trace architecture map to the requirements laid out in Section 3.1.2. A dot in a table column indicates that the feature is important in fulfilling the requirement in the corresponding line.

Finally, the task graph model naturally allows for incremental deployment of X-Trace, by the same or different parties. If, for instance, we only instrument the RPC client in Figure 3.2, we will get a graph with the events e_1, e_2, e_3, e_8, e_9, and e_{10}. If we later instrument the servers and trace another execution, we get the full graph from Figure 3.2(d). The edges e_2e_9 and e_3e_8 become redundant edges, but there is no need to change the instrumentation in the client.

3.3.2 Other issues

Time synchronization

An important feature of X-Trace is that it does not require that different machines that are part of a task be time-synchronized. Lack of time synchronization may imply erroneous sequencing for traditional log-based approaches that rely on time to order events. Because X-Trace explicitly captures the precedence among events, not only does it not require wallclock time to be correct among systems, it can actually help correct time offset and skew. Whenever X-Trace records a pair of edges in opposite directions among two machines, such as in an RPC request-response pair, it can use techniques similar to NTP [115], or to the ones in [128], to correct the clock in one of the machines, making the time consistent to within the magnitude of the network delay between the machines.

Managing Report Volume

The structure and complexity of an application's task graph have a strong bearing on the amount of report traffic generated by X-Trace nodes. We mention three mechanisms that can limit the volume of this traffic.

- *Batching and compressing.* Since X-Trace reports are delivered out-of-band, they can be *batched* and *compressed* before transmission. We have routinely observed a compression factor of around 10x for X-Trace generated reports using `bzip2`.

- *Correlated sampling.* Sampling can limit the number of requests that are tagged with X-Trace metadata to a rate specified by policy. X-Trace allows correlated sampling across multiple components of a system, by only selectively adding X-Trace metadata to a fraction of initial operations. Differently from independent sampling at each node, using X-Trace, each "sample" is a complete task graph. Sampling can independent or biased to specific types of tasks.

- *Per-task granularity.* Lastly, X-Trace allows variable granularity of logging per task. In Section 4.1.3 we show how an X-Trace metadata option, *severity*, allows most tasks to be logged at a coarse granularity, while letting a few selected tasks be logged in more detail.

As associated challenge with report volume is the aggregation of reports in a central location. Chun et al. citechun08d3 reported some initial progress on D3, a system that can answer declarative queries on X-Trace task graphs in a distributed way, without requiring all reports for a task to be collected to a central node. They do this by using hints, in the X-Trace reports, about what the next node with local reports is likely to be.

Security Considerations

It is important to discuss the potential for attacking the X-Trace infrastructure, as well as using that infrastructure to attack others.

First, one could mount an attack against an infrastructure that implements X-Trace by sending

an inordinate amount of traffic with X-Trace metadata requesting reports. We argue that propagating metadata on its own is unlikely to become a bottleneck in this situation. Generating reports, however, could become a significant source of load. A simple defense is for each device to rate-limit the generation of reports. Still, malicious clients could get more than their fair share of the reporting bandwidth. If this becomes a problem, and filtering specific sources of reports becomes an issue, providers might start requiring capabilities in the options part of X-Trace metadata to issue reports.

Another possible attack with the reporting infrastructure is for a malicious user to send packets with X-Trace metadata, with the destination for reports set as another user. In the worst case, many network devices and hosts would send reports towards the attacked user. While this attack is possible, it will not have an exponential growth effect on the attacker's power, as legitimate reporting nodes will not place X-Trace metadata into X-Trace reports. Most important, however, is that we do not expect a large traffic of wide-area reports: as we describe in Section 3.2.5, we expect ADs to generate few wire-area reports with pointers to detailed, independent stores for local reports within each AD. X-Trace keeps control of report generation rate and visibility with each report provider, which allows for defense mechanisms to be independently put in place.

3.3.3 Limitations

X-Trace is not without its limitations, partly due to choice of target systems and requirements, and we discuss three important ones next.

Modification to Systems

One of the main limitations of X-Trace is that it requires that legacy systems and protocols be modified. There is a tradeoff here between the ability to deterministically capture causality of

general distributed systems versus the degree of intrusiveness of the monitoring, as we discussed in more detail in Chapter 2.

While the modifications to add X-Trace tend to be simple, this can be an impediment to using X-Trace with many production and legacy systems. Modifications can also be hard or impossible to make in many hardware-based implementation of network protocols, for example.

Many protocols, such as HTTP, are designed with extensible fields, and these make adding X-Trace straightforward. An approach that makes the impact of the changes much smaller is the instrumentation of key libraries and middleware services with X-Trace, as we did, for example, with the `libasync` asynchronous event and RPC libraries.

Resiliency to Report loss

Up to this point we have assumed that all reports are reliably collected by the reporting infrastructure. If the reporting infrastructure loses any reports, the effect to the graph will be the deletion of nodes and edges represented by that report. This might make it impossible to fully reconstruct the causal connections, disconnecting the graph. In these cases, the reports sharing a common task identifier could be ordered temporally. Although not as descriptive, this linear graph might still pinpoint certain faults. Another promising approach in the case of lost reports is to infer about the existence of missing edges by comparing the structure of the tasks with missing reports to that of similar tasks.

Cross-task interaction

Another limitation of X-Trace as described here is that it is not natural to capture interaction among different tasks. For example, in a cache, when a task B retrieves an object that had been

previously stored in the cache by a task A, clearly the storage event in A preceded the retrieval event in B. In this case, it is possible to augment the edge representation and allow edges between events from different tasks.

X-Trace as proposed assumes that each event belongs to only one task, which might not be the case in some situations. For example, if we have a protocol U that batches several messages (from different tasks) to be transmitted together in a single message of an underlying protocol L, the sending and receiving events in the protocol L layer will be assigned to one of the tasks. The causality will be preserved correctly in the U protocol, but the events in L would have to belong to all k tasks simultaneously. It is possible again to extend the event representation to cross task boundaries, but X-Trace does not do that as presented here.

There are events that implicitly share a resource, such as a network link or the CPU time. In these cases there is no message or shared data access, and it would currently require the examining of the timing in each report to infer the relationships.

Having described the design and architecture of X-Trace, the next chapter delves into details of how we implemented X-Trace, including specifics of the metadata and reporting formats, the challenges we found integrating X-Trace into new and existing systems, and how we process and visualize X-Trace-produced data.

Chapter 4

X-Trace Implementation

In this chapter we describe details of our implementation of the X-Trace architecture presented in Chapter 3. We begin, in Sections 4.1 and 4.2, by describing the X-Trace metadata format and the reporting infrastructure. A very important issue with X-Trace is how to integrate it with existing and new systems. In Section 4.3 we describe how we integrate X-Trace into protocols, how we provide a unified API for integrating it into both event- and thread-based systems, and discuss important challenges such as capturing concurrency correctly and deferring computation. Finally, in Sections 4.4 and 4.5 we discuss how we process and visualize X-Trace data.

4.1 Metadata

Figure 4.1 shows the format with which we encode the X-Trace metadata. Since X-Trace includes a serialized version of the metadata in protocol messages of instrumented systems, it is important to achieve a compact representation.

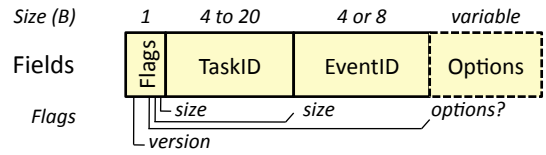


Figure 4.1: X-Trace metadata.

4.1.1 Identifiers

We represent the *TaskID* and the *EventID* as opaque integer fields. The *TaskID* can have a length of 4, 8, 12, or 20 bytes, and the *EventID* can have a length of 4 or 8 bytes. These identifiers are preceded by a *flags* byte, which indicates the X-Trace metadata version, the sizes of the *TaskID* and of the *EventID*, and indicates whether there are options in the metadata. To accommodate future extensions to the X-Trace identifier format, we include the options block. This block, if present, consists of one or more individual self-describing options. Each consists of a type, a length, and then a variable length payload.

We use randomly chosen identifiers for both the *TaskID* and *EventID*, as this can be done very fast without coordination by each instrumented component. Of course, there is still a chance of collision in the identifier space, but the probability of this happening can be bounded by the length of the identifier.

Collisions in the selection of the *TaskID* should be avoided for tasks that go to the same reporting database, during a given window of time that depends on typical task duration. In other words, the *TaskID* length can be made smaller if the duration of individual tasks can be bounded. The task graph for a *TaskID* with such collisions will have a number of disconnected components, and can be identified with simple graph reachability algorithms.

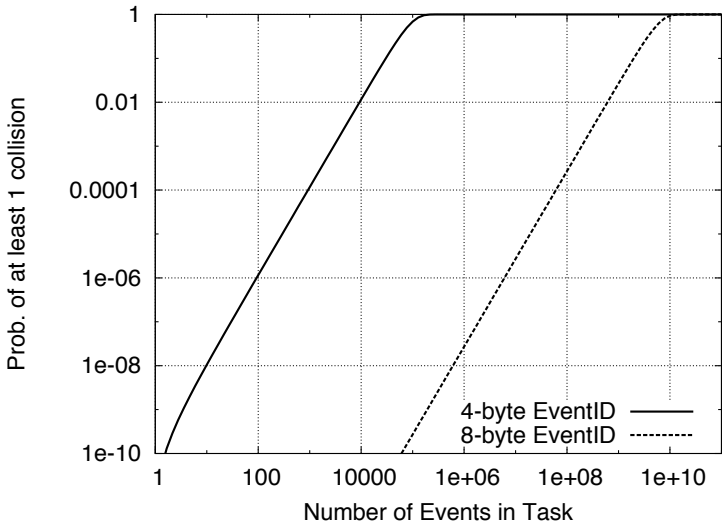


Figure 4.2: Approximate probability of at least one collision for the EventIDs *in the same task, versus the number of events in each task, for* EventIDs *of 4 and 8 bytes.*

Collisions in the selection of the *EventID* should be avoided within a single task only. These collisions are more problematic than the *TaskID* collisions, because they can introduce ambiguity in the causality graph. Although such ambiguities could be most likely solved by looking at the events timestamps, this is not ideal. To avoid collisions, one should set the size of the *EventID* identifier space according to the expected number of events in a task. The probability $p(n, d)$ of at least one collision for n identifiers chosen from a space of d bits can be approximated by $1 - e^{-n(n-1)/2^{d+1}}$ [51]. Figure 4.2 plots this probability versus the number of identifiers chosen (n), for two lengths of the *EventID*. The conclusion is that 4-byte *EventIDs* are only suitable if the expected task sizes are smaller than 100 or 1000 events. With 100 events and 4 bytes, it is expected that 1 in every $\sim 10^6$ tasks will have an internal collision, which can be deemed acceptable. With 1000 events per task, 1 in $\sim 10,000$ tasks will have a collision. However, if tasks have between 10,000

and 100,000 events, the collision probability is high: between $\sim$1% and $\sim$68% of tasks will have at least 1 collision, respectively. For 8-byte *EventIDs* the situation is much better, with tasks as large as 10^6 events having collisions in 1 out of $\sim 10^7$ tasks.

4.1.2 Destination Option

We have defined two options for the metadata, that affect report generation and aggregation. The specification is open, such that future options can be defined for new uses, such as carrying capabilities or signatures in the metadata.

The first of these options is the *destination* option, which allows a user to indicate interest in receiving information about the task in the wide-area. The example in Figure 3.7, in Section 3.2.5 shows how this option might be used.

The destination is specified as a pair of protocol and address, and we have currently defined the types listed on Table 4.1. The i3 destination type allows reports to be sent to an identifier in the Internet Indirection Infrastructure, an overlay routing architecture [155]. The OpenDHT destination type allows reports to be sent to the OpenDHT, a publicly accessible distributed hash table (DHT) service [141]. OpenDHT offers a hash table interface so that clients can store and later retrieve arbitrary information under specified keys. When this destination type is selected, nodes store reports on OpenDHT using the *TaskID* as the key, and the user can retrieve the reports later by fetching the value associated with the *TaskID*.

4.1.3 Severity Option

The other option we have defined is the *severity* option, which gives the developer and the administrator control over the granularity of tracing. X-Trace borrows the mechanism from traditional

Protocol	Destination
UDP	IPv4:port
TCP	IPv4:port
i3	I3 id
XMLRPC	OpenDHT key from *TaskID*

Table 4.1: Types of X-Trace report destinations.

Code	Severity
0	Emergency
1	Alert
2	Critical
3	Error
4	Warning
5	Notice
6	Informational
7	Debug

Table 4.2: X-Trace reporting severity levels

logging frameworks like the BSD Syslog protocol [103] and log4j [102].

Severity is a number from 0 to 7, representing increasing detail. The meanings are borrowed from Syslog, and are shown in Table 4.2. In traditional logging systems, such as Syslog and log4j [102], each log statement in the code has an associated severity (or level, in the case of log4j). The runtime logging system has, on the other hand, a global severity threshold which it uses to control the granularity of logging. Given a threshold t, the system will log a statement with associated severity s if $s \leq t$.

X-Trace uses a similar approach, but adds an important variation enabled by the X-Trace metadata. The metadata can carry a severity level for its *TaskID*, which has the effect of changing the severity threshold *for that TaskID only*. This enables specific tasks to be traced at greater detail.

The X-Trace runtime has two severity-related thresholds: *default* (t_d) and *maximum* (t_M). Each

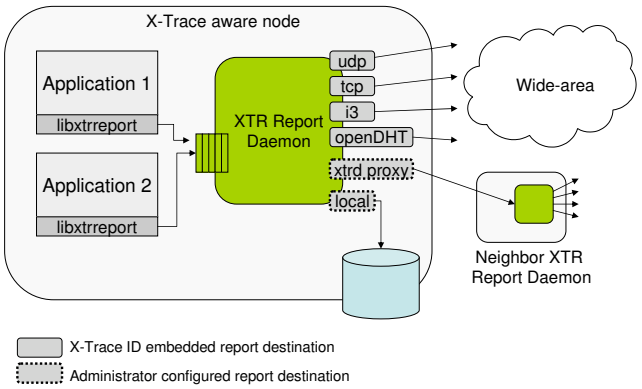


Figure 4.3: X-Trace reporting architecture.

event (akin to a log statement), has a (fixed) associated severity (s_e) value. Finally, each task, represented by its metadata, has an optional severity threshold t_t of its own. To see whether an event will be recorded, X-Trace does the following:

$$\text{severity threshold } t_s = \begin{cases} t_t & \text{if } t_t \text{ is present in metadata} \\ t_d & \text{otherwise} \end{cases}$$

$$\text{effective threshold } t_{eff} = \min(t_s, t_M)$$

$$\text{record event } if \quad s_e \leq t_{eff}$$

4.2 Reporting Architecture

We now describe a prototype reporting architecture that aggregates reports to a central location and allows post-mortem analysis and visualization of X-Trace task graphs. The architecture comprises a library against which applications link, a local daemon process that collects reports on a single machine, and a central repository that receives and stores the reports.

As described in Section 3.2.5, each event in an application generates a report with information about the task, the event, the immediately preceding events, and specific information about the

event. These reports are text-based sets of key-value pairs, in a format similar to the one defined in RFC 822 [38]. The format allows for arbitrary new keys to be added depending on the system being instrumented and the post-processing involved.

We implemented a library that can be linked into applications and software components that can generate properly formatted reports. In our prototype reporting infrastructure, this library also sends reports to a local *report daemon* that can perform further report forwarding.

The local daemon generally runs on the same machine as the instrumented application or component, but can also run in separate nodes. The communication between the X-Trace library and the daemon is non-blocking, and does not delay normal application processing. The daemon can be configured to store reports locally, forward all reports to a configured destination, or to forward reports to wide-area destinations if these are present in the destination option of the X-Trace metadata.

We also implemented a central database that can be used as the default report destination for reports from a given AD or network, much like site-local databases in Figure 3.7 from Section 3.2.5.

Figure 4.3 shows a typical X-Trace instrumented node, with the local X-Trace report daemon listening for reports from multiple applications. The daemon routes these reports appropriately.

4.2.1 Performance

To test the performance of the reporting infrastructure, we used the Apache web benchmarking tool, `ab`, against two otherwise identical Apache websites: one with reporting turned on and one without. The report store in this test was a separate Postgres database. Of the 10,000 requests we issued to the site, none of the reports were dropped by the reporting infrastructure. The regular server sustained 764 requests/sec, with a mean latency of 1.309 ms. The X-Trace enabled server

sustained 647 requests/sec, with mean latency of 1.544 ms, which shows a 15% decrease in total system throughput.

4.3 Integrating X-Trace

As X-Trace requires modifications to existing systems in some cases, it is key that the effort required to instrument systems be minimized. This section describes how we integrate X-Trace metadata into protocols, and how we perform propagation within applications. Integrating X-Trace into systems requires propagating X-Trace metadata through protocol messages, as well as within applications and software components. We added X-Trace to new and existing protocols and systems, and in the course of doing so implemented a set of libraries to assist the programmer with these tasks. We describe them in turn, providing examples, and describing the API that allows users to both propagate X-Trace metadata and generate X-Trace reports from different components. The API makes adding X-Trace annotations to applications akin to adding normal logs statements, with the proper propagation of X-Trace metadata being handled by our runtime library. We end the section by describing important challenges in the instrumentation, including the capturing of concurrency and propagation across structures that defer computation, such as queues.

4.3.1 Integrating into Protocols

Integrating X-Trace metadata into protocol messages involves carrying the metadata in the header of messages. This varies in difficulty and feasibility depending on the protocol. Adding support in new protocols is easy, as the format for X-Trace metadata is simple and well-specified. We currently use a compact binary representation or a textual representation of the hexadecimal

encoding of the metadata that uses two characters per byte, depending on the protocol. Other encodings appropriate for specific protocols can easily be devised.

For existing protocols, the feasibility depends on the protocol's extensibility. For example, it is simple for HTTP, because its specification [53] allows for extension headers, and dictates that unknown extensions be forwarded unmodified to next hops by proxies. Other protocols like SIP [143], Email [38], IP, TCP, and I3 share this characteristic.

For IP packets, adding options can cause packets to be dropped by some wide-area paths. In [57], we ran experiments sending IP packets with options on PlanetLab [22], and found that approximately 50% of paths dropped IP packets with any options. The positive finding, however, is that over 90% of these drops occur either at the source or at the destination AS, making IP options viable if the endpoint ASs support them.

For protocols without an extension mechanism, one has to resort to either changing the protocol, transparently adding metadata before or after protocol messages, or overloading some existing functionality. Another option is to add metadata to the protocol messages as they are sent, and remove this metadata on the other side before they are delivered to oblivious unmodified protocol code. In the implementation of Chord that comes with i3 citei3, we had to create a new type of message. For adding support to the Sun RPC protocol, we appended X-Trace metadata after the end of each message. While this is not a fully compatible solution, it worked in our tests even in environments where some elements were X-Trace aware and some weren't, because RPC messages were self-describing. Table 4.3 gives details on adding metadata to these and some other protocols.

We say that X-Trace is not supported by a protocol if adding X-Trace metadata to messages causes receivers to drop the messages.

Protocol	Metadata	Comment
HTTP, SIP, Email	Extension Header	Out-of-the box support for extensions.
IP	IP Option	Automatic propagation. Dropped by some ASs, wide-area support varies [57].
TCP	TCP Option	One-hop protocol, no next hop propagation. Linux kernel changes are needed.
I3	I3 Option	Support for options, but had to add handling code.
Chord	No support	Created separate message type.
SunRPC	Appended to message	Metadata invisible to unaware implementations.
DNS	EDNS0 OPT-RR	The EDNS0 [169] extension to DNS allows metadata to be added to messages.
PostgreSQL	SQL Comment	Possible to encode X-Trace metatada within a SQL comment.
UDP, Ethernet	No support	Must change protocol or use shim layer.

Table 4.3: Support for adding metadata to some protocols. We have implementations for the protocols in italics.

4.3.2 Integrating into Applications

Integrating X-Trace into applications and software components involves, as we described in Section 3.2, getting metadata from incoming messages, propagating the metadata internally, generating event reports, and setting metadata on the appropriate outgoing messages to other components. Our libraries make it straightforward to serialize and deserialize X-Trace metadata into protocol messages, and also facilitate the propagation through different runtime environments.

We implemented automated propagation of X-Trace metadata in two very different styles of runtime systems, *thread-based* and *event-based*, which we describe below. Interestingly, the interface X-Trace presents to the programmer, in both cases, is the same, as we show below.

Threaded Environments

A popular style of programming concurrent system is based on threads [23]. Using the terminology from Adya [1], these are programs with *preemptive multitasking* and *automatic stack management*. Computation is structured in threads, and each thread executes on behalf of only one task at a given time. Events in each thread are totally ordered, and the programming language maintains control flow information in the managed stack. These features make it simple to add X-Trace propagation to thread-based runtime environments.

Recall that when reporting an event, the event-logging code must have access to the immediately preceding event. The X-Trace library keeps a per-thread variable, globally accessible within the thread only, called `LastXtr`. `LastXtr` stores an X-Trace metadata, including the *TaskID* and *EventID* of the last recorded event. When an event is to be recorded, the logging code looks at `LastXtr` and adds an edge from its current value to the newly created *EventID*. `LastXtr` is then updated with the *EventID* of this new event, so that the following events can register the proper causality. Other edges can also be added to the event if necessary, coming from messages or data accesses, for example.

When used with multiple threads, each thread updates its own `LastXtr` variable. The one extension that is needed is for the thread libraries to be instrumented such that thread creation and thread synchronization primitives be properly recorded as events.

Event-Based Environments

An alternative to thread-based systems, popular on high-performance network servers and graphical user interface systems, is a programming model called *event-based* [124]. [1]

Again using the terminology from [1], programs in this style use *cooperative multitasking* and *manual stack management*. There is typically one thread per processor, each multiplexing several logical threads at the same time. There is an event loop that waits for an external event and starts the appropriate handler code when such an event happens. External events can be a timer firing, a disk read completing, a network packet arriving, or a user clicking a mouse, for example. Each handler runs uninterrupted for a short time in response to an external event, until it has to perform an operation that blocks waiting for another external event. When an event handler has to do a long operation, for example, read a file, it schedules the read, together with a *continuation*, which contains the handler to be called when the read completes, and optional contextual information for the new handler. In effect, each event handler does a deferred function call for a subsequent handler. Instead of using the stack for control flow, however, the programmer maintains the control flow in the chain of continuations stored in the heap, and hence the name manual stack management. Once the event handler returns control to the event loop, the system waits for the next event and calls the appropriate handler.

Our goal with X-Trace is to link causally related events in the logical threads that are multiplexed into one thread by the runtime. We can achieve that by keeping X-Trace metadata as the same per-thread global variable `LastXtr` during the execution of an event handler, but *storing the metadata with the continuation* whenever a deferred function call is scheduled. The event loop then has

[1] There is unfortunately an overload in the term *event* here. In event-based programming, the application responds to events, like network or user activity. X-Trace events, on the other hand, represent arbitrary operations in running systems.

the responsibility of restoring `LastXtr` from the continuation before transferring control to the handler.

We instrumented an event-based library for C++, libasync [111], for automatically propagating X-Trace metadata, an used this as a base for instrumenting two production systems built on libasync. Our instrumentation of the libasync runtime consisted of changing four functions used to schedule event handlers (callbacks, in libasync parlance), and part of the event loop that calls callbacks, totaling less than 20 lines of code. `timecb` is one of the scheduling functions, used to schedule a callback to be called in a given interval. To cause a function `f` to be called in `interval` seconds with argument `arg`, we write code like the following:

```
timecb_t cb = timecb(interval, wrap(f, arg))
```

`wrap` creates a callback object that encapsulates `f` and its arguments, and `timecb` actually schedules the callback to be called by the event loop. Our instrumentation adds a copy of `LastXtr` to `cb`. When the timer expires, the event loop sets `LastXtr` to the metadata stored at `cb`, and then calls f(arg).

The result is that inside an event handler, `LastXtr` behaves identically to its threaded-version counterpart, with information about the last preceding event in the current logical thread of execution. We use this similarity to expose the same API to programmers in both cases, as we describe next.

4.3.3 API

We instrumented a number of systems with X-Trace, and integrated automatic metadata propagation, as described above, in two representative implementations: one in the Java programming

language, and another in C++/libasync. The Java implementation follows the threaded-based model, while the libasync implementation follows the event-based model. A given computation can be expressed in both models [100, 170], and while the programming model is quite different, we were able to provide the same X-Trace Application Programming Interface for the programmer.

The X-Trace library removes from the programmer the burden of having to follow the logical thread of computation, as well as of having to worry about properly advancing `LastXtr` and creating a report. The library does the propagation by maintaining an X-Trace *context*, which contains the `LastXtr` variable. To add X-Trace to an application the programmer has the following tasks:

- When receiving a message, extract X-Trace metadata from the message, and set the X-Trace context: `xtr::Context::set(msg.getMetadata()).`
- When sending a causally related message, add X-Trace metadata to the message from the X-Trace context: `msg.setMetadata(xtr::Context::get()).`
- Whenever appropriate, log relevant events:

  `xtr::Context::logEvent(label, message).`

  This call advances the X-Trace context, creates a report, and sends it to the reporting infrastructure.

There is an alternative logging call that allows the programmer to add more information to the event, including causal edges from other events. The interface to the programmer is very similar to that of a logging API. In fact, X-Trace can be seen as a logging API that preserves the full causal relation among the logging statements.

4.3.4 Challenges

Extra work was required in two types of situations that were not correctly handled by the automatic propagation in `libasync` and in the RPC layers: capturing parallelism and propagating X-Trace metadata through application layer deferral structures, like queues.

Capturing Concurrency

We want to capture the concurrency among logical threads of computation, both in the case of event-based systems like `libasync`, and in the case of thread-based systems. This requires some extra steps in the case of the creation and termination of logical threads of computation.

The automatic propagation of X-Trace metadata described above is not sufficient to correctly capture the dependencies in the case of *barriers*, and currently the programmer instrumenting an application has to do some extra work. Figure 4.4 shows an example, in `libasync`, of a simple program that starts three logical threads (by calling `doSomething()`),[2] and then waits for all three to finish before logging the **end** event.

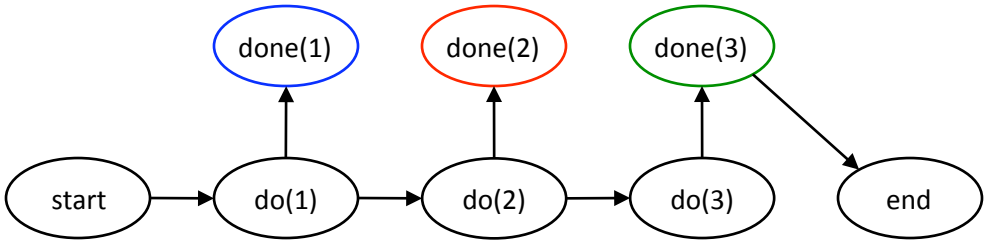
In some cases it should also be possible to statically annotate the code, to indicate that all execution paths starting from a given event should end in another event, and automatically infer the existence of the missing edges in post-processing. We leave this for future work.

From the figure, we observe that the **end** event is represented as only being dependent on the last **done** event. The graph would be different depending on the order of finishing of the three threads. The code implies that the **end** event in fact is dependent on all three **done** events. Another observation is that the **do** events are related to each other. While this is strictly true, from the sequence of operations in the code, it may not be what the programmer has in mind.

[2] In `libasync`, the equivalent of a thread is potentially created in every call that defers computation.

```
 1:  const int N = 3;
 2:
 3:  xtr::Context::logEvent("start");
 4:
 5:  for (i = 0; i < N; i++) {
 6:      xtr::Context::logEvent("do(%d)", i);
 7:      doSomething(i);
 8:  }
 9:
10:  ...
11:  int remaining = N;
12:  ...
13:
14:  void somethingDone(int i) {
15:      xtr::Context::logEvent("done(%d)",i);
16:      if ( --remaining == 0 )
17:         xtr::Context::logEvent("end");
18:  }
```

(a) Example C++ code with X-Trace logging calls.



(b) X-Trace graph resulting from the above code.

Figure 4.4: Example C++ code and resulting X-Trace graph. `doSomething()` *internally schedules concurrent processing, which eventually calls* `somethingDone()`*. This code only uses the automatic propagation of X-Trace metadata. Its capture of the forking is not intuitive, and the barrier is represented incorrectly.*

Figure 4.5 shows the same code, augmented using the X-Trace instrumentation API to correctly capture the concurrency. Recall that when logging an event, we advance the X-Trace metadata (*cf.* Section 4.3.3) to the event id just created. To represent that all three events are directly causally related to the **start** event, we keep the X-Trace metadata corresponding to the **start** event in a local variable, and reset the X-Trace Context to refer to it before each new fork. The representation of the fork is a matter of taste in this case, as the original one is correct. In thread-based code, this would be equivalent to passing to each thread, upon creation, the X-Trace metadata of the parent thread.

In the case of the barrier, the original representation is incorrect. The code in Figure 4.5(a) is augmented so that the first thread to finish creates a new event without immediately reporting (line 20). When the other threads finish, they add an incoming edge to the event (line 22), and the last one to finish generates the report for the event. Figure 4.5(b) shows the resulting graph. The equivalent instrumentation for thread-based code could be added to the specific calls required to create barriers, such as join. In this case, the threading library could store the X-Trace context of the joining threads and create an event with the appropriate incoming edges.

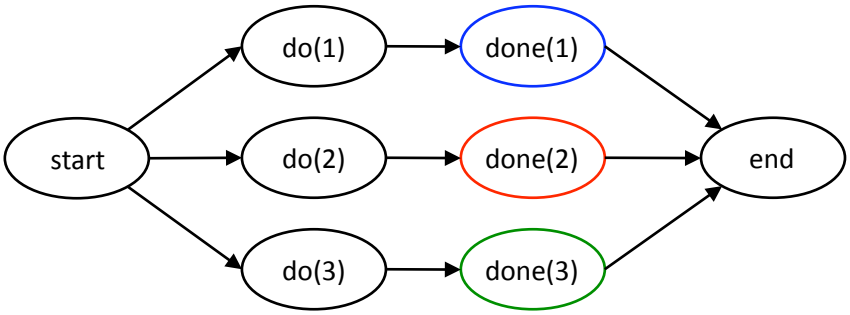
Deferral Structures

The second challenge was related to data structures in the applications that cause computation to be deferred. One example was an asynchronous DNS resolver in `libasync` that was shared by many tasks. The initial instrumentation was causing a number of tasks to end prematurely, while a few tasks were apparently much longer than they should be, as we can see in Figure 4.6(a). Closer inspection of the traces showed that the responses to the external DNS resolutions were all being attributed to the first task to instantiate the resolver.

To see why this was happening, we need to understand how the resolver works. The DNS re-

```
 1:  const int N = 3;
 2:
 3:  xtr::Context::logEvent("start");
 4:  xtr::Metadata startXtr = xtr::Context::get();
 5:
 6:  for (i = 0; i < N; i++) {
 7:      xtr::Context::logEvent("do(%d)", i);
 8:      xtr::Context::set(startXtr);
 9:      doSomething(i);
10:  }
11:
12:  ...
13:  int remaining = N;
14:  XtrEvent xte = null;
15:  ...
16:
17:  void somethingDone(int i) {
18:      xtr::Context::logEvent("done(%d)",i);
19:      if (xte == null)
20:         xte = xtr::Context::createEvent("end");
21:      else
22:         xte.addEdge(xtr::Context::get());
23:      if ( --remaining == 0 )
24:         xte.sendReport();
25:  }
```

(a) Same code as in Figure 4.4, with added X-Trace code for correctly capturing concurrency.



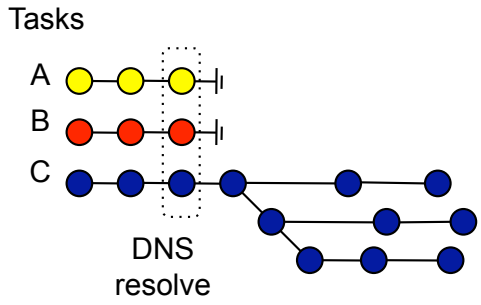
(b) Semantic concurrency captured by augmenting the instrumentation

Figure 4.5: Same code from Figure 4.4, augmented to capture the semantic *concurrency. The forks are more intuitive, and the barrier, correctly captured.*

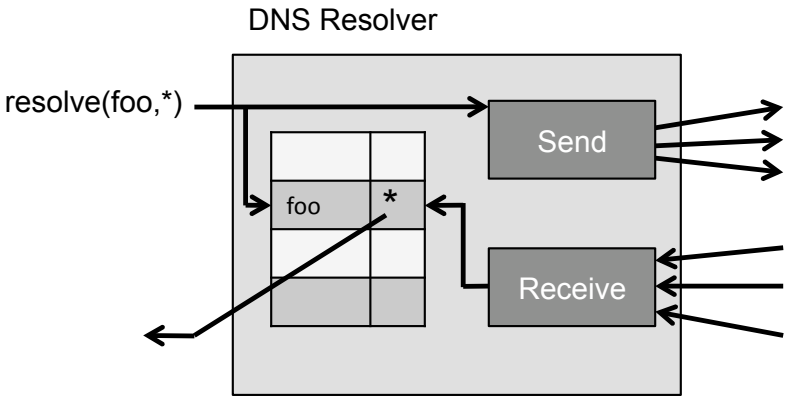
solver in `libasync` uses non-blocking I/O, and can be used by many clients at once. As illustrated in Figure 4.6(b), each request starts a resolution request but does not block the calling thread. Instead, the calling code registers a callback function to be called when the resolution completes. The callback is stored in a hash table, indexed by the DNS ID field of the generated request [116]. When a response comes back, the implementation looks the response's ID in the hash table and calls the corresponding callback function with the resolved name. The problem is that all arriving packets first cause the same callback function to be called: the function that parses the DNS response and looks the records up in the hash table. This callback is scheduled when the resolver is first initiated to respond to all incoming packets, and thus carries the X-Trace context of the task that first started the resolver.

We corrected this problem by storing the X-Trace metadata current when the request is made in the hash table, together with the callback function to be called when the resolution completes. The X-Trace metadata is then restored right before the callback function is called with the response.

This points out the general need to instrument data structures that represent deferrals in the computation. Another example of this is a queue that stores requests to be processed by one thread out of a thread pool, a pattern common in servers. In this case, we would have to instrument the queue by adding X-Trace metadata to the queue elements. In practice, we expect that most of these data structures and deferral mechanisms will be in libraries, such that one time modifications will be shared by many clients. The DNS resolver in `libasync`, for example, now correctly carries X-Trace metadata for its clients, and will work for all software that uses the library.



(a)



(b)

*Figure 4.6: Tracing through `libasync`'s DNS resolver. Initially (a), we observed some truncated tasks (A, B), with the corresponding missing events appearing in another task (C). This was due to the asynchronous DNS resolver in `libasync` (b), which uses a hash table to match responses with the corresponding callback. The * in the Figure represents the callback. We had to add X-Trace metadata to the callback stored at the table to correctly trace this case.*

4.4 Processing X-Trace Graphs

Instrumented applications and components generate reports for relevant events, and the reporting infrastructure enables the user to aggregate these reports. The user can then reconstruct the task graph, or the partial order of the execution, from these events. The presence of the edge information on the reports makes this reconstruction straightforward. Each event report is a node in the task graph, and each 'Edge' entry in a report from a preceding event constitutes a directed edge in the graph. It is useful, though, to extract structural information from the obtained graphs.

In this section we describe two algorithms for processing the task graphs. The first deals with identifying redundant edges (*cf.* Section 3.2.1) which are edges that abstract subgraphs of operations. The second shows how to obtain global timestamps (analogous to vector clocks [52, 109]) that are comparable between any two events in a task. Lastly, we describe our prototype visualization tool, based on the Graphviz package [46], that we implemented to examine the X-Trace graphs.

4.4.1 Finding Redundant Edges

As we described in Section 3.2.1, X-Trace task graphs can represent arbitrary subsets of the complete partial order graphs, *i.e.*, containing all the edges in the transitive reduction of the partial order, plus any redundant edges. Redundant edges are ones that can be removed without changing the reachability of the graph. In other words, an edge i, j is redundant if there is a path from i to j that does not use edge i, j. Determining the redundant edges is an important step in processing X-Trace graphs, as they can be useful in summarizing the graph, examining the parallel structure of the task, and providing more intuitive visualizations of the graph structure. To do this, we compute the transitive reduction of the graph.

X-Trace task graphs in practice have relatively few redundant edges, and are generally much closer to the transitive reduction of the partial order than to the transitive closure. Since they are relatively sparse, the most efficient representation for them is as an adjacency list, as opposed to a matrix representation.

X-Trace graphs are partial orders, which are directed acyclic graphs. Directed acyclic graphs have a unique transitive reduction [5]. The best known transitive reduction algorithm for general directed graphs uses boolean matrix multiplication, and has complexity $O(n^{2.376})$ [37]. For DAGs, we can do better. Goralčíková and Koubek described an algorithm for transitive reduction and closure, listed in Algorithm 1, that works in $O(|V| \cdot |E_{red}|)$, where E_{red} is set of edges in the transitive reduction [71]. In [148], Simon proposed an enhancement to this algorithm that works in $O(k \cdot |E_{red}|)$, where k is the number of chains in a chain decomposition of G (we describe chain decomposition in the next section). Finally, for some restricted classes of DAGs, the transitive reduction can be computed in time linear with the number of nodes and edges. For example, in [166] Valdes gives an $O(|V| + |E|)$ transitive reduction algorithm for Generalized Serial-Parallel (GSP) graphs. Unfortunately, there's no guarantee that X-Trace graphs are GSP graphs, and we have to resort to Simon's algorithm.

4.4.2 Generating Comparable Timestamps

The partial order obtained from a computation can be very useful for example for debugging, visualization, program understanding, fault detection, and performance analysis. Given a set of events, an operation that is central to many of these is the ability to compare two events from the execution, and determine if one happened before the other or if they are concurrent. The directed graph representation of X-Trace graphs has enough information to allow for this comparison. While

Algorithm 1 Transitive reduction (Goralćíková and Koubek [71])

Input: $G = (V, E)$

Output: E_{red}, the edges in the transitive reduction, and
$out^*(v)$, the set of all nodes reachable from v, $\forall v$.

```
1:  E_red ← ∅
2:  for v ∈ V, in reverse topological order
3:     out*(v) ← {v}
4:     for w ∈ out(v), in increasing topological order)
5:        if w ∉ out*(v)
6:           out*(v) ← out*(v) ∪ out*(w)
7:           E_red ← E_red ∪ {(v, w)}
8:        end if
9:     end for
10: end for
```

it allows uncoordinated, constant-time capturing of events by different processes, however, our representation is not efficient for precedence testing between two arbitrary events, as this can require traversing the graph for each comparison.

The most common approach for making precedence tests efficient is the use of comparable timestamps that exhibit Lamport's Strong Clock Condition [99]. For two event a and b, their timestamps $C(a)$ and $C(b)$, and a comparison function $<$ between two timestamps, the Strong Clock Condition says that

$$a \rightarrow b \iff C(a) < C(b),$$

where $\rightarrow$ is the happened-before relation.

The Vector Clock algorithm simultaneously discovered by Fidge and Mattern [52, 109] exhibits this property, with a constant-time comparison function. It has two disadvantages that make it unsuitable for our purposes, though. First, it requires that the number p of processes in the system be known a priori, and each timestamp is a vector of p components. Second, it requires that during

the computation the timestamps be carried by processes and messages, which presents scalability and overhead problems.

Agarwal and Garg's Chain Clocks [2], however, present the same properties as vector clocks, and don't require a predetermined number of processes. Further, chain clocks in most cases require less components than the number of processes. Next we show an offline procedure to compute chain clocks from X-Trace graphs, such that the size of the clocks is equal to the number of *chains* in a decomposition of the graph.

The algorithm consists of two steps. First, we decompose the X-Trace graph into k chains, and then assign a timestamp of at most k components to each event.

Chain decomposition

A *chain* is a subset of a partially ordered set of events $C \subseteq E$ that are all pairwise comparable. In other words, a chain is any *path* in the partial order graph. Similarly, an *antichain* is a subset of events $A \subseteq E$ such that all nodes in A are pairwise concurrent. The *width* of a partial order is the size of the maximum antichain.

A *chain decomposition* of a partial order P is a partition of the events of P into k chains, $C_1, \ldots, C_k$ ($\forall_{i \in 1 \ldots k} C_i \neq \emptyset$ and $\bigcup_{i \in 1 \ldots k} C_i = P$).

We can generate a chain decomposition that is optimal, thanks to a fundamental result in partial order theory, Dilworth's theorem, which says that width of a partial order is also the smallest number of chains into which it can be decomposed [42]. This decomposition can be obtained in $O(n^3)$ worst-case running time, using equivalent reductions to bipartite matching [84], to maximum flow [86], or algorithms proposed by Bogart and Magagnosc [65]. Ikiz and Garg present a comparison of these algorithms in [84]. While this decomposition will generate the smallest num-

ber of chains, and thus the smallest chain clock timestamps, it can be expensive. Since it has worse time complexity than transitive reduction, this algorithm is not useful to speed up the reduction as described above.

We can also derive a heuristic decomposition directly from DFS in $O(|N| + |E|)$ time. Jagadish [86] evaluated a few heuristics, and found the one derived from DFS (also described in [148]) to perform best, and acceptably close to optimal.

Algorithm 2 Assigning chain clock timestamps

Input: $\forall e, C(e) \in 1 \ldots k$, the chain of event e

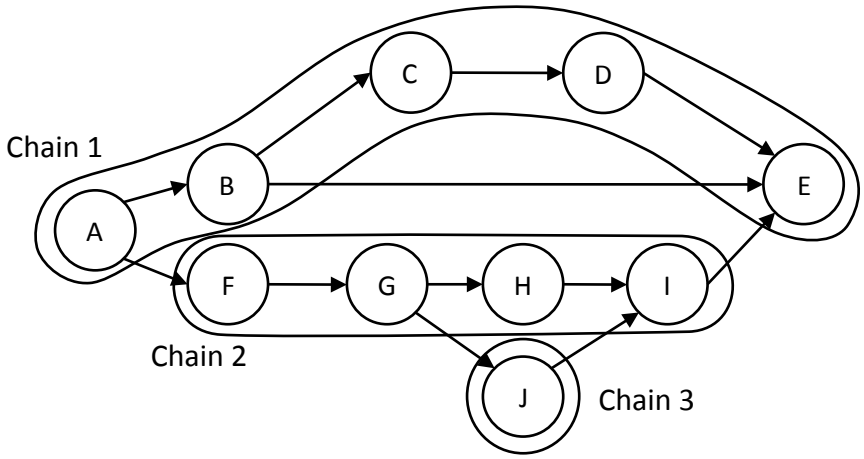
Output: $\forall e, V_e$, the chain clock timestamp for event e

```
1: for e ∈ V, in topological order
2:     V_e ← nil
3:     for p ∈ pred(e)
4:         V_e ← max(V_e, V_p)
5:     end for
6:     V_e[C(e)] + +
7: end for
```

Chain Clocks

Given a chain decomposition, we can derive chain clock timestamps for each event as given by Algorithm 2, adapted from [2]. The algorithm visits the nodes in topological order. The timestamps are represented by vectors of variable length, and a missing component is assumed to be 0 when doing component-wise comparisons. For each node, the algorigthm determines the component-wise maximum of all its predecessors, and then adds 1 to the component corresponding to the node's own chain. Because nodes are visited in topological order, it is guaranteed that the timestamps for all of a node's predecessors will be determined before the node is visited.

In line 6 of the algorithm, the $C(e)$th component of the chain clock is incremented. The vector



(a) Example graph with a chain decomposition.

Order	Node e	$C(e)$	V_e
1	A	1	[**1**]
2	B	1	[**2**]
3	C	1	[**3**]
4	D	1	[**4**]
5	F	2	[1,**1**]
6	G	2	[1,**2**]
7	H	2	[1,**3**]
8	J	3	[1,3,**1**]
9	I	2	[1,**4**,1]
10	E	1	[**5**,4,1]

(b) Chain clocks V_e as derived from Algorithm 2.

$A \rightarrow I$	$V_A = [1], V_I = [1, 4, 1]$	$V_A[1] \leq V_I[1] \wedge V_A[2] < V_I[2]$
$C \not\rightarrow G$	$V_C = [3], V_G = [1, 2]$	$V_C[1] \not\leq V_G[1] \wedge V_C[2] < V_G[2]$
$G \not\rightarrow G$		$V_G[2] \not\leq V_C[2] \wedge V_G[1] < V_C[1]$

(c) Two example comparisons between timestamps, according to Equation 4.1.

Figure 4.7: Derivation of chain clocks for an example task graph, given a topological order and a chain decomposition of the graph.

can grow as more components are added, and is padded with 0's for intermediate, non-incremented components. The chain clocks thus computed present the strong clock condition, and allow constant time comparison between any two events. The following lemma from [2] shows how to do the comparison:

$$\forall e, f \in P : e \rightarrow f \iff (V_e[C(e)] \leq V_f[C(e)]) \wedge (V_e[C(f)] < V_f[C(f)]) \tag{4.1}$$

Figure 4.7(a) shows an example task graph decomposed into chains, and the result of running Algorithm 2. In the table, the bold components in the chain clocks are the ones incremented for the given nodes. The Figure also shows, in (c), the comparison between two pairs of nodes, according to Equation 4.1. Node A precedes node I, but nodes C and G are concurrent, as neither precedes the other.

4.5 Visualization

There are many possible ways to visualize X-Trace graphs, and we describe a prototype tool we developed, using the Graphviz toolkit [46].

After a trace is completed, we use the processing steps described above to extract useful structural information about the graphs. The first step is to topologically sort the graph, using depth-first search. We then run DFS a second time, this time visiting the neighbors of each node in topological order, to perform a chain decomposition of the graph. This is equivalent to the algorithm for decomposition in [148], and produces longer (and fewer) chains.

We use the chain decomposition to run Simon's algorithm [148], a slight variation of Algorithm 1, to find the redundant edges. Lastly, for each redundant edge (u, v), we determine the

subgraph G_{uv} that it abstracts. Using the terminology from Section 4.4.1, we define

$$G_{uv} \triangleq out^*(v) - out^*(u).$$

After this pre-processing, our tool generates a Graphviz graph with the following features:

- each event is a node, and each edge captured in the trace is drawn between the nodes
- each node's outline color depends on the node's *chain*
- each redundant edge is drawn in *red*
- each subgraph G_{uv}, abstracted by a redundant edge (u, v), is drawn inside a box
- each edge that crosses machines is drawn *dashed*

Each node is positioned based on the logical clocks. Lastly, the color with which each node is filled can vary depending on other dimensions of the trace. For example, we have used different types or RPC calls and different hosts to visualize different trace characteristics.

Figure 4.8 shows an example of part of an X-Trace graph from the Coral CDN, generated by our visualization script and rendered by Graphviz, showing the various attributes captured by the trace. In the next two chapters we have a few examples of X-Trace-generated graphs, generated by DNS, web server, and overlay network instrumentations (Chapter 5), and by our instrumentations of Oasis and Coral (Chapter 6).

These graph are typically rendered and presented interactively to the user via a Java Applet [130], so that the user can zoom, pan, and get detailed information about each event. This is just a simple prototype, and many enhancements and alternative representations are possible, such as:

1. Collapse subgraphs upon clicking on the corresponding redundant edges,

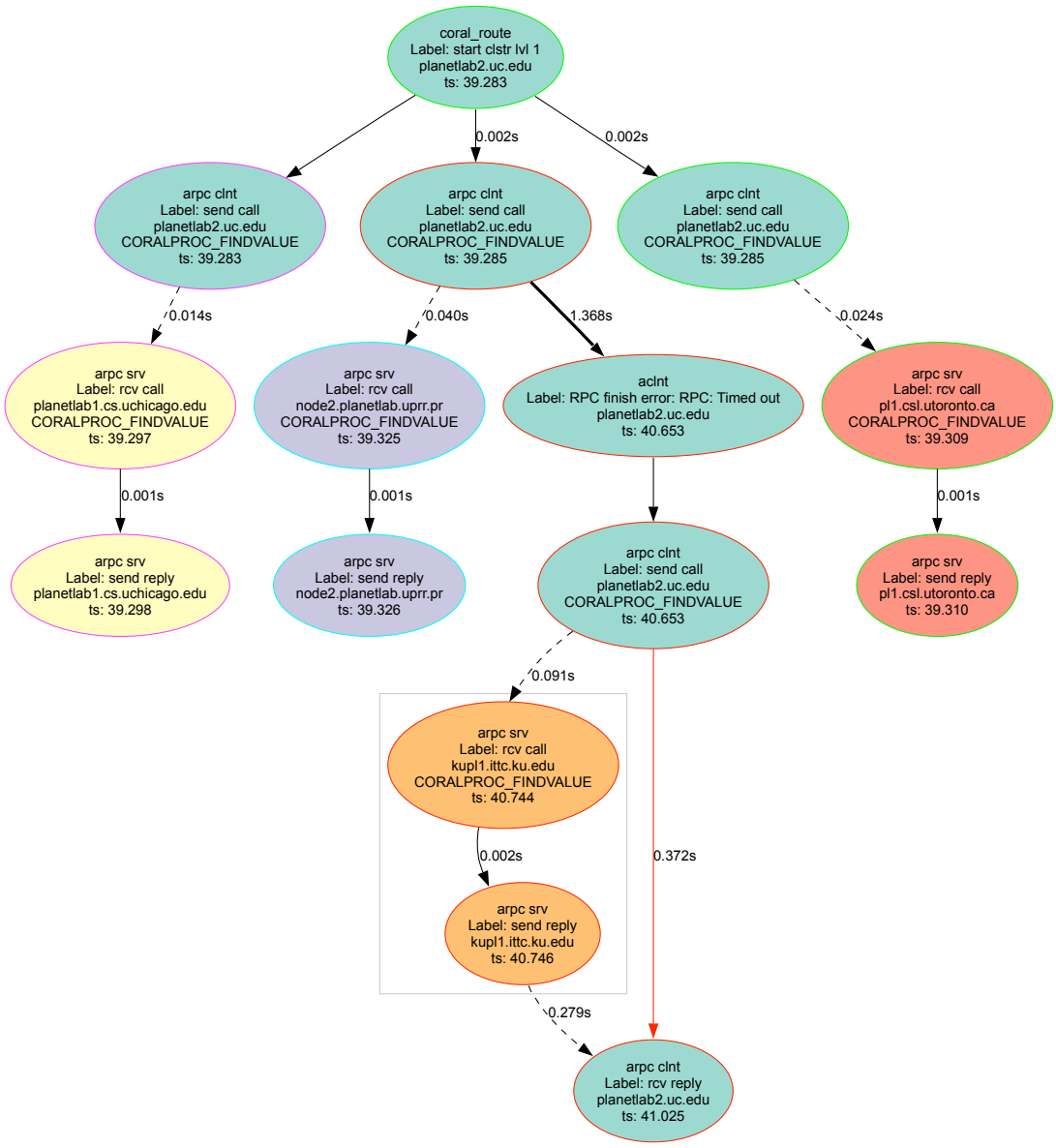


Figure 4.8: Example portion of X-Trace task graph from Coral as generated by our visualization script and Graphviz. For nodes, the fill color represents the machine, and the outline color, the sequential chain. For edges, the thickness is proportional to the time between events. Dashed edges span machines, and red edges are redundant. The node labels have the software agent, a message, the machine, and the time, in seconds.

2. Lay out nodes using wallclock time instead of logical time,

3. Lay out nodes using hosts as another dimension.

We believe that 1 will be very helpful for abstracting large subgraphs, and exposing higher level structures, while 2 and 3 may be useful for identifying timing problems in communication and execution.

In the next two chapters we provide a practical evaluation of X-Trace, by integrating it into a number of test and production systems. These examples provided experience in integrating X-Trace in many individual protocols, as well as new and existing applications. The X-Trace instrumentation API we presented in this chapter evolved from our experience, and proved easy to use by us and by other researchers. We also look at what kinds of faults we can find with X-Trace. In the test systems, we were able to detect injected faults, and in the production systems, Coral and Oasis, we were able to find a number of bugs and performance problems.

Chapter 5

Simple X-Trace Usage Scenarios

In this Chapter, we describe simple proof-of-concept scenarios we constructed to evaluate aspects of integrating and deploying X-Trace. In each scenario we discuss how X-Trace could be used to help identify faults. We discuss three examples in detail: a simple web request and accompanying recursive DNS queries in Section 5.1, a web hosting site in Section 5.2, and an overlay network in Section 5.3. We deployed these examples within a single administrative domain, and thus do not make use of the wide-area reporting mechanisms of X-Trace. We follow these examples with a description of other possible scenarios. The following chapter takes the lessons learned here and apply them to two production systems in the wide area instrumented with X-Trace.

The simple scenarios we examine exercise different aspects of X-Trace. The first one shows that it is possible to add X-Trace to the DNS infrastructure, which is naturally recursive, by using the same DNS extensions that are used in DNSSEC [12]. The second scenario has a web server behind a cache and a load balancer, running a database-backed PHP site, and represents a widespread set of components. Here we inject different faults and verify that we are able to distinguish the

causes using the resulting X-Trace graphs. The last scenario is not so common, but illustrates how X-Trace can integrate three different overlay network layers on top of the IP network. In this scenario we introduce different faults that show the same symptoms to the client, and use X-Trace to disambiguate them.

5.1 Web request and recursive DNS queries

5.1.1 Overview

The first scenario that we consider is a combined DNS and HTTP request by an end-user Web browser. This familiar task has two sequential phases, coordinated by the browser: the DNS lookup for the host IP address, and the HTTP request. Without X-Trace, tracing through these subtasks can be challenging, especially if one wants to correlate the two as part of the same higher level task. HTTP requests could be forwarded through proxies or caches, masking their ultimate destination. DNS requests are recursive in nature, are cached at intermediate servers, and span different administrative domains.

In this scenario, the Web browser requesting the DNS resolution is the only entity that has enough information to capture the causal relation between the DNS resolution and the subsequent HTTP request. Other approaches, such as intercepting traffic and using temporal heuristics, might succeed in finding the correlation, but cannot be guaranteed to always do so.

We implemented a simple HTTP client in Java, using the dnsjava library [172] for the DNS client and the HTTPClient library [165] for the HTTP part. To have control over all the components involved, we set up a parallel DNS hierarchy, independent of the global DNS infrastructure, with four DNS servers responsible, respectively, for the '.', 'xtrace.', 'berkeley.xtrace.', and

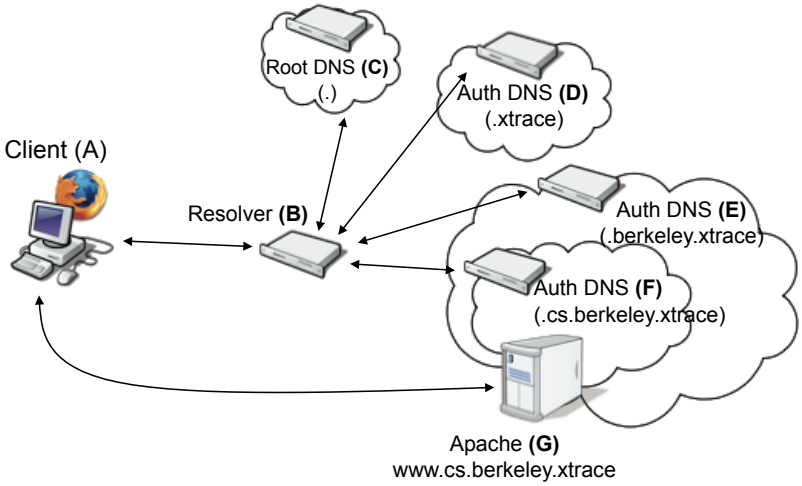


Figure 5.1: X-Trace scenario with recursive DNS and HTTP requests.

'cs.berkeley.xtrace.' domains. For each of these servers, we used a small authoritative DNS server coded in Java also using the dnsjava library. These were accessed through a recursive DNS resolver implemented in C++, part of the PowerDNS software suite [136]. Lastly, we used the Apache Web server running on a machine configured as 'www.cs.berkeley.xtrace.' to serve the requests. Our setup is shown schematically in Figure 5.1.

The user starts a request by typing a URL into her browser, in this case *http://www.cs.berkeley.-xtrace/index.html*. The browser's host first looks up the provided hostname using a nearby DNS resolver, which returns the IP address of that host (10.0.132.232). If the resolver does not have the requested address in its cache, it will recursively contact other DNS servers until a match is found. It can then issue the HTTP request to the resolved IP address.

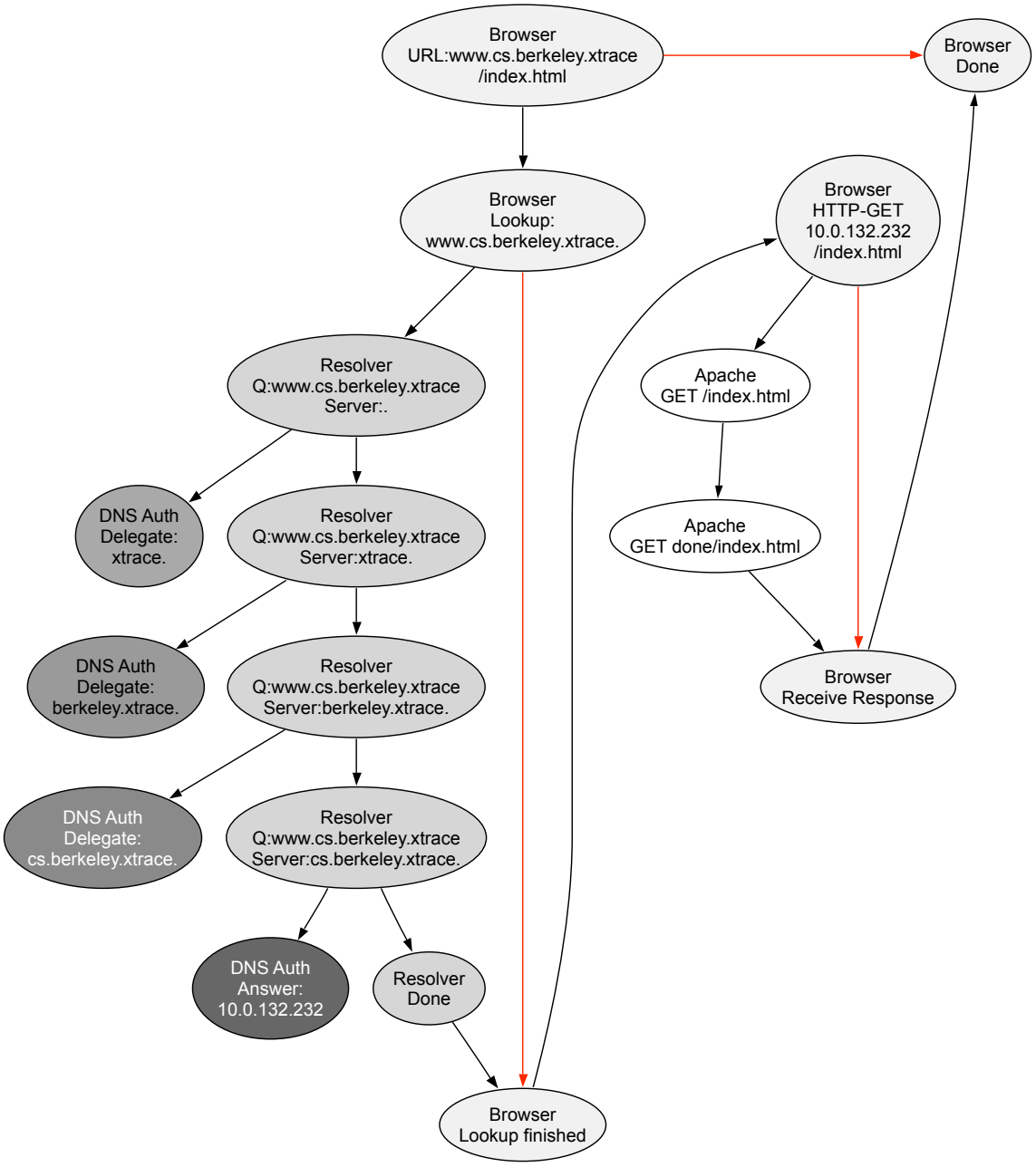


Figure 5.2: The complete X-Trace Task Graph reconstructed by X-Trace. Edges represent causality between two events. The red edges are redundant edges that abstract the respective subgraph.

Component	Implementation	Language	X-Trace			
			Metadata	Add	Propagate	Report
Client DNS Library	dnsjava [172]	Java	ENDS0 Opt.	•	•	•
Client HTTP Library	HTTPClient [165]	Java	HTTP Header Ext.	•	•	•
DNS Resolver	PowerDNS-recursor [136]	C++	EDNS0 Opt.		•	•
DNS Auth. Server	dnsjava	Java	EDNS0 Opt.			•
HTTP Server	Apache [61]	C/C++	HTTP Header Ext.		•	•

Table 5.1: Components instrumented with X-Trace in the DNS scenario. This examples involve adding X-Trace metadata to the DNS and HTTP protocols, and implementing propagation and reporting in the different components.

5.1.2 Adding X-Trace Instrumentation

We instrumented all of the components with X-Trace and issued requests to record the corresponding X-Trace task graph. Table 5.1 lists details of the instrumentation.

We added support for X-Trace to the DNS protocol by using the EDNS0 [169] extension mechanism. This backwards-compatible mechanism allows metadata to be associated with DNS messages, and is increasingly supported in the wide area. Specifically, EDSN0 allows one pseudo RR to be added to the additional data section of either a DNS request or a response, and optional data to be added to the record. We modified the DNS client, the DNS resolver, and the DNS server to add X-Trace support. The client adds X-Trace metadata to the outgoing requests, and reports two events: sending the request and receiving the response. The resolver propagates the X-Trace metadata to each request it generates, and reports when it sends the requests and when it finalizes processing. In our instrumentation the server only reported on receiving a request, but didn't propagate the metadata to the responses. This level of partial support by the server is enough to verify that it received a request.

After resolving the DNS name, the client reports the event and initiates the HTTP portion of

the transaction. We add the X-Trace metadata to the HTTP header as an extension field, which is interpreted by a C++ module we added to the Apache Web server, which we describe in more detail in the next scenario.

We deployed this software in our local testbed, and Figure 5.2 shows a complete X-Trace task graph produced by our visualization tool. The edges in the graph are causal edges between events. Note that the partial instrumentation of the DNS servers cause the corresponding reports to be leaves in the graph, having no outgoing edges back to the resolver events.

5.1.3 Fault Isolation

While we did not inject any faults in our tests of this scenario, an X-Trace enabled DNS infrastructure might uncover several faults that are difficult to diagnose today. At each step of the recursive resolution described above, servers cache entries to reduce load on the top-level servers. A misconfigured or buggy nameserver might cache these entries longer than it should. If a server's IP address changes, these out-of-date servers might return erroneous results. A trace like that in Figure 5.2 would pinpoint the server responsible for the faulty data.

Faults could occur in the HTTP portion of the task as well. We describe the application of X-Trace to web traffic in the following section.

5.2 A web application scenario

5.2.1 Overview

The second scenario that we consider is a web hosting service that allows users to post and share photographs. We deployed the Gallery open-source photo application [66] in our network on

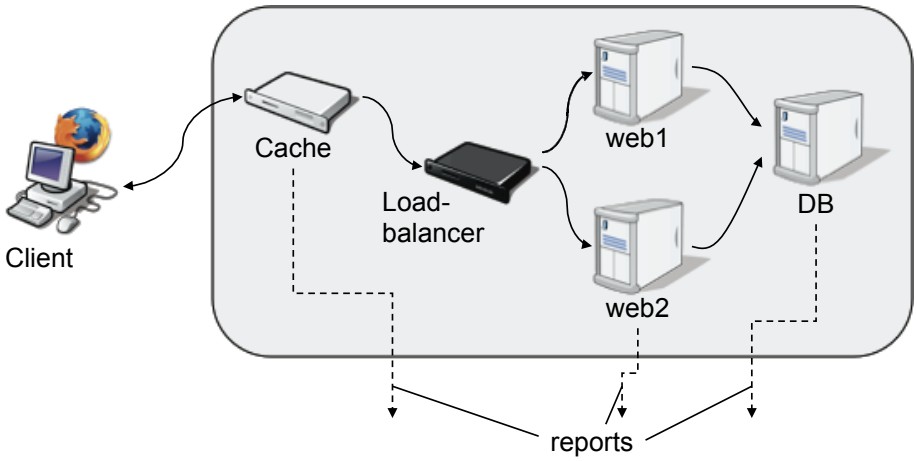


Figure 5.3: Architecture of the web hosting scenario.

an IBM Bladecenter small cluster. Our deployment included a frontend installation of the Apache Web server working as a cache, a load balancer, and two sets of application servers. Each application server was an Apache Web server with a PHP installation of the photo gallery, and both connected to the same backend PostgreSQL database. The photos were stored as files on the application servers' disks, and the metadata, album structure, and comments were stored in the database. Figure 5.3 shows the components just listed.

5.2.2 Adding X-Trace Instrumentation

For this site to support X-Trace, we implemented an X-Trace module for the Firefox browser, reused the Apache module described in the previous section for the cache and the Web server, and added an external reporting module for PostgreSQL. We also changed the PHP photo gallery to propagate X-Trace metadata via an SQL comment to the PostgreSQL database server.

Table 5.2 lists details of the instrumentation. Our module for Firefox adds X-Trace metadata to outgoing requests with either a fixed probability, or when chosen by the user. The metadata is

Component	Implementation	Instrumentation Language	X-Trace		
			Metadata	Add	Propagate	Report
Web Client	Firefox module	Javascript	HTTP Header Ext.	•	•	$*^a$
HTTP Proxy	Apache	C/C++	HTTP Header Ext.	$*^b$	•	•
Load Balancer	Hardware-based	n/a	HTTP Header Ext.		$*^c$	
HTTP Server	Apache	C/C++	HTTP Header Ext.		•	•
Gallery Application	Menalto Gallery [66]	PHP	CGI Env. Variable		•	
SQL Database	PostgreSQL	Perl[d]	SQL Comment			•[d]

[a] Manual reporting back to the web site when failure detected.
[b] Adds metadata when metadata not already set by the client.
[c] Passive propagation of unknown HTTP extension header field.
[d] External Perl program that tails the PostgreSQL log.

Table 5.2: Components instrumented with X-Trace in the Web application scenario. Some components only propagate metadata without reporting; the database only reports, and was not altered.

added as an extension header field in the HTTP request. The module doesn't send X-Trace regular X-Trace reports, but upon error the user can trigger special reports, which we describe below.

The Apache instrumentation module performs three functions:

1. Adds a new X-Trace header to a request if not already present

2. Issues reports for the request receiving and corresponding response sending events, which also have the effect of updating the event ID field of the metadata.

3. Forwarding the updated metadata to recursive requests.

The first function is useful for legacy clients that don't add X-Trace metadata to requests themselves. We did not change the hardware load balancer used in the scenario, but it passively forwarded the X-Trace metadata in the HTTP headers, since the HTTP specification dictates that implementations must forward unknown HTTP headers untouched.

Apache passes the X-Trace metadata to the PHP application via the CGI environment vari-

ables [142]. We changed the PHP database access layer to read the X-Trace metadata from the environment and add it to all outgoing database requests. We did not change the PostgreSQL client-server protocol, but instead added the metadata as comment in the SQL query sent to the database. We did not change the database server either. To collect X-Trace reports from it we wrote a simple Perl script to read the database query log, which preserved the comments embedded in the query. When the Web server generates the response, it includes the X-Trace metadata. Both the Apache at the application server, and the Apache at The Apache modules at both the application and cache servers generate reports of the response events, and propagate the metadata.

If any additional requests are generated because of the response (e.g., for images), the Firefox extension will use the same *TaskID*. For clients that don't support X-Trace, then each request (including images) will be considered independent.

Finally, on the response side, we implemented a novel reporting feature that leverages the X-Trace *TaskID* as a unique handle on the detailed execution of a request. Internal details of how a service are implemented are irrelevant to an end-user, and sometimes hidden for strategic reasons. However, in some situations there can be semantic failures that go unnoticed by low level automated checks by a site's operators, but can be easily detected by a user. An example might be stale content, like in the Wikipedia example of Chapter 3, or the failed addition of an item to a Web shopping cart.

Our Apache module adds an X-Trace header to the outgoing responses that contains the X-Trace metadata, as usual, but also contains the address of a Web form that can receive error reports. The browser extension module recognizes this information, and displays, on X-Trace enabled pages, a button that allows the user to report a problem. This button will generate an error report with the *TaskID* of the respective page, which will allow an operator to access the complete history of

that particular execution, without revealing any information to the user, or requiring any special knowledge from her. This mechanism is not necessary for all faults, since many requests might generate anomalous task graphs that can be analyzed with methods such as Pinpoint [31].

This scenario also that X-Trace is flexible as a tracing framework, as different components can exhibit different combinations of adding X-Trace metadata, propagating it, and generating reports, and that they can work together even when not all components perform all functions.

5.2.3 Fault Isolation

We introduced different faults to the photo hosting site, and used X-Trace to distinguish their causes. The first fault we consider is that of a malfunctioning PHP script on the front-end web servers. From the user's point of view, this could either be a fault in the PHP script, or a fault in the database. Figure 5.4 shows the task graph generated by the request. From the figure, we can conclude that the fault is the former: there are no reports from the database, pinpointing the problem to the PHP script. The square node in the figure is generated by a handler in the server that responds to a user-generated problem report. In addition to triggering an alarm for the operator, the report node indicates which page caused the problem, in this case, `/faults/query.php`, located on `web1`.

Next, based on the Wikipedia example, we implemented a web cache that inadvertently returns stale images from its cache. Diagnosis in this case is simple. The request trace includes nodes up to and including the cache, but does not include the origin server.

The last fault we consider in this scenario is that of a malfunctioning web load balancer, which sends traffic to a server that doesn't contain the appropriate content. When users request pages from the site, they will sometimes get the pages they wanted, while other times they will get `404`

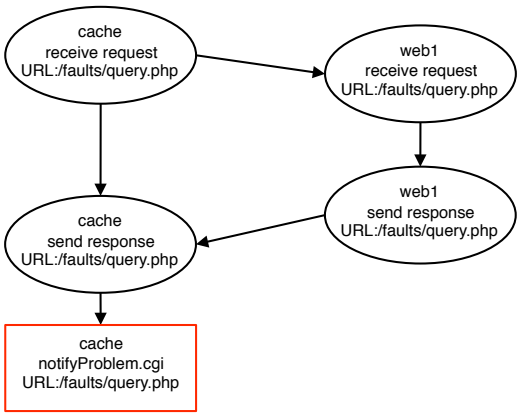


Figure 5.4: A request fault, annotated with user input.

`File Not Found` errors. In both cases, the load balancer issues a report with the request URL. Successful requests also include reports from the working web server and backend database, while unsuccessful requests only include a report from the web server.

5.3 An overlay network

5.3.1 Overview

The third scenario we look at in detail is an overlay network. This scenario is a proof-of-concept scenario to demonstrate X-Trace tracing across multiple layers. Overlay networks are routing infrastructures that create communication paths by stitching together more than one end-to-end path on top of the underlying IP network. Overlays have been built to provide multicast [82], reliability [8], telephony [143], and data storage [156] services. It is difficult to understand the behavior and diagnose faults in these systems, as there are no tools or common frameworks to allow tracing of data connections through them.

In our example, we use the I3 overlay network [155]. For our purposes, it suffices to say that

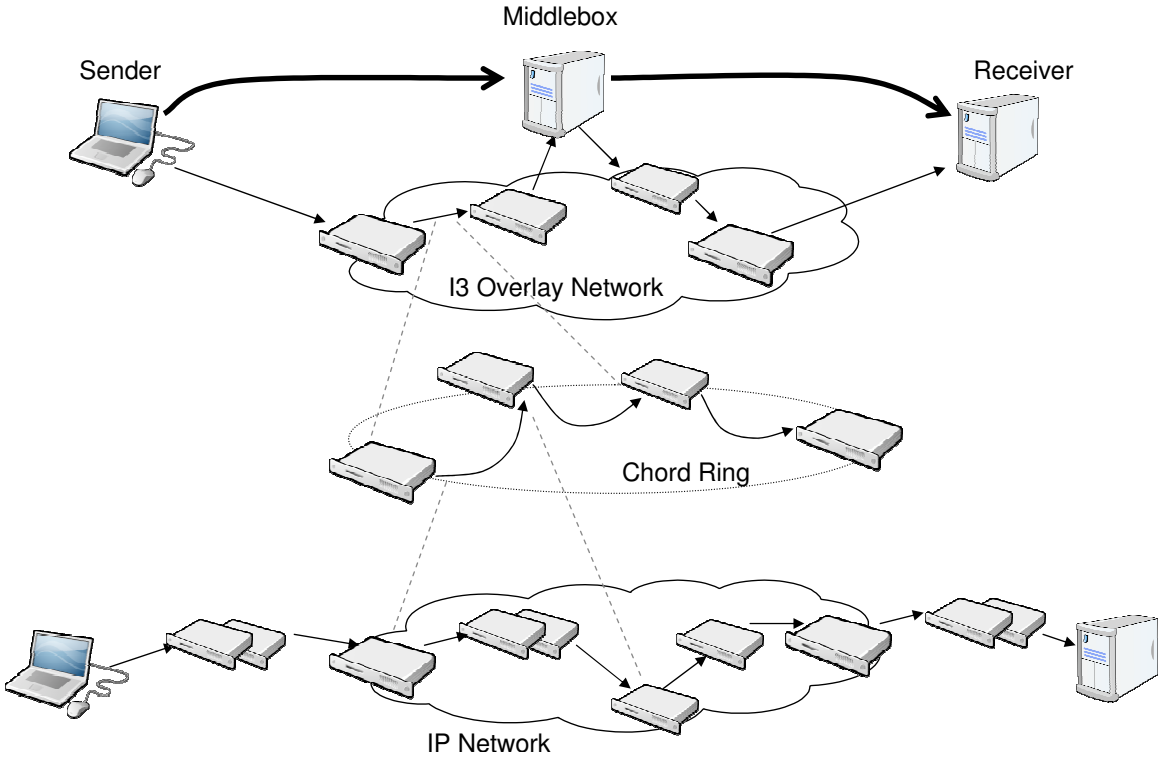


Figure 5.5: X-Trace on an I3 overlay scenario. A client and a server communicate over I3. Shown are the Chord network on top of which the I3 servers communicate, and the underlying IP network.

Component	Implementation	Language	X-Trace			
			Metadata	Add	Propagate	Report
Application	SNP Client/Server	C	SNP Header field[a]	•	•	•
I3 Overlay	I3 Reference Implementation	C	I3 Header option		•	•
Chord Overlay	I3 Reference Implementation	C	Chord Header field[b]		•	•

[a] Toy protocol we created for the example.
[b] We had to change the protocol to add the metadata.

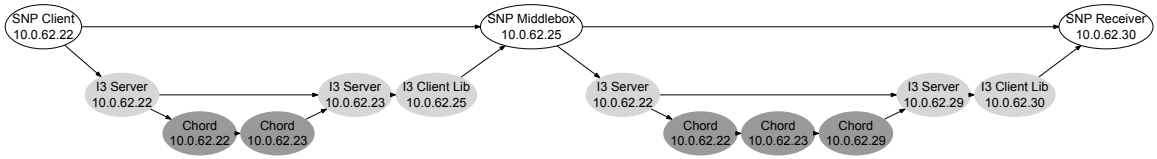
Table 5.3: Components instrumented with X-Trace in the overlay network scenario. We didn't instrument the IP layer, as all IP paths were 1 hop long.

I3 provides a clean way to implement service composition, by interposing middleboxes on the communication path. The implementation of I3 we used runs on top of the Chord DHT [156], which provides efficient routing to flat identifiers and is an overlay network on its own.

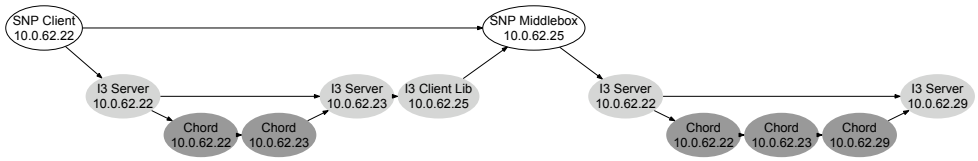
The scenario topology is shown in Figure 5.5, and consists, at the highest layer, of a very simple protocol involving a sender, a receiver, and a middlebox interposed in the path by the sender. We used a toy protocol we called SNP – Simple Number Protocol – that simply sends a number to the other party. The middlebox adds 10000 to any number it receives and forwards the request on, but it could also be, say, an HTTP proxy or a video transcoder. SNP also carries X-Trace metadata in its header. Each segment of the path in the SNP layer corresponds to a complete I3 path. Each I3 path, in turn, is formed by a combination of IP and Chord paths. Finally, each Chord path is formed by a combination of IP paths.

5.3.2 Adding X-Trace Instrumentation

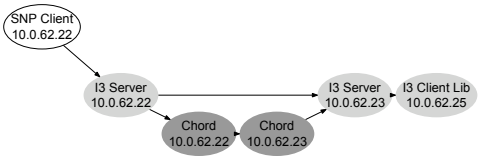
We added X-Trace metadata to the I3 and Chord protocols, code to perform the propagation operations, as well as calls to the X-Trace reporting library. Table 5.3 has some details on the instrumentation of these components. We deployed an I3 network consisting of 3 machines, each



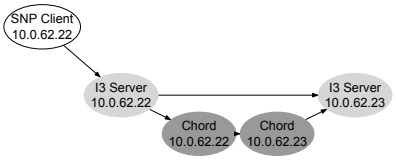
(a) Task graph for normal operation



(b) Fault 1: The receiver host fails



(c) Fault 2: Middlebox process crash



(d) Fault 3: The middlebox host fails

Figure 5.6: (a)X-Trace tree corresponding to the i3 example scenario with a sender, a receiver, and a sender-imposed middlebox. (b), (c) and (d) correspond respectively to faults: a receiver crash, a middlebox process crash, and a crash of the entire middlebox machine.

of which was also Chord node. The SNP client, receiver, and middlebox are on separate machines.

In Figure 5.6(a) we show the reconstructed X-Trace task graph from a sample run of the scenario. This graph was generated from X-Trace reports by our visualization tool. We didn't instrument the IP portions of the paths: since the machines were on the same switched LAN, all IP paths were one hop.

The SNP client sends a message to the the SNP receiver (see Figure 5.6), and it interposes the SNP middlebox on the path. The following is a detailed look at the transmission of a message in this scenario.

When the SNP client creates a message, it chooses a *TaskID*, creates an initial X-Trace report, and includes X-Trace metadata in the SNP header. It chooses the I3 identifier stack ($ID_{middlebox}$, ID_{server}) as the destination (an identifier stack is simply a source-routed path in I3). The client also copies the metadata into the I3 layer via an augmented I3 API function. The I3 client library further propagates the metadata to the Chord layer. The message is sent to the first I3 server, in this case at address 10.0.62.222. That server receives the message, and as it goes up the network stack, each layer generates and sends a report. The I3 server routes a message to the middlebox's I3 identifier, stored in the server 10.0.62.223. The I3 layer has a mapping between $ID_{middlebox}$ and the IP address 10.0.62.225. This message is delivered over IP to the I3 Client Library on that node, and then to the SNP Middlebox process.

The middlebox receives the message and processes it, sending a report from each of its layers. It removes its I3 address from the identifier stack, leaving only the address of the server, ID_{server}. Like the client, it propagates the X-Trace metadata to the lower layer and to the outgoing I3 message. The first I3 hop in this second application hop corresponds to a 2-hop Chord path. The process continues for the next I3 server, and finally the message is received by the receiver. At the receiver, we see a report from the I3 client library, and from the SNP application.

5.3.3 Fault Isolation

In Figures 5.6(b), (c), and (d) we injected different types of faults and show how the resulting X-Trace task graph detected them. We failed different components of the system that prevented the

receiver from receiving the message. Normally it would be difficult or impossible for the sender to differentiate between these faults.

Fault 1: The receiver host fails In Figure 5.6(b) we simulated a crash in the receiver machine. I3 expires the pointer to the receiver machine after a timeout, and the result is that the message gets to the last I3 server before the receiver, but there is no report from either the SNP Receiver or I3 Client library at the receiver machine.

Fault 2: The middlebox process fails In Figure 5.6(c) we simulated a bug in the middlebox that made it crash upon receiving a specific payload and prevented it from forwarding the message. We see here that there is a report from the I3 Client library in the third I3 report node, but no report from the SNP middlebox or from any part of the tree after that. This indicates that the node was functioning at the time the message arrived. However, the lack of a report from the middlebox, coupled with no reports thereafter, points to the middlebox as the failure.

Fault 3: The middlebox host fails Finally, in Figure 5.6(d), we completely crashed the middlebox process. I3 expired the pointer to the machine, and we see the message stop at the last I3 server before the middlebox. The lack of any reports from the middlebox node, as well as no reports from the second half of the graph indicates that the entire node has failed.

5.4 Additional X-Trace Uses

Here we describe, in much briefer form, other scenarios where X-Trace could be used. This list isn't meant to be exhaustive, merely illustrative.

Tunnels: IPv6 and VPNs A tunnel is a network mechanism in which one data connection is sent in the payload of another connection. Two common uses are IPv6 and Virtual Private Networks

(VPNs). Typically, it is not possible to trace a data path while it is in a tunnel. However, with X-Trace, the tunnel can be considered simply an additional layer. If the encapsulated protocol has X-Trace metadata, and that metadata is propagated to the encapsulating protocol, the tunnel itself will contain the X-Trace identifier needed to send trace data about the tunnel to the sender.

ISP Connectivity Troubleshooting For consumers connecting to the Internet via an ISP, diagnosing connectivity problems can be quite challenging. ISP technical support staff members have to spend time trying to determine the location of faults that prevent the user from successfully connecting. Complicating this process is the myriad of protocols necessary to bring the user online: DHCP, PPPoE, DNS, firewalls, NATs, and higher layer applications such as E-mail and web caches.

By including X-Trace software in the client, as well as X-Trace support in the equipment at the premises, the ISP can determine the extent to which the user's traffic entered the ISP. This could help quickly identify the location of the problem, and thus reduce support costs.

Link layer tracing An enterprise network might want to trace the link layer, especially if there are highly lossy links such as a wireless access network. The effect of faults in these networks can have a profound effect on higher layer protocols, especially TCP [16]. Retrofitting X-Trace into Ethernet is not straightforward, due to its lack of extensibility. However, X-Trace metadata can easily be stored in a shim layer above Ethernet, but below other protocols. Since all of the hosts on a LAN make use of the same LAN protocol, it would be possible to deploy X-Trace enabled network devices within one enterprise without requiring higher level changes.

Development Tracing tasks is needed at one point or another in the development of distributed applications and protocols for debugging and verification. Like with standard logging subsystems,

developers can integrate X-Trace into their applications. X-Trace was used by the team who developed DONA [94], a research prototype of a novel content-based routing scheme for the Internet.

In the next chapter we move from controlled, test scenarios in the lab to deployments of X-Trace instrumented systems in the wide-area. We had to cope with new challenges like scale, unpredictability, and failures, and were able to identify performance and correctness problems in the instrumented systems.

Chapter 6

Using X-Trace in the Wide Area

6.1 Introduction

In this chapter we look at a use of X-Trace in two wide-area production systems and show how we used it to find correctness and performance problems. We instrumented Coral [63], a public and open content distribution network, and OASIS [64], an anycast distributed system used for service selection. Both systems are deployed and open to public use, and run on PlanetLab [22], a global research network that allows researchers to test, deploy, and run wide-area network services. As of July 2008, 903 nodes from 461 sites were part of PlanetLab [131].

These wide-area deployments presented a number of challenges when compared to the local-area deployments. Among these are inherent challenges such as variable load, volatility and unpredictability of network characteristics. Sometimes nodes would be unreachable. In some cases X-Trace reports were lost, and the resulting X-Trace graphs were disconnected. The deployments were larger and live for a longer period of time, and X-Trace's ability to do correlated sampling allowed us to keep the reporting volume to a manageable level, given that we were using a single

machine to collect reports. We noticed persistent clock drifts and skews among some of the nodes, and X-Trace's deterministic causality allowed us to correct for those. We also verified significant variability in the executions of similar tasks, due to different requests, different network locations, and timeout and self-healing behavior in the two systems. We were able to find subtle, long-standing bugs in both Oasis and Coral that would have been difficult to identify with normal log-inspection techniques.

The chapter begins with a brief description of OASIS and Coral, in Section 6.1, with the necessary detail to understand the instrumentation. Section 6.2 describes how we instrumented the two systems, including the integration of X-Trace into the `libasync` event-based runtime library, and into `libasync`'s asynchronous implementation of SunRPC. Section 6.3 presents a number of interesting results we obtained from the deployment. In particular, in Section 6.3.1 we show two examples from Coral in which the same symptoms, as seen by the client, had very different causes. In Section 6.3.2 we present two case-studies, one from Oasis and one from Coral, in which we used X-Trace task graphs to identify actual bugs in the systems. We also briefly describe six other bugs we identified in Coral. Section 6.4 ends the chapter with some discussion of the results.

6.1.1 OASIS

In a replicated service, a client usually has a choice among several replicas to perform an action such as downloading a file or issuing a DNS query. A *good* replica may be one with low latency, small load, or one with high available bandwidth. An effective replica selection strategy can have significant impact on service times, load distribution, and service costs.

There are many approaches to this replica selection problem that generally involve a combination of active probing between the client and different network vantage points. OASIS, or Overlay-

based Anycast Service Infrastructure, is a server-selection infrastructure that can be shared by many services simultaneously. Each additional service contributes with probes that improve the accuracy of the system, while at the same time reduces the per replica cost of the probing. OASIS is deployed in PlanetLab and is shared by about 10 different services.

OASIS has a set of core nodes which help clients select appropriate replicas of a given service. There are relatively few core nodes, which are shared among all services. Each service's replicas report load and liveness data to the core nodes, and help these with measurement probes. Data about each service is aggregated in a subset of the core nodes, selected by consistent hashing. These core nodes are called *rendezvous* servers for the particular service. Clients don't need to be modified, and interact with core nodes via DNS and/or HTTP interfaces. While a full description of OASIS is beyond the scope of this dissertation, we briefly describe how a client can find a replica using OASIS via DNS resolutions.

A client can find a replica of a service `s` by resolving the DNS name `s.nyuld.net`, where `nyuld.net` is a domain name assigned to OASIS core nodes. A typical DNS resolution proceeds as follows. The client's DNS resolver queries the DNS hierarchy, which directs it to one of OASIS core nodes. When looking for a service for a client, a core node first maps the client's IP address to approximate geographical coordinates, and then queries one of the rendezvous servers for `s` for a replica of `s` close to the client. The actual process has many more details, and we refer the reader to [64] or [62] for the complete steps.

Regular clients can also use OASIS by HTTP redirection mechanisms, and there is a richer RPC interface for OASIS-aware clients. From the tracing point of view, this is an interesting system. The system architecture is highly modular and uses a number of protocols. The resolution outlined

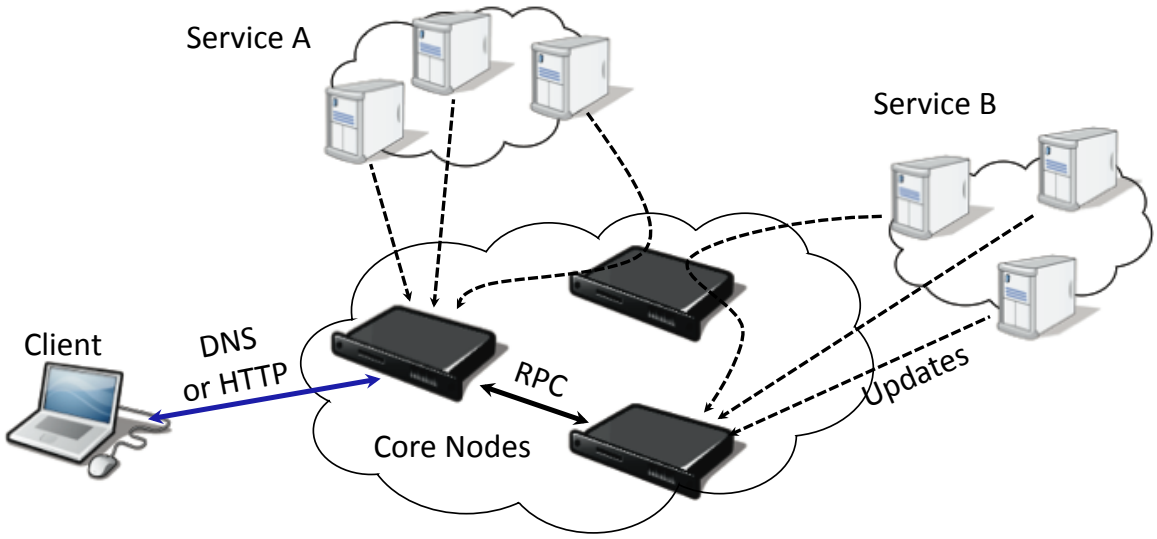


Figure 6.1: Overview of OASIS. Replicas of different services update information about themselves and measurements to the OASIS core nodes. Clients query core nodes via DNS, HTTP, or RPC interfaces. Some queries may result in redirects and can generate multiple recursive RPC calls among core nodes.

above, for example, involves a DNS server, RPC calls between OASIS modules in the same core node, and, potentially, RPC calls between different core nodes.

OASIS is programmed in C++, in an event-driven model. It uses the `libasync` library for managing the event control flow and asynchronous I/O, and the associated asynchronous, event-based implementation of the SunRPC protocol [151]. Traditional debugging tools are harder to use in such an environment, both because of the loosely-coupled nature of the modules, which run on different processes and different machines, but also because of the programming style, since the processor call stack only provides a very narrow window on the control flow, specifically only relevant to the current event-handler.

6.1.2 Coral

Coral [63] is a public and open distributed content distribution network that runs on PlanetLab [22]. Coral is composed of several infrastructure nodes that serve as a large shared cache for Web content. These nodes form a variant of a Distributed Hash Table [137, 144, 157, 181] called Distributed Sloppy Hash Table (DSHT). A DHT implements a put-get interface for key-value pairs. Participating nodes select a random ID in a large identifier space, and keys are also taken from the same space. A node with the *closest* ID to a given key is responsible for storing puts with that key. The definition of closest varies for different DHT proposals. A DSHT weakens the semantics of a DHT, allowing key-value pairs to be stored in nodes other than the root for a key. This allows a load distribution for popular content on a tree rooted at the root for the key. Coral's DSHT is based on the Kademlia DHT[110].

Coral uses the DSHT to store mappings from the URL to the nodes that have that URL's content. To preserve locality, Coral nodes form hierarchical clusters, so that ideally a client A searching for

content from server S will not have to query a node in Coral much further from A than S.

Users access Coral by sending modified URLs for content to one of Coral's infrastructure nodes. This is done by appending a domain name controlled by the Coral network to the domain name in the original URL. For example, to access a cached copy of the URL

`www.berkeley.edu/somepage`,

the user replaces it with

`www.berkeley.edu.nyud.net/somepage`.

The domain `nyud.net` is resolved by DNS to a Coral node, which responds to the client.

If the Coral node has the content in its local cache, it returns the page to the client. Otherwise, it queries the DSHT to find out a subset of the nodes that have the content. This lookup may involve multiple RPC calls between the initiator and successive Coral nodes. If some Coral nodes are found to have the content, the initiator issues an HTTP request to one or more of these. If no nodes are found, or if these secondary requests fail, the initiator node contacts the origin server (`www.berkeley.edu` in this case), for the content, and finally returns to the client. The initiator also locally stores a copy of the content, and registers in the DSHT the fact that it now has such a copy. Figure 6.2 shows a simplified version of this process, and we refer the reader to [63] for the full details.

Similarly to Oasis, Coral is also implemented in C++, using the event-based asynchronous I/O `libasync` library, and its associated asynchronous RPC implementation.

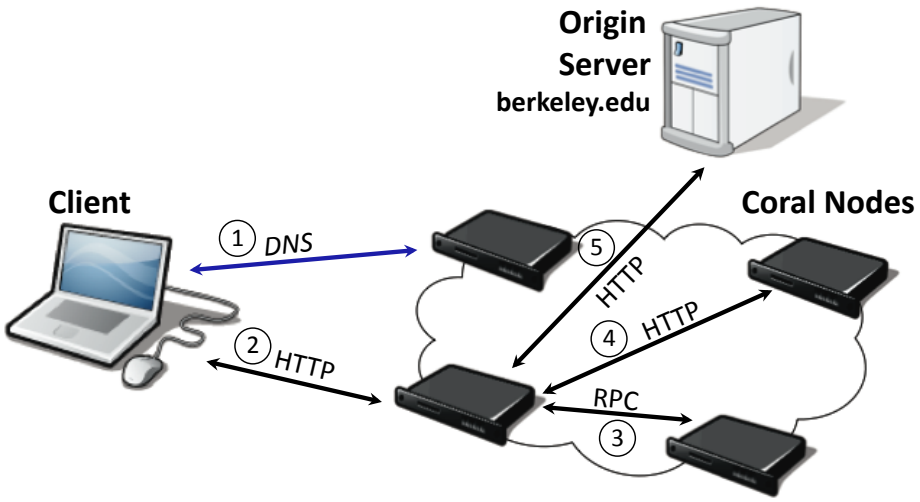


Figure 6.2: Overview of a Coral request. The client first uses DNS to resolve a modified URL (1), and issues an HTTP request to a Coral node (2). The node may respond to the request locally, or query the DSHT for other Coral nodes who may have the content (3). It can then request the content from one of these nodes (4), or, if all else fails, request the content directly from the origin server (5).

6.2 Instrumentation

We instrumented both OASIS and Coral with X-Trace to obtain traces of the user-initiated tasks. For OASIS we instrumented HTTP and DNS queries, and the resulting processing across core nodes. For Coral, we instrumented the HTTP requests by clients. The instrumentation leverages the X-Trace C++ libraries for handling X-Trace metadata, and is divided in three levels: the instrumentation of the `libasync` runtime, the RPC implementation, and of the applications themselves.

We divide `libasync`'s instrumentation into its core event-loop, the RPC library, and the DNS resolver. These components are shared among all clients of the library. Coral and OASIS refer to the instrumentation done in the application code. The instrumentation is also divided in two categories, propagation and reporting. Propagation includes the code necessary for correctly tracking the computation within the component, including, when necessary, extracting and storing X-Trace metadata from and to messages and data structures. Reporting accounts for the X-Trace code that actually generates event reports.

6.2.1 `libasync`

First, we integrated X-Trace into the `libasync` library, as described in Section 4.3 for event-based systems. The integration with `libasync` implements the automatic carrying of X-Trace metadata such that the programmer can always access information about the last event in the current logical thread of execution. This happens transparently across event handler scheduling and dispatch. The instrumentation implements the API in Section 4.3, allowing the programmer to start tracing a task, logging events, and extracting the current X-Trace metadata to add to messages.

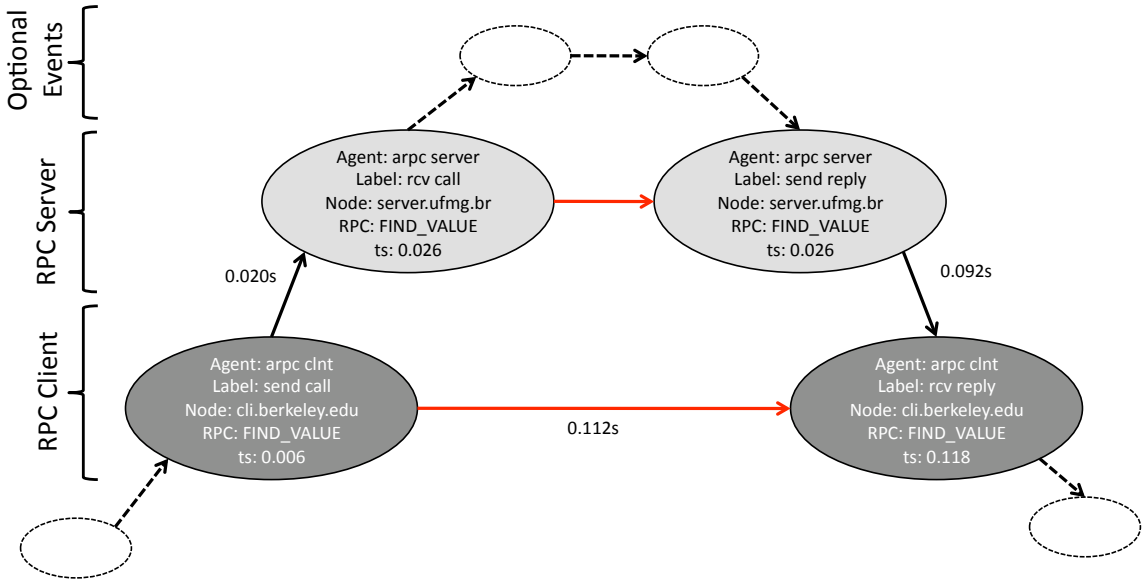


Figure 6.3: X-Trace task graph for an RPC call. The darker events happen at the client, and the lighter events, at the server. The red horizontal edges are redundant edges in the graph, and abstract the corresponding subgraphs. The time, in seconds, is show for each edge.

6.2.2 RPC

Leveraging the core `libasync` instrumentation, we added X-Trace instrumentation to the asynchronous implementation of SunRPC in the `libasync` library. In the normal case, the instrumentation will automatically add four events to the task, if a task is already being logged: client send, server receive, server send, and client receive. In the client side, the RPC instrumentation is automatically activated if the information about the previous event exists in the runtime when the RPC call is made. In other words, if the programmer started tracing some task prior to invoking an RPC, the RPC will generate tracing information automatically.

The implementation then transparently encodes X-Trace metadata in the RPC message. If the server is also instrumented, it will generate a new event when the call is received, set the X-Trace last event before calling the actual remote function called, and generate another event when sending the reply. As the X-Trace context is set before the remote function is called, any events logged by the remote execution will be correctly linked.

The server will then add the correct X-Trace metadata to the response, which the client will log upon receiving. Redundant edges are added between the two events at the client and the two events at the server. If the server is not instrumented we can still get the client-side information. Analogously, if the server function executing the remote call is not instrumented, we can still get the server-side RPC level trace.

Figure 6.3 shows a portion of a task graph, as produced by our visualization tool, with the four events generated by the automatic RPC instrumentation. The red horizontal edges show the redundant edges that both guarantee continuity in the trace and serve as explicit abstraction indicators.

6.2.3 Applications

With the `libasync` and RPC instrumentation in place, adding X-Trace to both Coral and OASIS was made easier. In our experience, adding X-Trace instrumentation to the two applications, in the common case, required comparable effort from the programmer as when adding traditional logging statements.

For OASIS, we instrumented the HTTP and DNS interfaces, creating a task for each user initiated request. For Coral, we instrumented the HTTP interface, also creating a new task for each user initiated request. In both cases, the X-Trace metadata is added (a new task is created) with a certain sampling probability at these initial points. Also, if the incoming message already has

X-Trace metadata, we preserve it instead of creating a new one. Coral uses the same interface for HTTP request between Coral nodes, when performing peer HTTP fetches to fulfill a user request. In these cases, instead of creating a new task, we added X-Trace metadata to the HTTP requests and responses, so that the events in the recursive request could be linked to the same original task.

6.3 Experience

We deployed our instrumented versions of Coral and OASIS in the production network on PlanetLab, and report some of our findings here.

At the time of deployment, Coral had already been running for over three years on PlanetLab, and OASIS for over one year, and there were no obvious bugs unknown to the developer. The analysis we present here is based on a portion of the X-Trace logs we collected from these systems for short periods of time. We try to highlight interesting aspects of what X-Trace enables a developer or operator to examine in such systems. Table 6.1 lists some key numbers from the collected data about the two systems. In our collection, we had an X-Trace reporting daemon running on each of the PlanetLab machines, relaying reports to a central database located in Berkeley.

We focus our attention on two aspects. First, we look at two examples of similar symptoms observed from one vantage point that have different underlying causes: RPCs with the same execution time may be delayed at the server or by the communication channel; RPCs that timeout may do so because the timeout value was too short, or because there was indeed a failure; and Coral, when responding to a user request, may be delayed by a number of different causes. What these examples have in common is that it is hard to distinguish the causes of the symptoms without correlating events across nodes.

	OASIS	Coral
Period (days)	1	2.5 days
Fraction	100%	0.1%
Hosts	19	258
Tasks	223,961	44,133
Reports	3,559,730	859,311
Edges	4,197,394	1,018,968
Bytes (raw)	783.3MB	235.6MB
Bytes (comp)	81.7MB	27.1MB

Table 6.1: Basic data for the OASIS and Coral traces analyzed in this chapter. The second row shows the fraction of initiated tasks that was instrumented.

Next we look at subtle bugs we found in both applications by analyzing X-Trace task graphs.

6.3.1 Same symptoms, different causes

X-Trace correlates events that happen in different processes on the same machine, on different machines, and across software layers, making it easier to differentiate the causes of problems with the same symptoms.

RPC Timeouts

An aspect that can be examined with X-Trace is the cause of timeouts in the client of RPC calls. Often, in distributed systems, developers will want to bound the time to perform some operation that involves remote communication by using timeouts.

This is fairly common in the two applications we examined here, Coral and OASIS. RPCs can be called with a specified timeout, after which the client declares an error in the call and moves on. It is sometimes better for the overall response time to retry the call on another server than to wait indefinitely. Setting timeout values elicits a tradeoff between failure rate and worst-case response

time, especially if the service time distribution is heavy tailed.

However, exploring this tradeoff after the fact is very hard with single-node logging techniques, because it requires correlating the execution at two different nodes. When a timeout occurs, the client node does not receive a response from the server. To determine if the timeout value is too short, it is useful to know whether the server completed the call, and what the distribution of completion times is, even when the client times out.

For analyzing this, we define four time intervals in an RPC call: T1 is the time between the client sending the request and the server receiving it. Ts is the time the server takes to serve the request. T2 is the time between the server sending the response and the client receiving it. Finally, Tc is the time between the client sending the request and receiving the response. In the `libasync` RPC implementation, when a client sends an RPC call, it will either receive a response, in time Tc, or declare a timeout, but not both.

Figure 6.4 has data for 34,518 executions of a particular Coral RPC call, FINDNODE, which is used by Coral when storing data on the DSHT. The graph has two CDF curves. The first, for 32,442 calls that completed normally, is the CDF of the client time, Tc. The second, for 2,076 calls that completed execution in the server but were timed out by the client, is the CDF of (T1 + Ts), and is a lower bound on how much time the call would have taken. Coral estimates the timeout values for this call dynamically, based on running statistics on the round trip time from the client to the server, but places an upper bound of 4 seconds. The graph also shows, for each point in the timeout CDF, the *corresponding timeout value recorded by the client*. These are the dots scattered around the lower CDF.

There are two situations: if a timeout dot is to the left of the green curve, it means that Coral

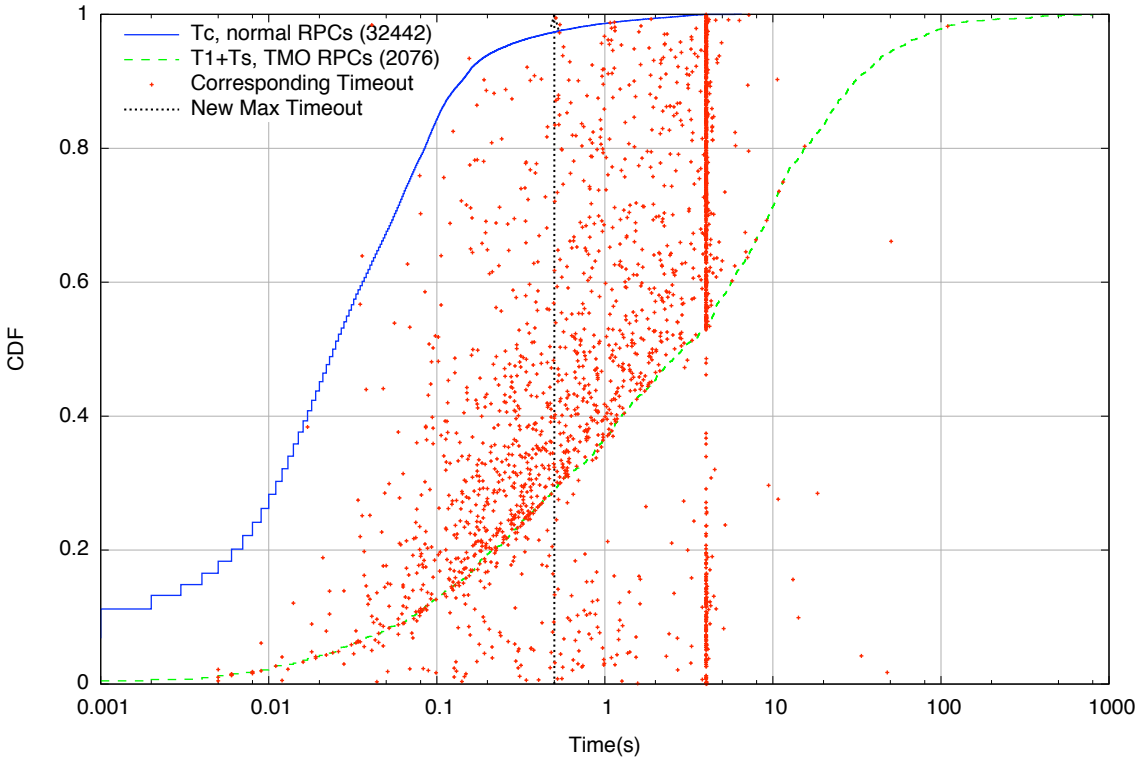


Figure 6.4: CDF of completed and timed out Coral FINDNODE RPC call. The blue solid curve is the CDF for the completion time of finished RPCs. The green dashed curve is the CDF of the client to server network time plus the server execution time, for RPCs that timed out. For each timed out RPC, a red dot at the same y value shows the corresponding timeout value at the client. If the timeout value is to the left of the green curve, Coral estimated that the call was taking too long. If the timeout value is to the right of the curve, the call completed but didn't reach the client.

estimated that the call should take less time that it took to even execute, and gives up waiting. This decision reduces latency, as the node probably retries. If a timeout dot is to the right of the green curve, it means that the call completed earlier than the Coral timeout, and for some reason did not reach the client. So Coral gives up waiting for messages that would probably never arrive.

From the figure we can clearly see the 4-second maximum timeout. The graph also gives some information about how to set the maximum timeout. For example, if we set the maximum timeout to 0.5 (also shown in Figure 6.4, we would see a reduction of only 2.6% in the number of completed calls (blue curve), for an 8x reduction on the maximum timeout. This reduction would certainly decrease wasted wait time for calls that don't complete (the calls that timeout and for which the red to is to right of the CDF). From this dataset, we observed that these comprise about 20% of the timed out calls.

Coral Response Time

The graph in Figure 6.5 shows, for 20,000 requests to Coral, the time Coral took to process the request versus the size of the requested object. The colors in the graph separate the requests by the class of the response code returned. The object sizes span more than five orders of magnitude, and it is striking that for a given size, the processing time spans up to more than seven orders of magnitude.

We can obtain the data for this graph without X-Trace, by just logging this information in the Coral node that receives the client request and ultimately generates the response. It is very hard, however, for an operator to distinguish the underlying causes for these times from this data alone, especially when more than one Coral node is involved.

X-Trace can provide much more in-depth information about specific executions. To illustrate

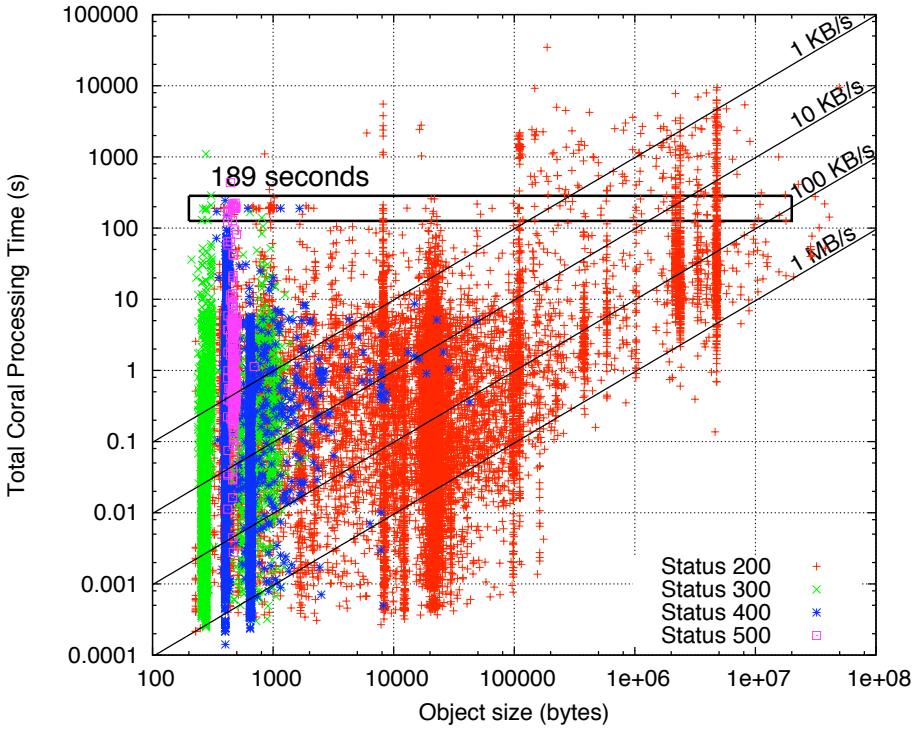


Figure 6.5: Coral processing time versus object size for 20,000 requests. Several different problems may have the same causes.

this point, in this section we focus on a set of requests that took close to 189 seconds to complete, and show how the X-Trace graphs corresponding to particular executions shed light on the underlying causes for the delays. These request are outlined in the graph, and we were surprised that there was a fair number of requests with this specific duration, with widely varying object sizes.

By examining the X-Trace task graphs generated by these requests, we identified different causes for approximately the same processing time. We list these causes below:

(a) Object found in local Coral node, slow connection between local Coral node and client

(b) Object fetched from origin, slow connection between local Coral node and client

(c) Object fetched from origin, slow connection between origin and local Coral node

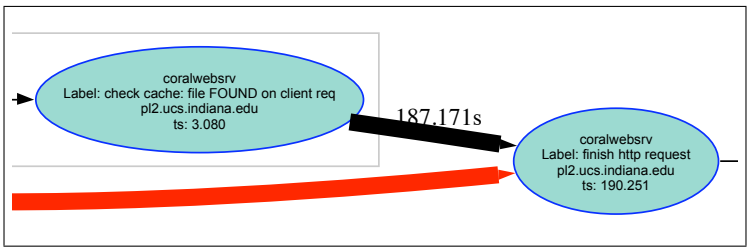
(d) TCP connection timeout when local Coral node revalidating to origin

(e) TCP connection timeout when remote Coral node revalidating to origin

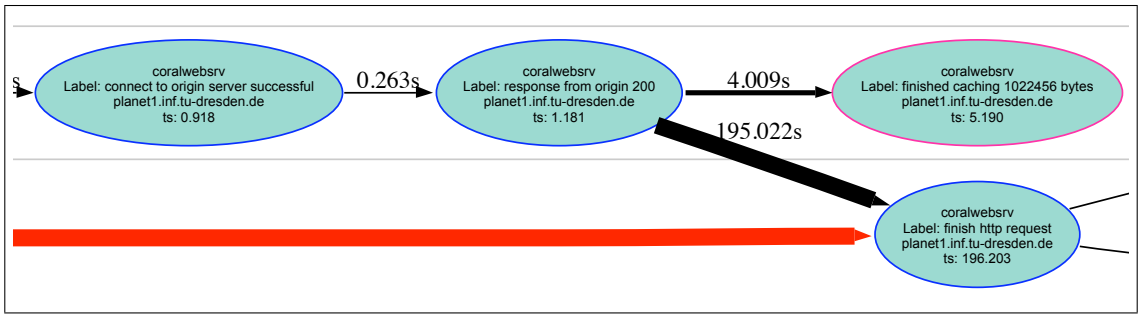
(f) Timeout in Coral RPC, due to PlanetLab scheduling

In the list, *local Coral node* is the first Coral node contacted by the client, *remote Coral node* is a Coral peer, and *origin* is the remote server with the original content. Figure 6.6 shows portions of example graphs for the first four cases in the list above. The relevant information in the graphs is which edge represents the most time. Cases (a), (b), and (c) show that a combination of object size and (slow) link speed, either between the origin server and a Coral node, or between the Coral node and the client, resulted in the long delay.

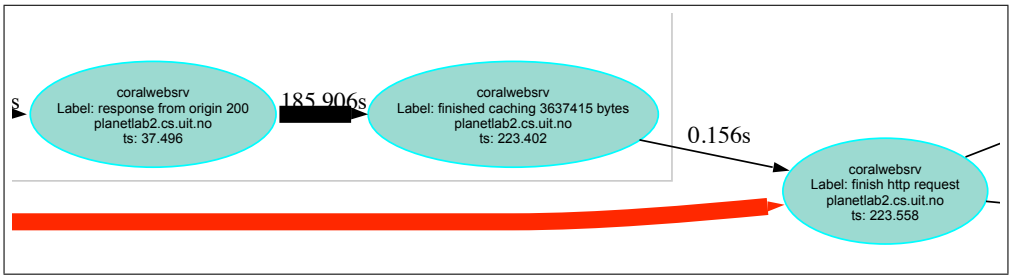
Cases (d) and (e - not shown in the figure) are due to a TCP timeout in the Coral machines. The timeout occurred because the other end was not responding to packets, most likely because it was



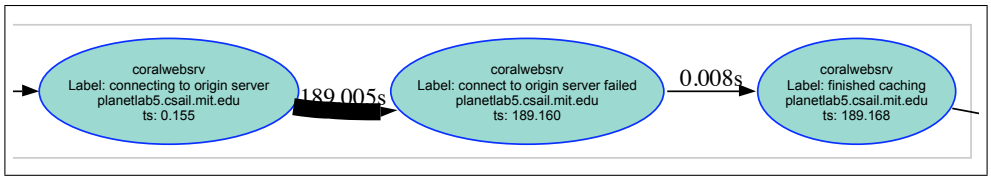
(a) From local, slow connection between proxy and client.



(b) From origin, slow connection between proxy and client.



(c) From origin, slow connection between origin and proxy.



(d) TCP connection timeout between proxy and origin.

Figure 6.6: Portions of example X-Trace task graphs for the first four causes of the 189-second delay listed in the text. The thickness of the edge between two events is proportional to the time between the events.

overloaded. The value of the timeout depends on the TCP implementation at the client. The version of Linux used by Coral (kernel 2.6.x) has a TCP timeout of 189 seconds when the remote node does not respond to SYN packets. The timeout value starts at 3 seconds, doubles at each timeout, and cannot be larger than 120 seconds [24]. Thus, we observe the sequence of timeouts of 3, 6, 12, 24, 48, and 96 seconds, adding up to 189 seconds.

Lastly, for case (f) (not shown in the figure), the cause was the PlanetLab scheduler. PlanetLab offers a virtualized environment to applications, and if a node is overloaded, can schedule a *slice*[1] out for a relatively long time. In one of the tasks this happened, and we observed a combination of RPC timeouts in Coral that led to an overall response time of 190 seconds.

6.3.2 Uncovered bugs

Oasis Multiple Queries

Our X-Trace instrumentation of OASIS combines events at the application and RPC layers, across processes and across machines. This allows traces to properly nest recursive RPC calls, which greatly facilitated our finding a DNS resolution-related bug in OASIS.

When a client issues a DNS request for a service to OASIS, the request goes to a core node. Call this node A. As briefly described in Section 6.1.1, A will contact one of the core rendezvous nodes, B, for the service in question. When this contact fails (either because of a timeout, or because B has no answer for the query), A does the same lookup on up to two other nodes, in series. In some cases, however, we observed tasks in which A would contact three nodes, say B, C, and D, and then would contact the same three nodes again, but in a random order.

Figure 6.7 illustrates the problem. The resolution starts with a *RECS* RPC call between two

[1] A slice is the unit of virtualization in PlanetLab.

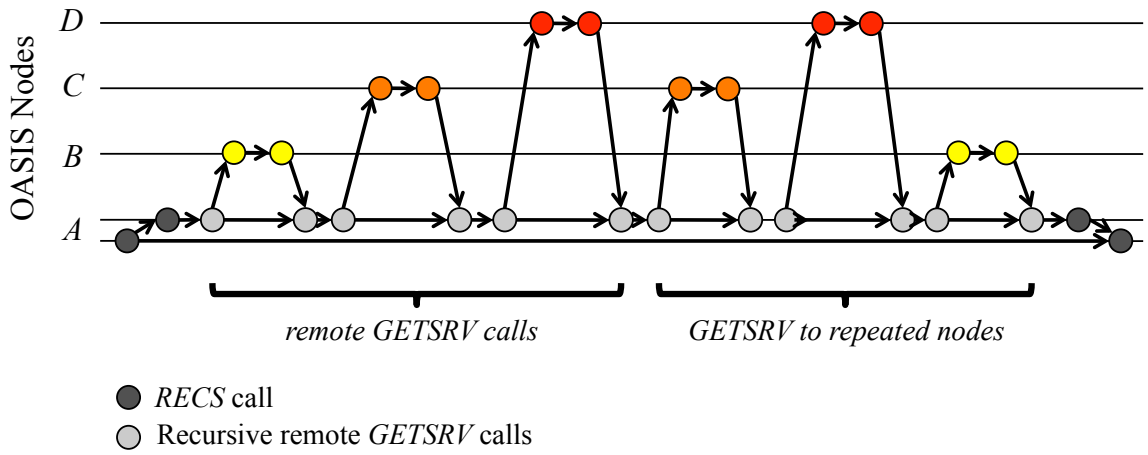


Figure 6.7: Simplified X-Trace graph showing OASIS repeated nodes bug. A DNS lookup in OASIS is done through a RECS RPC call, which may invoke remote GETSRV RPC calls recursively. We found cases in which the calling node would issue unnecessary GETSRV calls to repeated nodes, increasing latency and load.

OASIS processes in A. The server process then initiates the resolution. If the answer is cached locally, A contacts no more nodes; otherwise it contacts one or more of the rendezvous servers using *GETSRV* RPC calls. It turns out that when successively failing to obtain an answer, OASIS would try to select nodes to query twice: first, by choosing three of the *closest* rendezvous nodes to the client, and then, by choosing three *random* rendezvous nodes. It did not, however, mark the nodes selected in the first selection as already used, causing the repetition. The repeated queries delay the answer to the client, and present extra load to the OASIS servers.

Figure 6.8 shows data that quantifies the impact of this problem. It looks at all resolutions that did not involve a timeout in either the *RECS* or the *GETSRV*, comprising 99.7% of all resolutions recorded. The table in Figure 6.8(a) partitions the resolutions by the number of recursive calls. From the table, 3.32% of the resolutions exhibited the bug, with 6 recursive calls to *GETSRV*, 3 of them repeated. The timeout value for *RECS* calls was set to 10 seconds, which is why the maximum for all partitions is never higher than 10. The timing data in the table, and the curves in Figure 6.8(b), show that resolutions with the bug are significantly slower than those without the repeated lookups: the average time for a *RECS* call with 6 *GETSRV* calls is 1.8× that of resolutions with 3 *GETSRV* calls. Fixing this bug would have a significant impact in the tail of the overall OASIS response time distribution.

We found this bug by counting how many times each node was called in each resolution, according to the task graph nesting structure. This observation would be much harder to make if we only had RPC-level traces. The problem could be found with careful application-level logging, but the solution would be ad-hoc and require more effort to log each RPC call. X-Trace, in contrast, simplifies application instrumentation because it leverages the lower layers instrumentation "for

free".

Multiple Revalidations in Coral

The second bug we examine in detail we found in Coral. When the content for a URL requested to a Coral node is found in the local Coral cache but has expired, the node normally contacts the origin server for revalidation, just like a regular Web cache would do. In Coral, however, under specific circumstances, this behavior can lead to an amplification of traffic the origin server.

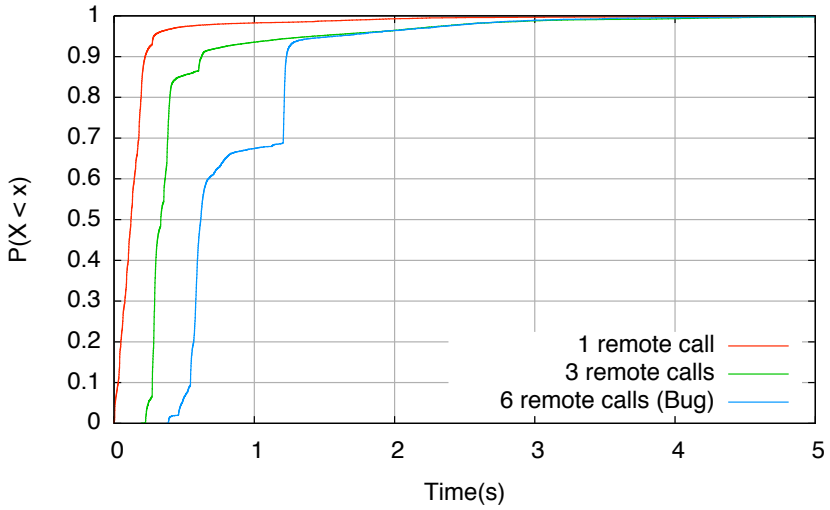
Consider the following situation: a client requests an URL to a Coral node A. A does not have the content cached, and looks up the URL in the DSHT, obtaining 3 other nodes, B, C, and D, that have the content. A then chooses one of these, B, and sends a recursive HTTP request to B. B receives the request, and notices that it indeed has the content, but its local copy has expired. B then contacts the origin server with a revalidation request. If this operation takes too long (more than A's timeout value for a peer request), A gives up, and proceeds to the next node in the list, C. If C also has a stale copy of the object, it too will contact the origin server for revalidation.

This was a rare occurrence in our trace, but is potentially problematic because it is more likely to happen exactly when the remote server is overloaded. We observed only 23 tasks (out of 44,133) that had multiple requests to the origin server from Coral peers. 1 had 4 requests, 4 had 3 requests, and 18 had 2 requests.

One of the cases in which we observed this problem was also when we observed TCP connection timeouts of 189 seconds at the origin server. The server was taking too long to even return TCP SYN packets, and instead of one extra connection that a regular client would generate, Coral generated 3. Since this content was very popular at this time, presumably more than one client would try to download it through Coral, and their traffic to the origin server would be amplified by Coral.

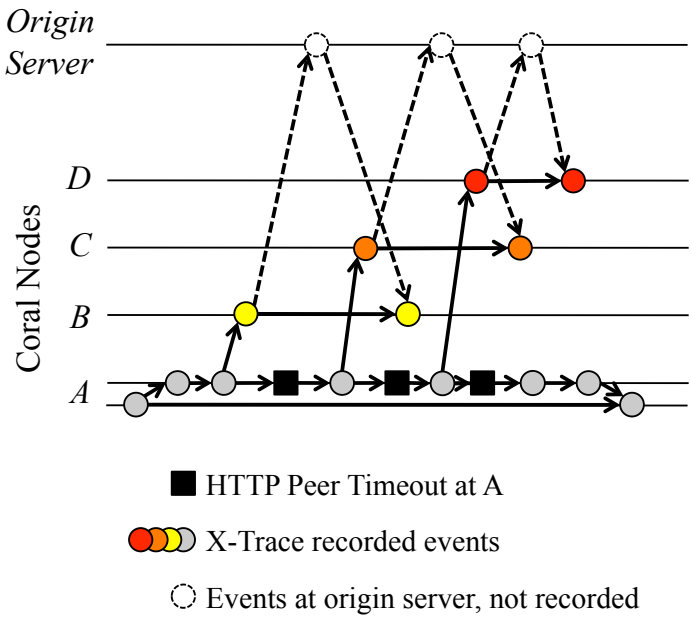
	Number of Remote GETSRV calls				
	0	1	3	6 (Bug)	Others
Total RECS Calls	251,906	42,142	19,672	10,780	359
Fraction	77.54%	12.97%	6.06%	3.32%	0.11%
Mean (s)	0.038	0.167	0.481	0.870	-
Minimum (s)	0.000	0.001	0.194	0.370	-
Median (s)	0.002	0.121	0.334	0.618	-
Maximum (s)	9.968	9.769	9.673	9.854	-

(a) Fraction of RECS calls with each number of remote GETSRV calls. Times correspond to the curves in (b).

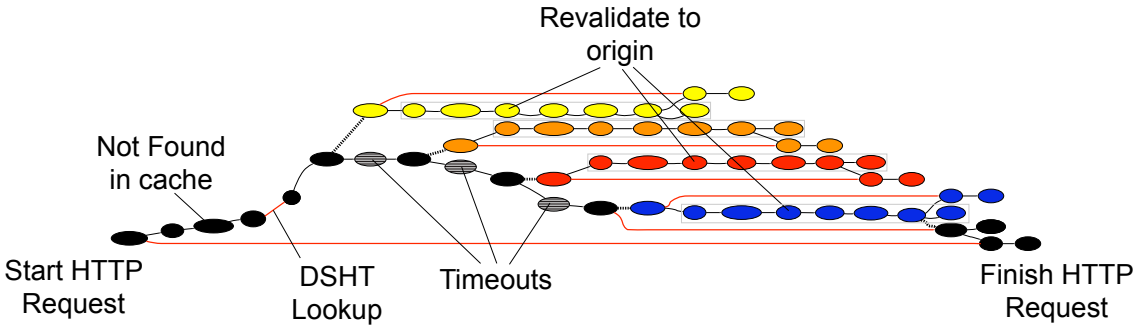


(b) CDF of RECS RPC completion time, separated by the number of remote recursive calls. The spike around 1.2s for the 6 remote calls line is due to a server in Japan with higher latency to other nodes.

Figure 6.8: Distribution of completion time for the RECS *RPCs in the OASIS trace. Each* RECS *call causes 0 or more* GETSRV *calls. All cases with 6 calls, 3.32% of all* RECS *calls, showed the repeated node bug. The mean time for for these calls is 1.8× that of the* RECS *calls with 3 recursive calls.*



(a) Simplified X-Trace graph showing multiple revalidations to the origin server. A tries to fetch an object from the Coral peers B, C, and D, timing out for each. B, C, and D try to send revalidation requests to the origin.



(b) Annotated X-Trace graph produced by our visualization tool for a real occurrence of the above pattern with 4 revalidations. Nodes with different colors represent events in different Coral peers. Individual node labels omitted.

Figure 6.9: Coral X-Trace graphs with multiple revalidations to origin server.

A solution might include, for example, A adding a flag to its recursive requests to prevent the peer from revalidating the content. A could then add this flag to all but one of the recursive requests. Another approach would be to increase the value of the timeout, although this might have an adverse effect on other calls that don't suffer from this problem.

We found this situation with similar analysis techniques as in the OASIS bug described above. Specifically, by walking the X-Trace graph we could count how many recursive HTTP calls corresponded to each client HTTP request, how many of these timed out, and how many of, when executing in the other Coral nodes, generated requests to the origin server. Doing this type of analysis without X-Trace would be quite labor intensive, as multiple logs from individual nodes would have to be joined and queried. In this case, the join would be approximate, unless special identifiers were included and logged in the HTTP transactions. The alternative, of having the calling node record this information, would break abstraction barriers, because it is generally oblivious, by design, of how exactly the peers obtain the content.

Other Coral Bugs

We also found a few other bugs in Coral examining X-Trace graphs, and only briefly mention them here.

1. *Forbidden response meant for peer proxy passed to client.* We identified instances in which a forbidden response status passed from one Coral peer to another was being relayed unmodified to the client. X-Trace task graphs make it easy to correlate the two sets of HTTP communications.

2. *Revalidation never uses peer proxy.* This bug is related to the multiple revalidations bug

described above. We verified that no revalidation request was issued to peer Coral nodes. A Coral proxy could be modified to check if one of its peers have a newer copy of an expired cached entry.

3. *All calls to RPC OFFREP timeout at the caller.* This was a minor bug. OFFREP is a Coral RPC with `void` return type. In this RPC, the client doesn't wait for a response, and the server never sends a response, but the RPC layer implementation only released resources when the call times out. This is not likely to be a problem for this RPC is quite uncommon in Coral.

4. *Some paths when server fetch fails may not kill client connection.* We observed some long client connections timing out, and were able to correlate those with some instances in which the Coral node failed to fetch content from the origin server.

5. *Coral node issues a last lookup at a given level and immediately declares failure before getting any response.* Coral peers organize themselves in a three-level hierarchy based on latencies to each other, and try lookups at each level in succession. An "off-by-one" error caused the last one of the lookups at a level to be ignored.

6. *Does Coral lookup even when likely to be over cache size* A Coral node will try to fetch remote objects even when it is likely there will be no space to store the entry. In this case a node should forward the connection to the origin server.

6.4 Discussion

In this chapter we demonstrated X-Trace usefulness in two deployments on wide-area production systems. X-Trace task graphs allow flexible queries on specific executions, including queries

that involve activity on different layers and on different nodes. In particular, we were able to find the cause of distinct problems with the same symptoms, find performance problems, and identify bugs that had gone unnoticed in over three years of use. The ability to sample specific tasks end-to-end was important to reduce the data collected with Coral, while obtaining meaningful traces.

Other researchers have also used X-Trace to instrument different systems, with promising results. The authors of DONA [94] used X-Trace during development to find bugs in their distributed router code; Porter [134] used X-Trace to find the root cause of faults in an 802.1X enterprise deployment; and Zaharia et al. [177] instrumented the Hadoop system [73], an open-source implementation of Google's Map-Reduce and distributed file system framework [39, 69], with X-Trace. They successfully used the instrumentation in experimental and production Hadoop clusters to detect problem node, tune configuration parameters, and to guide improvements to the job scheduling mechanisms in Hadoop.

These results open important avenues for continued research. To be used continuously in production, X-Trace requires a robust data collection and processing infrastructure. Given limited resources, we had to sample the Coral traffic to cope with the volume of data. Porter, in [134], gives an important step in that direction with TraceOn, a more scalable data collection and processing infrastructure. We hope to be able to use infrastructure like Hadoop [73] to process larger volumes of X-Trace data. Another important direction is on automating the analysis of X-Trace data to detect correctness and performance problems. The approaches in Pip [139] and Pinpoint [31], for example, serve as good starting points.

One challenge we faced in these deployments concerns tasks with missing reports. Missing reports result in incomplete graphs, most likely disconnected. In some cases it is possible to infer

that specific reports were missing, by comparing to expected behavior or to other similar tasks. In fact, preliminary clustering analysis we did[2] found that in some cases tasks with a few missing reports would still be clustered together with similar, but complete, tasks. There are cases, however, in which it is not possible to distinguish a task graph that stops abruptly because of a failure in the execution from one that stops because of missing reports. In these cases, an operator can start a diagnostics process in the last machine that did generate valid reports for that task.

This chapter concludes our evaluation of X-Trace. In the next chapter we shift to another environment and introduce Quanto, a framework that uses similar techniques of following the execution in a distributed system for profiling resource usage in embedded wireless networks.

[2]Credit on these analyses goes to Fabian Waulthier.

Chapter 7

Quanto: Tracing Embedded Wireless Systems

In this chapter we leave the realm of large-scale distributed systems, and shift our focus to improving the visibility of networked embedded systems. We present Quanto, a network-wide time and energy profiler for embedded network devices, and describe its concrete implementation on the TinyOS operating system [78]. Specifically, we look at how to profile the time and energy consumption of tasks in wireless sensor network applications, using techniques of tracing and execution-following similar to those in X-Trace. As we did with their large-scale counterparts, in these systems we track causally related events across boundaries of software and hardware components, and across different nodes in the network. Our solution, though, is shaped by different questions and by the severe resource constraints that these embedded device impose. Quanto combines causal tracking of programmer-defined activities with a novel high-resolution energy metering technique to map where and how energy and time are spent on nodes and across a network. This can

be invaluable for developers and deployers to understand and optimize the behavior of applications.

Wireless sensor networks, or simply, sensornets, are collections of small devices with integrated computing, sensing, and networking [48]. The nodes in the network, called *motes*, are embedded in the physical environment, and run distributed applications that interact with the environment by sensing, computing, and possibly controlling actuators. There have been finding applications in many fields, such as in experimental biology [164], structural monitoring [126], flood warning systems [21], soil ecology monitoring [118], and asset tracking [106], to name a few.

Energy is the key limiting resource in sensor network deployments. Long-term deployments are desired for most applications. However, nodes may be in hard-to-reach or outright inaccessible places (such as embedded inside a wall in a building), making battery replacement difficult or impossible. Even if energy is harvested from the environment [160], it may be scarce and unpredictable. Network lifetime thus depends crucially on judicious use of energy by applications, and optimizing computation, communication, and memory usage for energy usage a key concern.

Although we share with X-Trace many of the same techniques and the design decision of propagating small, constant-sized metadata along with the computation, different concerns and severe resource constraints make the X-Trace solution not directly applicable for this context. Instead of reconstructing the happens-before relation among events, we are primarily interested here in using the causal relation to *group* events, providing the right granularity for resource accounting (*cf.* Section 7.3). Metadata in Quanto is only two bytes in length, and information collected during the execution trades the generality of text-based reports for a highly optimized binary format. Another difference is that, in these platforms, the majority of the energy is spent by devices other than the CPU, and we have to track on behalf of which activities these peripherals are doing work.

This chapter describes in detail our implementation of Quanto in TinyOS. Section 7.2 provides an outline of the solution. The next two sections describe the two major components in Quanto. Section 7.3 presents activity tracking, which uses causal tracking of related events to answer *why* energy is spent. It defines what we mean by activities, how to propagate them, the API used by programmers, and how we record the information. Section 7.4 describes energy tracking, which answers the question of *where* energy was spent. It describes the hardware we use to measure energy, and the method to appropriately divide the aggregate energy usage among the different hardware components. In Chapter 8 we provide an evaluation of our implementation of Quanto, and further discussions on the design decisions, related and future work.

7.1 Motivation

Energy is a scarce resource in embedded, battery-operated systems such as sensor networks. This scarcity has motivated research into new system architectures [75], platform designs [77], medium access control protocols [176], networking abstractions [133], transport layers [117], operating system abstractions [92], middleware protocols [47], and data aggregation services [105]. In practice, however, the energy consumption of deployed systems differs greatly from expectations or what lab tests suggest. In one network designed to monitor the microclimate of redwood trees, for example, 15% of the nodes died after one week, while the rest lasted for months [163]. The deployers of the network hypothesize that environmental conditions – poor radio connectivity, leading to time synchronization failure – caused the early demise of these nodes, but a lack of data makes the exact cause unknown.

Understanding how and why an embedded application spends energy requires answering numer-

ous questions. For example, how much energy do individual operations, such as sampling sensors, receiving packets, or using CPU, cost? What is the energy breakdown of a node, in terms of activity, hardware, and time? Network-wide, how much energy do network services such as routing, time synchronization, and localization, consume?

Three factors make these questions difficult to answer. First, there is a semantic gap between common abstractions, such as threads or subsystems, and the actual entities a developer cares about for resource accounting. This gap requires a profiling system to tie together separate operations across multiple energy consumers, such as sampling sensors, sending packets, and CPU operations, as well as across multiple nodes. Second, nodes need to be able to measure the actual draw of their hardware components. While software models of system energy are reasonably accurate in controlled environments, networks in the wild often experience externalities, such as 100° temperature shifts [159], electrical shorts due to condensation [158], and 802.11 interference [129]. Finally, nodes have limited storage capability, on the order of kilobytes of RAM, and profile collection must be very lightweight, so it is energy efficient and minimizes its effect on system behavior.

Our approach, Quanto, addresses these challenges through three research contributions:

1. We describe a simple labeling mechanism that causally connects resource usage to high-level, programmer-defined activities. We extend these techniques to track network-wide resource usage in terms of node-local actions.

2. We leverage an energy sensor based on a simple switching regulator [44] to enable an OS to take fine-grained measurements of energy usage as cheaply as reading a counter.

3. We show that a post-facto regression can distinguish the energy draw of individual hardware components, thereby only requiring the OS to sample aggregate system consumption.

Combining these techniques, Quanto is able to partition energy usage per activity and hardware component over time. In the implementation we describe, we perform this accounting by recording events regarding activity and energy draw changes. Recording each event takes 101 processor cycles and 12 bytes of memory, and can be done in a relatively unobtrusive fashion. We briefly outline these ideas next.

7.2 Solution Outline

Energy in an embedded system is spent by a set of hardware components operating concurrently in each node, responding to application and external events. As a first step in understanding energy usage, Quanto defines the the appropriate granularity to accumulate and account for energy usage. We want to attribute energy usage to high-level tasks that are meaningful to the programmer, such as sensing, routing, or computing. Earlier work has profiled energy usage at the level of instructions [40], performance events [35], program counter [54], procedures [55], processes [153], and software modules [147]. We borrow from the RIALTO operating system [88] the concept of an *activity* as the abstraction for a resource principal [18]. An activity is a logical set of operations whose resource usage should be grouped together. In the embedded systems we consider, most of the energy is not spent by the CPU, and it is essential that activities span all hardware components. Likewise, given the networked nature of applications, activities must also span different nodes.

Each activity is given a label, and the OS propagates this label to all causally related operations. As an analogy, this tracking is accomplished by conceptually "painting" a hardware component the same "color" as the activity for which it is doing work. To transfer activity labels across nodes, Quanto inserts a field in each packet that includes the initiating activity's label. More specifically,

when a packet is passed to the network stack for transmission, the packet's activity field is set to the CPU's then-current activity. This ensures that a transmitted packet is labeled the same as the activity which initiated its submission. Upon reception, Quanto reads the packet's activity field and sets the CPU activity to the activity noted in the packet.

The concept of an activity in Quanto is very similar to the concept of an X-Trace *task*, from Chapter 3.[1] In X-Trace, we are interested in registering key events that happen as part of a task, together with their causal relations. In Quanto, instead of registering individual events, we accumulate the time that each hardware component spends operating on behalf of each activity. This allows us to more directly answer the question of how and where energy is spent, while requiring less resources for the instrumentation.

Given the time profile of activities, the second step is to determine the energy breakdown by hardware component over time. A system has several hardware components, like the CPU, radio, and flash memory, and each one has different functional units, which we call *energy sinks*. Each energy sink has operating modes with different power draws, which we call *power states*. At any given time, the aggregate power draw for a system is determined by the set of active power states of its energy sinks.

In many embedded systems, the system software can closely track the hardware components' power states and state transitions. We modify device drivers to track and expose hardware power states to the OS in real-time. The OS combines this information with fine-grained, timely measurements of system-wide energy usage taken using a high-resolution, low-latency energy meter. Every time any hardware component changes its power state, the OS records how much energy was

[1] TinyOS already has a very specific meaning for the term task, which prevented us from using the same name. A task in TinyOS is a special type of function that represents the unit of schedulable computation by the OS.

used, and how much time has passed since the immediately preceding power state change. For each interval during which the power states are constant, this generates one equation relating the active power states, the energy used, the time spent, and the unknown power draw of a particular energy sink's power state. Over time, a family of equations are generated and can be solved (i.e. the power draw of individual energy sinks can be estimated) using multivariate linear regression.

The final step is to merge the information from these two sources. Quanto records events for both activity and power state changes for each hardware resource. In our current prototype, we use these logs to perform this step *post-facto*. From the power states log and the regression, we know the active power state and the power draw for each hardware component; from the activities log, we know on behalf of which activity the component was doing work. Combining these two pieces of information provides a thorough breakdown of energy consumption over time.

In the next section we give a more precise definition of activities, and detail how Quanto implements activity tracking across both the hardware components of a single node and across the nodes in a network. Then in Section 7.4 we present details of the approach for tracking the energy usage by each hardware component over time.

7.3 Activity Tracking

This section addresses the question of *why* energy is spent on a system. The key here is to attribute energy usage to entities – or resource principals – that are meaningful to the programmer. In traditional operating systems, processes or threads combine the roles of protection domain, schedulable unit, and resource principal, but there are many situations in which it is desirable that these notions be independent. This idea was previously explored in the context of high-performance

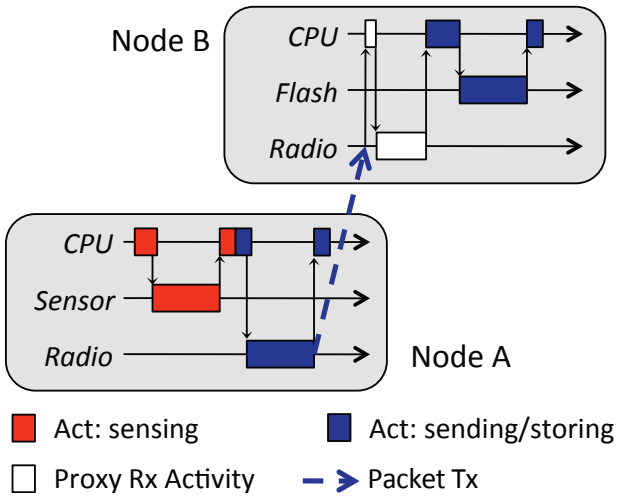


Figure 7.1: Activity tracking for a sensing, sending, and storing a sample across two nodes. The developer chose sending as a separate activity. Receiving is part of a proxy activity until the CPU can decode the true activity and correctly bind the resource usage.

network servers [18] but it is also especially true in networked embedded systems.

We borrow from earlier work the concept of an *activity* as our resource principal. In the RIALTO system in particular [88], an activity was defined as the "*the abstraction to which resources are allocated and to which resource usage is charged*." In other words, an activity is a set of operations whose resource consumption should be grouped together. In the environments we consider, where most of the resource consumption does not happen at the CPU, and sometimes not even on the same node that initiated an activity, it is fundamental to support activities that span different hardware components and multiple nodes.

7.3.1 Overview

To account for the resource consumption of activities, we track when a hardware component, or device, is performing operations on behalf of an activity. A useful analogy is to think of an activity as a color, and devices as being painted with the activity's color when working on its behalf. By properly recording devices' successive colors over time and their respective resource consumptions, we assign to each activity its share of the energy usage.

Figure 7.1 shows an example of how activities can span multiple devices and nodes. In the figure, the programmer marks the start of an activity by assigning to the CPU the *sensing* activity ("painting the CPU red"). We represent activities by activity labels, which Quanto carries automatically to causally related operations. For example, when a CPU that is "painted red" invokes an operation on the sensor, the CPU paints the sensor red as well. The programmer may decide to change the CPU activity if it starts work on behalf of a new logical activity, such as when transitioning from sensing to sending (red to blue in the figure). Again the system will propagate the new activity to other devices automatically.

This propagation includes carrying activity labels on network messages, such that operations on node B can be assigned to the activity started on node A. This example also highlights an important aspect of the propagation, namely *proxy* activities. When the CPU on node B receives an interrupt indicating that the radio is starting to receive a packet, the activity to which the receiving belongs is not known. This is generally true in the case of interrupts and external events. Proxy activities are a solution to this problem. The resources used by a proxy activity are accounted for separately, and then assigned to the real activity as soon as the system can determine what this activity is. In this example the CPU can determine that it should be colored blue as soon as it decodes the activity

label in the radio packet. It terminates the proxy activity by *binding* it to the blue activity.

The programmer can define the granularity of activities in a flexible way, guided by how she wants to divide the resource consumption of the system. Some operations do not clearly belong to specific activities, such as data structure maintenance or garbage collection. One option is to give these operations their own activities, representing this fact explicitly.

The mechanisms for tracking activities are divided into three parts, which we describe in more detail next, in the context of TinyOS: (i) an API that allows the programmer to create meaningful activity labels, (ii) a set of mechanisms to propagate these labels along with the operations that comprise the activity, (iii) and a mechanism to account for the resources used by the activities.

7.3.2 API

We represent activity labels with pairs of the form ⟨*origin node*:*id*⟩, where *id* is a statically defined integer, and *origin node* indicates the node where the activity starts.

We provide an API that allows the assignment of activity labels to devices over time. This API is shown in Figures 7.2 and 7.3, respectively, for devices that can only be performing operations on behalf of one, or possibly multiple activities simultaneously. Most devices, including CPUs, are `SingleActivityDevices`.

There are two classes of users for the API, application programmers and system programmers. Application programmers simply have to define the start of high-level activities, and assign labels to the CPU immediately before their start. System programmers, in turn, use the API to propagate activities in the lower levels of the system such as device drivers. We instrumented core parts of the OS, such as interrupt routines, the scheduler, arbiters [92], the network stack, radio, and the timer system. Figure 7.4 shows an excerpt of a sense-and-send application similar to the one described

```
interface SingleActivityDevice {
  // Returns the current activity
  async command act_t get();

  // Sets the current activity
  async command void set(act_t newActivity);

  // Sets the current activity and indicates
  // that the previous activity's resource
  // usage should be charged to the new one
  async command void bind(act_t newActivity);
}
```

Figure 7.2: The `SingleActivityDevice` *interface. This interface represents hardware components that can only be part of one activity at a time, such as the CPU or the transmit part of the radio.*

```
interface MultiActivityDevice {
  // Adds an activity to the set of current
  // activities for this device
  async command error_t add(act_t activity);

  // Removes an activity from the set of current
  // activities for this device
  async command error_t remove(act_t activity);
}
```

Figure 7.3: The `MultiActivityDevice` *interface. This interface represents hardware components that can be working simultaneously on behalf of multiple activities. Examples include hardware timers and the receiver circuitry in the radio (when listening).*

```
task void sensorTask() {
  call CPUActivity.set(ACT_HUM);
  call Humidity.read();
  call CPUActivity.set(ACT_TEMP);
  call Temperature.read();
}

void sendIfDone() {
  if (sensingDone) {
    call CPUActivity.set(ACT_PKT);
    post sendTask();
    sensingDone = 0;
  }
}
```

Figure 7.4: Excerpt from a sense-and-send application, showing how an application programmer "paints" the CPU to start tracking activities.

in [92], in which the application programmer "paints" the CPU using the `CPUActivity.set` method (an instance of the `SingleActivityDevice` interface) before the start of each logical activity. The OS takes care of correctly propagating the labels with the following execution.

7.3.3 Propagation

Once we have application level activities set by the application programmer, the OS has to carry activity labels to all operations related to each activity. This involves 4 major components: (i) transfer activity labels across devices, (ii) transfer activity labels across nodes, (iii) bind proxy activities to real activities when interrupts occur, and (iv) follow logical threads of computation across several control flow deferral and multiplexing mechanisms.

To transfer activity labels across devices in our TinyOS instrumentation, we use the `Single-` and `MultiActivityDevice` APIs. Each hardware component is represented by one instantiation of such interfaces, and keeps the activity state for that component globally accessible to code.

The CPU is represented by a `SingleActivityDevice`, and is responsible for transferring activity labels to and from other devices. An example of this transfer is shown in Figure 7.5, where the code "paints" the radio device with the current CPU activity. Device drivers must be instrumented to correctly transfer activities between the CPU and the devices they manage. In our prototype implementation we instrumented several devices, including the CC2420 radio and the SHT11 sensor chip. Also, we instrumented the Arbiter abstraction [92], which controls access to a number of shared hardware components, to automatically transfer activity labels to and from the managed device.

To transfer activity labels across nodes, we added a hidden field to the TinyOS Active Message (AM) implementation (the default communication abstraction). When a packet is submitted to the OS for transmission, the packet's activity field is set to the CPU's current activity. This ensures the packet is colored the same as the activity which initiated its submission. We currently encode the labels as 16-bit integers representing both the node id and the activity id, which is sufficient for networks of up to 256 nodes with 256 distinct activity ids. Upon decoding a packet, the AM layer on the receiving node sets the CPU activity to the activity in the packet, and binds resources used between the interrupt for the packet reception and the decoding to the same activity.

More generally, this type of resource binding is done when we have interrupts. Our prototype implementation uses the Texas Instruments MSP430F1611 microcontroller. Since TinyOS does not have reentrant interrupts on this platform, we statically assign to each interrupt handling routine a fixed proxy activity. An interrupt routine temporarily sets the CPU activity to its own proxy activity, and the nature of interrupt processing is such that very quickly, in most cases, we can determine to which real activity the proxy activity should be bound. One example is the decoding of the radio

packets at the Active Message layer. Another example is an interrupt caused by a device signaling the completion of a task. In this case, the device driver will have stored locally both the state required to process the interrupt and the activity to which this processing should be assigned.

Lastly, the propagation of activity labels should follow the control flow of the logical threads of execution across deferral and multiplexing mechanisms. The most important and general of these mechanisms in TinyOS are *tasks* and *timers*.

TinyOS has a single stack, and uses an event-based execution model to multiplex several parallel activities among its components. The schedulable unit is a *task*. Tasks run to completion and do not preempt other tasks, but can be preempted by asynchronous events triggered by interrupts. To achieve high degrees of concurrency, tasks are generally short lived, and break larger computations in units that schedule each other by posting new tasks. We instrumented the TinyOS scheduler to save the current CPU activity when a task is posted, and restore it just before giving control to the task when it executes, thereby maintaining the activities bound to tasks in face of arbitrary multiplexing. This instrumentation follows exactly the same principles as the instrumentation of callbacks in `libasync`, described in Chapter 3. Timers are also an important control flow deferral mechanism, and we instrumented the virtual timer subsystem to automatically save and restore the CPU activity of scheduled timers.

There are other less general structures that effectively defer processing of an activity, such as forwarding queues in protocols, and we have to instrument these to also store and restore the CPU activity associated with the queue entry. As we show in Section 8.4.2, changes to support propagation in a number of core OS services were small and localized.

```
void loadTXFIFO() {
  ...
  //prepare packet
  ...
  call RadioActivity.set(call CPUActivity.get());
  call TXFIFO.write((uint8_t*)header,
                    header->length - 1);
}
```

Figure 7.5: Excerpt from the CC2420 transmit code that loads the TXFIFO with the packet data. The instrumentation sets the `RadioActivity` *to the current value of the* `CPUActivity`.

7.3.4 Recording and Accounting

The final element of activity tracking is recording the usage of resources for accounting and charging purposes. Similarly to how we track power states, we implement the observer pattern through the `SingleActivityTrack` and `MultiActivityTrack` interfaces (Figure 7.6). These are provided by a module that listens to the activity changes of devices and is currently connected to a logger. In our prototype we log these events to RAM and do the accounting offline. For single-activity devices, this is straightforward, as time is partitioned among activities. For multi-activity devices, the the log records the set of activities for a device over time, and how to divide the resource consumption among the activities for each period is a policy decision. We currently divide resources equally, but other policies are certainly possible.

7.4 Energy Tracking

In this section, we present how Quanto answers the question of *where* energy is spent in the system. This requires distinguishing the energy consumption of individual hardware components or peripherals that are operating concurrently when only their aggregate energy usage is observable.

```
interface SingleActivityTrack {
  async event void changed(act_t newActivity);
  async event void bound(act_t newActivity);
}
interface MultiActivityTrack {
  async event void added(act_t activity);
  async event void removed(act_t activity);
}
```

Figure 7.6: `Single-` *and* `MultiActivityTrack` *interfaces provided by device abstractions. Different accounting modules can listen to these events.*

We ground our discussion on the specific hardware and software platform in our prototype, although we believe the techniques to be applicable to other platforms as well. We briefly sketch our approach in the next subsection and then use the remainder of this section to describe Quanto's energy tracking framework in detail.

7.4.1 Overview

In many embedded systems, the OS can track the power states and state transitions of the platform's various energy sinks. This power state information can be combined with snapshots of the aggregate energy consumption to infer the consumption of individual sinks.

We call each functional unit in a system an *energy sink*, and their different operating modes *power states*. Quanto modifies the device drivers to intercept all events which change the power state of an energy sink. The OS itself keeps track of both the energy usage, ΔE, and elapsed time, Δt, between any two such events. Since the OS tracks the active sinks and their power states, it is able to generate one linear equation of the following form for each interval

$$\Delta E = \Delta t \sum_{i=0}^{n} \alpha_i p_i \tag{7.1}$$

where the average power over the interval, P, is $\Delta E/\Delta t$. The variable α_i is a binary variable indicating whether the i-th power state was active during the interval, and p_i is the (unknown) power draw of the i-th state. The limit n represents the total number of power states over all energy sinks in the system. In one time interval, this equation is not solvable (unless only one power state is active), but over time, an application generates a system of equations as different energy sinks transition through different power states. When the system of equations is sufficiently constrained, a simple linear regression yields the individual power draws.

7.4.2 Hardware Platform

Because it samples the accumulated energy consumption at every hardware power state change, Quanto requires high-resolution, low-latency, and low-overhead energy measurements. These readings must closely reflect the energy consumed during the preceding interval. To accomplish this, our implementation uses the iCount energy meter [44]. The iCount implementation on this platform exhibits a maximum error of $\pm 15\%$ over five orders of magnitude in current draw, an energy resolution of approximately 1 μJ, a read latency of 24 μs (24 instruction cycles), and a power overhead that ranges from 1% when the node is in standby to 0.01% when the node is active, for a typical workload.

We used a custom mote platform, called Hydrowatch [45], which incorporates the iCount hardware. Figure 7.7 shows a Hydrowatch mote. It uses the Texas Instruments 16-bit MSP430F1611 microcontroller running at 1 MHz, with 48 KB of internal flash memory and 10 KB of RAM, an 802.15.4-compliant CC2420 radio, and an Atmel 16-Mbit AT45DB161D NOR flash memory. The platform also includes three LEDs.

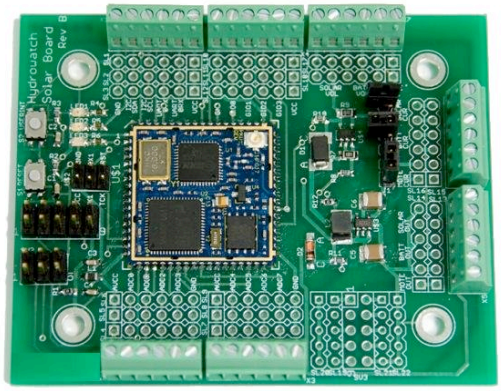


Figure 7.7: A custom hardware platform that integrates the iCount energy meter [44]. This meter allows the OS to sample the system's aggregate energy usage in 24 instruction cycles.

7.4.3 Energy Sinks and Power States

The Hydrowatch platform's energy sinks, and their nominal current draws, are shown in Table 7.1. The microcontroller includes several different functional units. The microcontroller's eight energy sinks have sixteen power states but since many of the energy sinks can operate independently, the microcontroller can exhibit hundreds of distinct draw profiles. The five energy sinks in the radio have fourteen power states. Some of these states are mutually exclusive. For example, the radio cannot use both receive and transmit at the same time. Similarly, the flash memory can operate in several distinct power states. Collectively, the energy sinks represented by the microcontroller, radio, flash memory, and LEDs can operate independently, so, in principle, the system may exhibit hundreds or thousands of distinct power profiles.

Energy Sink	Power State	Current
Microcontroller		
CPU	ACTIVE	500 μA
	LPM0	75 μA
	LPM1†	75 μA
	LPM2	17 μA
	LPM3	2.6 μA
	LPM4	0.2 μA
Voltage Reference	ON	500 μA
ADC	CONVERTING	800 μA
DAC	CONVERTING-2	50 μA
	CONVERTING-5	200 μA
	CONVERTING-7	700 μA
Internal Flash	PROGRAM	3 mA
	ERASE	3 mA
Temperature Sensor	SAMPLE	60 μA
Analog Comparator	COMPARE	45 μA
Supply Supervisor	ON	15 μA
Radio		
Regulator	OFF	1 μA
	ON	22 μA
	POWER_DOWN	20 μA
Batter Monitor	ENABLED	30 μA
Control Path	IDLE	426 μA
Rx Data Path	RX (LISTEN)	19.7 mA
Tx Data Path	TX (+0 dBm)	17.4 mA
	TX (-1 dBm)	16.5 mA
	TX (-3 dBm)	15.2 mA
	TX (-5 dBm)	13.9 mA
	TX (-7 dBm)	12.5 mA
	TX (-10 dBm)	11.2 mA
	TX (-15 dBm)	9.9 mA
	TX (-25 dBm)	8.5 mA
Flash	POWER_DOWN	9 μA
	STANDBY	25 μA
	READ	7 mA
	WRITE	12 mA
	ERASE	12 mA
LED0 (Red)	ON	4.3 mA
LED1 (Green)	ON	3.7 mA
LED2 (Blue)	ON	1.7 mA

Table 7.1: The platform energy sinks, their power states, and the nominal current draws in those states at a supply voltage of 3 V and clock speed of 1 MHz, compiled from the datasheets. †Assumed.

7.4.4 Exposing and Tracking Power States

Tracking power states involves a collaborative effort between device drivers and the OS: we modify the device driver that abstracts a hardware resource to expose the device power states through a simple interface, while the OS tracks and logs the power states across the system.

Quanto defines the `PowerState` interface, shown in Figure 7.8, and provides a generic com-

```
interface PowerState {
  // Sets the powerstate to value.
  async command void set(powerstate_t value);

  // Sets the bits represented by mask to value.
  async command void setBits(powerstate_t mask,
    uint8_t offset, powerstate_t value);
}
```

Figure 7.8: Device drivers must be modified to expose device power states using the `PowerState` *interface.*

ponent that implements it. A device driver merely declares that it uses this interface and signals hardware power state changes through its simple calls. This approach eliminates state tracking in many device drivers and simplifies the instrumentation of more complex device drivers. Multiple calls to the `PowerState` interface signaling the same state are idempotent: such calls do not result in multiple notifications to the OS.

Figure 7.9 illustrates the changes to the LED device driver to expose power states. This requires intercepting calls to turn the LED on and off and notifying the OS of these events. For a simple device like the LED which only has two states and whose power states are under complete control of the processor, exposing the power state is a simple and relatively low-overhead matter.

More involved changes to the device driver are needed if a device's power state can change outside of direct CPU control. Flash memory accesses, for example, go through a handshaking process during which the power states and transitions are visible to the processor but not directly controlled by it. Prior to a write request, a flash chip may be in an idle or sleep state. When the processor asserts the flash chip enable line, the flash transitions to a ready state and asserts the ready line. Upon detecting this condition, the processor can issue a write command over a serial bus, framed by a write line assertion. The flash may then signal that it is busy writing the requested data

```
async command void Leds.led0On() {
  call Led0PowerState.set(1);
  // Setting pin to low turns Led on
  call Led0.clr();
}

async command void Leds.led0Off() {
  call Led0PowerState.set(0);
  // Setting pin to high turns Led off
  call Led0.set();
}
```

Figure 7.9: Implementing power state tracking is simple for many devices, like LEDs, and requires signaling power state changes using the `PowerState` *interface.*

```
interface PowerStateTrack {
  // Called if an energy sink power state changes
  async event void changed(powerstate_t value);
}
```

Figure 7.10: The `PowerStateTrack` *interface is used by the OS and applications to receive power state change events in real-time.*

by asserting the busy signal. When finished with the write, the flash asserts the ready signal. In this example, the device driver should monitor hardware handshake lines or use timeouts to shadow and expose the hardware power state.

The glue between the device drivers and OS is a component that exposes the `PowerState` interface to device drivers and provides the `PowerStateTrack` interface, shown in Figure 7.10, to the OS and application. This component tracks the power states change events and only notifies the OS and registered application listeners when an actual state change occurs. Each time a power state changes, Quanto logs the current value of the energy meter, the time, and the vector of power states.

$\Delta E^{\dagger}$	Δt (μs)	α_1	α_2	α_3	α_4
...	...	...	...	...	...
45	8300	1	1	1	1
36	12207	1	0	1	0
964	1937469	0	0	0	0
4709	1943634	0	1	0	0
5709	1940917	0	0	1	1
30	9246	1	1	0	0
49	11810	1	0	1	1
7895	1944091	0	1	1	0
...	...	...	...	...	...

Table 7.2: A small snippet of the power state log shows the energy consumed (ΔE) over an interval (Δt) and the power states of all devices during that interval. †The unit of energy is a cycle of the switching regulator which must be determined initially through calibration.

7.4.5 Estimating Energy Breakdown

The purpose of estimating the energy breakdown is to attribute to each energy sink its share of the energy consumption. Quanto uses weighted multivariate least squares to estimate the power draw of each energy sink. The input to this offline regression process is a log like the one shown in Table 7.2. This log records, for each interval during which the power states are same, the aggregate energy consumed during that interval (ΔE), the length of the interval (Δt), and the power states of all devices during the interval ($\alpha_1, \ldots, \alpha_n$).

We estimate the power draw of the i-th energy sink as follows. First we group all intervals from the log that have the same power state j (a particular setting of $\alpha_1, \ldots, \alpha_n$), adding the time t_j and energy E_j spent at that power state. For each power state j, possibly ranging from 1 to 2^n, we determine the average aggregate power y_j:

$$y_j = E_j / t_j,$$

and collect them in a column vector $\mathbf{Y}$ over all j:

$$\mathbf{Y} = \begin{bmatrix} y_1 & \dots & y_j & \dots & y_m \end{bmatrix}^T .$$

Due to quantization effects in both our time and energy measurements, the confidence in y_j increases with both E_j and t_j. Correspondingly, we use a weight, $w_j = \sqrt{E_j t_j}$ for each estimate in the regression, and group them in a diagonal weight matrix $\mathbf{W}$. We use the square root because, for a constant power level, E_j and t_j are linearly dependent.

We first collect the observed power states $\alpha_{j,i}$ in a matrix $\mathbf{X}$:

$$\mathbf{X} = \begin{bmatrix} \alpha_{1,1} & \dots & \alpha_{1,n} \\ \vdots & \ddots & \vdots \\ \alpha_{m,1} & \dots & \alpha_{m,n} \end{bmatrix}$$

Then, the unknown power draws are estimated:

$$\mathbf{\Pi} = (\mathbf{X}^T\mathbf{W}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{W}\mathbf{Y},$$

and finally, the residual errors are given by:

$$\epsilon = \mathbf{Y} - \mathbf{X}\mathbf{\Pi} .$$

This chapter described the two main key parts of Quanto, activity and energy tracking. To evaluate the functionality and performance of Quanto, we implemented the framework in TinyOS. Implementing our approach required small changes to six OS abstractions – timers, TinyOS tasks, arbiters, network stack, interrupt handlers, and device drivers. We changed 22 files and 171 lines of code for core OS primitives, and 16 files and 148 lines of code for representative device drivers, to support activity tracing and exposing of power states. The next chapter presents some results of the

applications of Quanto to real applications in TinyOS, as well as some evaluations of its costs and overhead. In light of the evaluation, we then discuss tradeoffs in Quanto's design, its limitations, related and future work.

Chapter 8

Evaluating Quanto

In this chapter we evaluate our concrete implementation of Quanto, as described in the previous chapter, in TinyOS. We first look at two simple applications, *Blink* and *Bounce*, that illustrate how Quanto combines activity tracking, power-state tracking, and energy metering into a complete energy map of the application. In Section 8.1 we use the first, *Blink*, to calibrate Quanto against ground truth provided by an oscilloscope, and in Section 8.2.1 we use it as an example of a multi-activity, single-node application. In Section 8.2.2, we use the second, *Bounce*, as an example with activities that span different nodes. Section 8.3 looks at three case studies in which Quanto exposes real-world effects and costs of application design decisions, and lastly we quantify, in Sections 8.4, some of the costs involved in using Quanto itself. In these experiments processed Quanto data with a set of tools we wrote to parse and visualize the logs. We used GNU Octave to perform the regressions. We conclude the chapter with a discussion, in Section 8.5, on the tradeoffs in Quanto's design, some of its limitations, related work, and further research that Quanto enables.

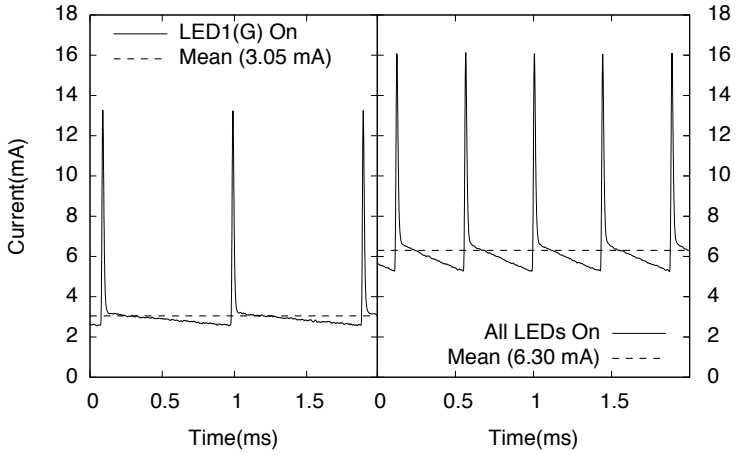


Figure 8.1: Current over time for two states of Blink recorded with the oscilloscope, showing the mean current and the iCount pulses that Quanto accumulates.

8.1 Calibration

We set up a simple experiment to calibrate Quanto against the ground truth provided by a digital oscilloscope. The goal is to establish that Quanto can indeed measure the aggregate energy used by the mote, and that the regression does separate this energy use by hardware components.

We use Blink, the *hello world* application in TinyOS. Blink is very simple; it starts three independent timers with intervals of 1, 2, and 4s. When these timers fire, the red, green, and blue LEDs are toggled, such that in 8 seconds Blink goes through 8 steady states, with all combinations of the three LEDs on and off. The CPU is in its sleep state during these steady states, and only goes active to perform the transitions.

Using the Hydrowatch board (*cf.* Section 7.4.2), we connected a Tektronix MSO4104 oscilloscope to measure the voltage across a 10Ω resistor inserted between iCount circuit and the mote power input. We measured the voltage provided by the regulator for the mote to be 3.0V.

X				**Y**
L0	L1	L2	C	$I(mA)$
0	0	0	1	0.74
1	0	0	1	3.32
0	1	0	1	3.05
1	1	0	1	5.53
0	0	1	1	1.62
1	0	1	1	4.15
0	1	1	1	3.88
1	1	1	1	6.30

(a)

	Π
	$I(mA)$
LED0	2.50
LED1	2.23
LED2	0.83
Const.	0.79

(b)

XΠ
$I(mA)$
0.79
3.29
3.02
5.53
1.62
4.12
3.85
6.36

(c)

Table 8.1: Oscilloscope measurements of the current for the steady states of Blink (a), and the results of the regression with the current draw per hardware component (b). (c) shows the currents that the model predicts for each state. The relative error, given by $(\|\mathbf{Y} - \mathbf{X\Pi}\|/\|\mathbf{Y}\|)$*, is 0.83%.*

We confirmed the result from [44] that the switching frequency of iCount varies linearly with the current. Figure 8.1 shows the current for two sample states of Blink. This curve has a wealth of information: from it we can derive both the switching frequency of the regulator, which is what Quanto measures directly, and the actual average current, I_{avg}. We verified over the 8 power states that I_{avg}, in mA, and the switching frequency f_{iC}, in kHz, have a linear dependency given by $I_{avg} = 2.77 f_{iC} - 0.05$, with an R^2 value of 0.99995. We can infer from this that each iCount pulse corresponds, in this hardware, at 3 V, to 8.33 uJ. We also verified that I_{avg} was stable during each interval.

Lastly we tested the regression methodology from Section 7.4.5, using the average current measured by the oscilloscope in each state of Blink and the external state of the LEDs as the inputs. We also added a constant term to account for any residual current not captured by the LED state. Table 8.1 shows the results, and the small relative error indicates that for this case the linearity assumptions hold reasonably well, and that the regression is able to produce a good breakdown of the

power draws per hardware device.

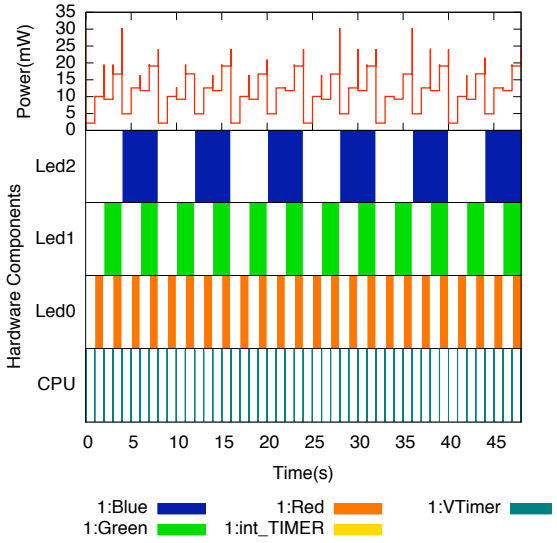
8.2 Two Illustrative Examples

8.2.1 Blink

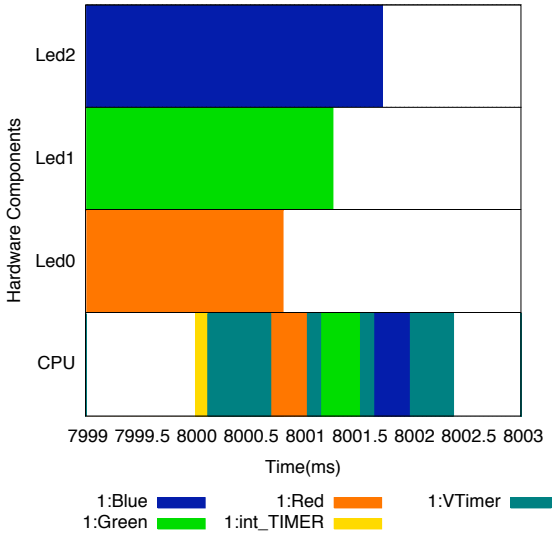
We instrumented Blink with Quanto to verify the results from the calibration and to demonstrate a simple case of tracking multiple activities on a single node. We divided the application into 3 main activities: Red, Green, and Blue, which perform the operations related to toggling each LED. Each LED, when on, gets labeled with the respective activity by the CPU, such that its energy consumption can be charged to the correct activity. We also created an activity to represent the managing of the timers by the CPU (VTimer). We recorded the power states of each LED (simply on and off), and consider the CPU to only have two states as well: active, and idle.

Figures 8.2(a) and (b) show details of a 48-second run of Blink. In these plots, the X axis represents time, and each color represents one activity. The lower part of (a) shows how each hardware component divided its time among the activities. The topmost portion of the graph shows the aggregate power draw measured by iCount. There are eight distinct stable draws, corresponding to the eight states of the LEDs.

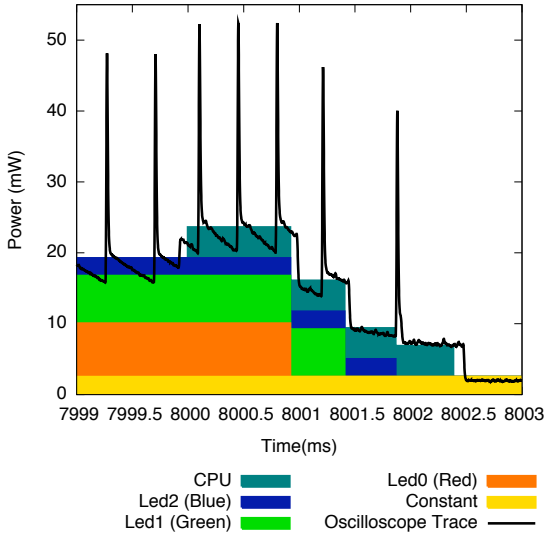
Part (b) zooms in on a particular state transition spanning 4 ms, around 8 s into the trace, when all three LEDs simultaneously go from their on to off state, and cease spending energy on behalf of their respective activities. At this time scale it is interesting to observe the CPU activities. For clarity, we did not aggregate the proxy activities from the interrupts into the activities they are bound to. At 8.000 s the timer interrupt fires, and the CPU gets labeled with the int_TIMERB0 and VTimer activities. VTimer, after examining the scheduled timers, yields to the Red, Green, and



(a) Power draw measured by Quanto and the activities over time for each hardware component.



(b) Detail of a transition from all on to all off, with the activities for each hardware component.



(c) Stacked power draw from the regression for the hardware components, overlaid with the oscilloscope-measured power.

Figure 8.2: Activity and power profiles for a 48-second run of Blink.

Activities	Hardware Components - Time(s) LED0	LED1	LED2	CPU
1:Red	24.01	0	0	0.0176
1:Green	0	24.00	0	0.0091
1:Blue	0	0	24.00	0.0045
1:Vtimer	0	0	0	0.0450
1:int_Timer	0	0	0	0.0092
1:Idle	23.99	24.00	24.00	47.9169
Total	48.00	48.00	48.00	48.0024

(a) Time break down.

	Hardware Components - $\mathbf{\Pi}$ LED0	LED1	LED2	CPU	*Const.*
I_{avg}(mA)	2.51	2.24	0.83	1.43	0.83
P_{avg}(mW)	7.53	6.71	2.49	4.29	2.48

(b) Result of the regression.

	$\sum E_{HW}$(mJ)
LED0	180.71
LED1	161.06
LED2	59.84
CPU	0.37
Const.	119.26
Total	521.23

(c) Total Energy per Hardware Component.

Activities	$\sum E_{act}$(mJ)
1:Red	180.78
1:Green	161.10
1:Blue	59.86
1:Vtimer	0.19
1:int_Timer	0.04
1:Idle	0.00
Const.	119.26
Total	521.23

(d) Total Energy per Activity.

Table 8.2: Where the joules have gone in Blink. *The tables show how activities spend time on hardware components (a), the regression results (b), and a breakdown of the energy usage by activity (c) and hardware component (d).*

Blue activities in succession. Each activity turns off its respective LED, clears its activities, and sets its power state to off. VTimer performs some bookkeeping and then the CPU sleeps.

Table 8.2(a) shows, for the same run, the total time when each hardware component spent energy on behalf of each activity. The CPU is active for only 0.178% of the time. Also, although the LEDs

stay on for the same amount of time, they change state a different number of times, and the CPU time dedicated to each corresponding activity reflects that overhead.

We ran the regression as described in Section 7.4.5 to identify the power draw of each hardware component. Table 8.2(b) shows the result in current and power. This information, combined with the time breakdown, allows us to compute the energy breakdown by hardware component (c), and by activity (d). The correlation between the corresponding components of Table 8.2(b) and the current breakdown in Table 8.1 is 0.99988. Note that we don't have the CPU component in the oscilloscope measurements because it was hard to identify in the oscilloscope trace exactly when the CPU was active, something that is easy with Quanto.

From the power draw of the individual hardware components we can reconstruct the power draw of each power state and verify the quality of the regression. The relative error between the total energy measured by Quanto and the energy derived from the reconstructed power state traces was 0.004% for this run of Blink.

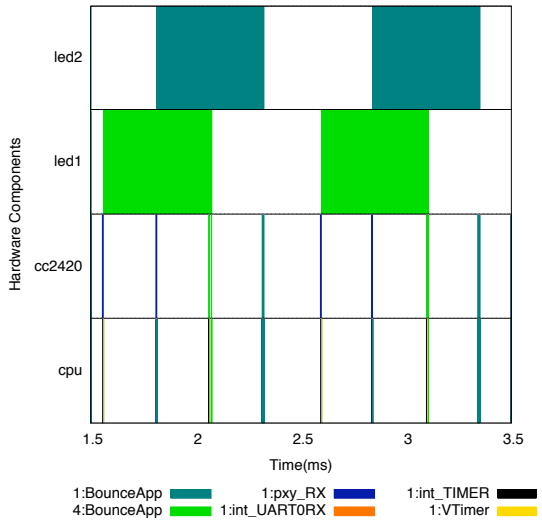
Figure 8.2(c) shows a stacked breakdown of the measured energy envelope, reconstructed from the power state time series and the results of the regression. The shades in this graph represent the different hardware components, and at each interval the stack shows which components are active, and in what proportion they contribute to the overall energy consumption. The graph also shows an overlaid power curve measured with the oscilloscope for the same run. The graph shows a very good match between the two sources, both in the time and energy dimensions. We can notice small time delays between the two curves, on the order of 100 μs, due to the time Quanto takes to record a measurement.

8.2.2 Bounce

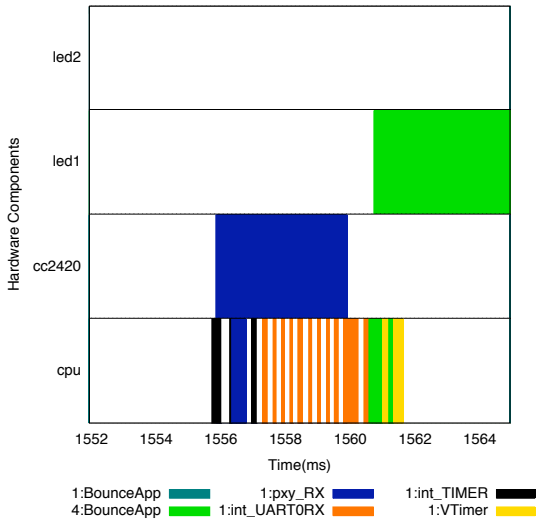
The second example we look at illustrates how Quanto keeps track of activities across nodes. Bounce is a simple application in which two nodes keep exchanging two packets, each one originating from one of the nodes. In this example we had nodes with ids 1 and 4 participate. All of the work done by node 1 to receive, process, and send node 4's original packet is attributed to the '4:BounceApp' activity. Although this is a trivial example, the same idea applies to other scenarios, like protocol beacon messages and multihop routing of packets.

Figure 8.3 shows a 2-second trace from node 1 of a run of Bounce. The log at the other node is symmetrical. On part (a) we see the entire window, and the activities by the CPU, the radio, and two LEDs that are on when the node has "possession" of each packet. In this figure, node 1 receives a packet which carries the 4:BounceApp activity, and turns LED1 on because of that. The energy spent by this LED will be attributed to node 4's original activity. The node then receives another packet, which carries its own 1:BounceApp activity. LED2's energy spending will be assigned to node 1's activity, as well as the subsequent transmission of this same packet.

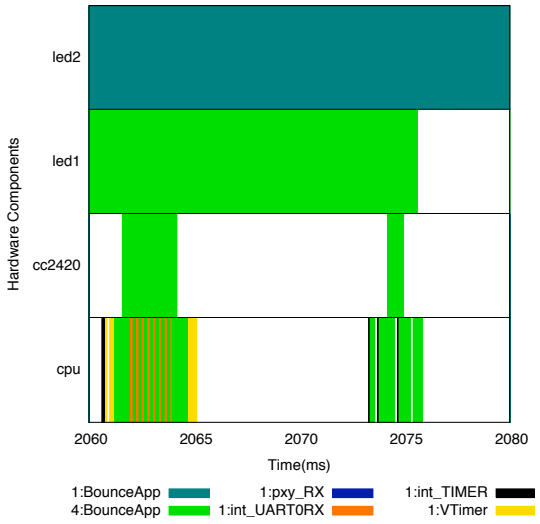
Figures 8.3(b) and (c) show in detail a packet reception and transmission, and how activity tracking takes place in these two operations. Again, we keep the interrupt proxy activities separated, although when accounting for resource consumption we should assign the consumption of a proxy activity to the activity to which it binds. The receive operation starts with a timer interrupt for the start of frame delimiter, followed by a long transfer from the radio FIFO buffer to the processor, via the SPI bus. This transfer uses an interrupt for every 2 bytes. When finished, the packet is decoded by the radio stack, and the activity in the packet can be read and assigned to the CPU. The CPU then "paints" the LED with this activity and schedules a timer to send the packet.



(a) A 2-second window on a run of Bounce at a node with id 1.



(b) Detail of a packet reception with an activity label from node 4.



(c) Detail of a packet transmission on node 1 as part of the activity started at node 4.

Figure 8.3: Activity tracking for Bounce. *Each packet carries the activity current at the time it was generated, and the receiving node executes some operations as part of that remote activity.*

Transmission in Bounce is triggered by a timer interrupt that was scheduled upon receive. The timer carries and restores the activity, and "paints" the radio. There are two main phases for transmission. First, the data is transferred to the radio via the SPI bus, and then, after a backoff interval, the actual transmission happens. When the transmission is done, the CPU then turns the LED off and sets its activity to idle.

8.3 Case Studies

Quanto allows a developer to precisely understand and quantify the effects of design decisions, and we discuss three case studies from the TinyOS codebase.

The first one is an investigation of the effect of interference from an 802.11 b/g network on the operation of low-power listening [132]. Low-power listening (LPL) is a family of duty-cycle regimes for the radio in which the receiver stays mostly off, and periodically wakes up to detect whether there is activity on the channel. If there is, it stays on to receive packets, otherwise it goes back to sleep. In the simplest version, a sender must transmit a packet for an interval as long as the receiver's sleep interval. A higher level of energy in the channel, due to interference from other sources, can cause the receiver to falsely detect activity, and stay on unnecessarily. Since 802.11 b/g and 802.15.4 radios share the 2.4 GHz band, and the former generally has much higher power than the latter, this scenario can be quite common. We used Quanto to measure the impact of such interference. We set an 802.11 b access point to operate on channel 6, with central frequency of 2.437 GHz, and programmed a TinyOS node to listen on LPL mode, first on the 802.15.4 channel 17 (central frequency 2.453 GHz), and then on channel 26 (central frequency 2.480 GHz). We set the TinyOS node to sample the channel every 500 ms, and placed it 10 cm away from the access

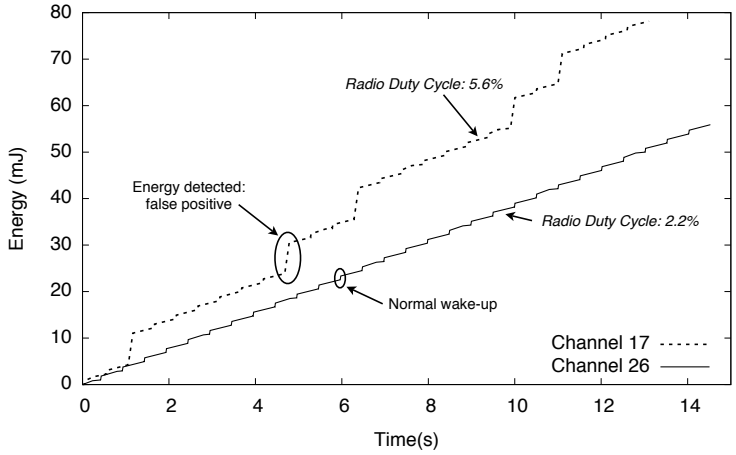


Figure 8.4: 802.11 b/g interference on the mote 802.15.4 radio. In the top curve the mote was set to the 802.15.4 channel 17, and in the bottom curve, to channel 26. These are, respectively, the closest to and furthest from the 802.11 b channel 6 used in the experiment.

point. We collected data for 5 14-second periods at each of the two channels.

We verified a significant impact of the interference: when on channel 17, the node falsely detected activity on the channel 17.8% of the time, had a radio duty cycle of 5.58$\pm$0.005%, and an average power draw of 1.43$\pm$0.08 mW. The nodes on channel 26, on the other hand, detected *no* false positives, had a duty cycle of 2.22$\pm$0.0027%, and an average power draw of 0.919$\pm$0.006 mW.

Figure 8.4 shows one measurement at each channel. The steps on the channel 17 curve are false positives, and have a marked effect on the cumulative energy consumption. Using Quanto, we estimated the current for the radio listen mode to be 18.46 mA, with a power draw of 61.8 mW (this particular mote was operating with a 3.35V switching regulator). Figure 8.5 shows two sampling events on channel 17. For both the radio and the CPU, the graph shows the power draw when active, and the respective activities. We can see the VTimer activity, which schedules the wake-ups, and the proxy receive activity, which doesn't get bound to any subsequent higher level activity. This is

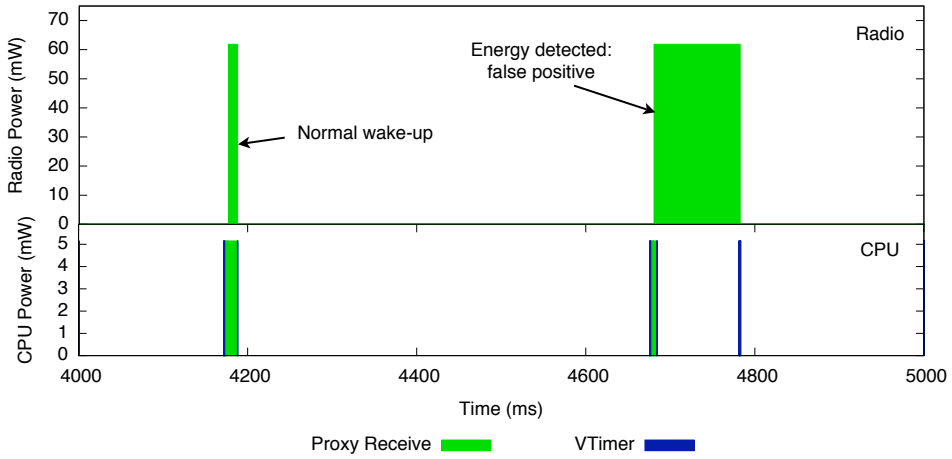


Figure 8.5: Detail of two radio wake-ups in LPL. The first is a normal wake-up with no activity, where the radio quickly returns to sleep. The second is a false-positive activity detection: the CPU keeps the radio on for 100 ms, and no packet is received.

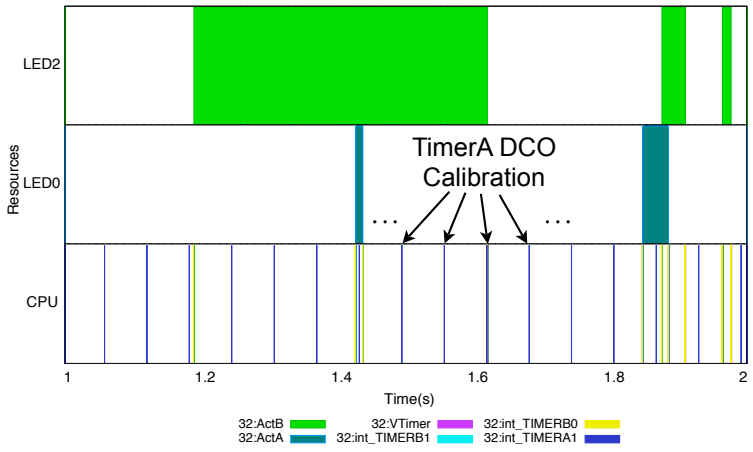


Figure 8.6: An unexpected result from instrumenting a simple application with Quanto: we noticed that a particular timer interrupt was firing 16 times per second for oscillator calibration, even when such calibration was unnecessary.

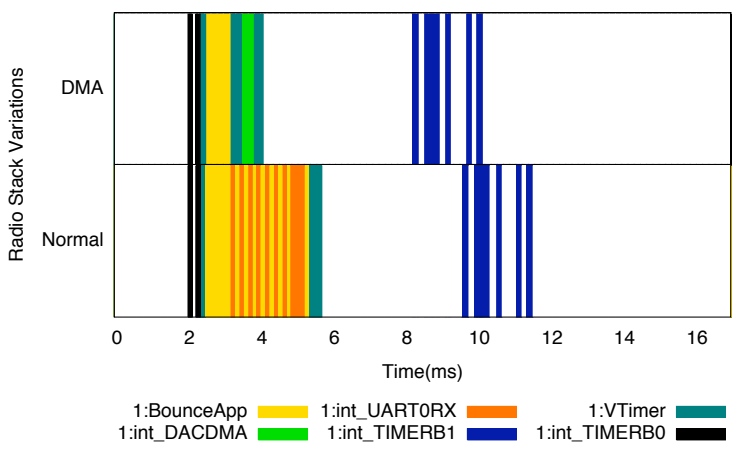


Figure 8.7: Timing behavior of a packet transmission using interrupt-driven and DMA-based communication between the CPU and the radio chip. Quanto allows the developer to understand the precise timing behavior of both options.

a simple example, but Quanto would be able to distinguish these activities even if the node were performing other tasks.

Our second example concerns an effect we noticed when we instrumented a simple timer-based application on a single node. A particular timer interrupt, TimerA1, was firing repeatedly at 16Hz, as can be seen in Figure 8.6. This timer is used for calibrating a digital oscillator, which is not needed unless the node requires asynchronous serial communication. However, it was set to be always on, a behavior that surprised many of the TinyOS developers. The lack of visibility into the system made this behavior go unnoticed.

Our last example studies the effect of a particular setting of the radio stack: whether the CPU communicates with the radio chip using interrupts or a DMA channel. Figure 8.7 shows the timings captured by Quanto for a packet transmission, using both settings.

From the figure it is apparent that the DMA transfer is at least twice as fast as the interrupt-

driven transfer. This has implications on how fast one can send packets, but more importantly, can influence the behavior of the MAC protocol. If two nodes A and B receive the same packet from a third node, and need to respond to it immediately, and if A uses DMA while B uses the interrupt-driven communication, A will gain access to the medium more often than B, subverting MAC fairness.

8.4 Costs

We now look at some of the costs associated with our prototype implementation of Quanto. These are summarized in Table 8.3.

8.4.1 Cost of logging.

The design of Quanto decouples generating event information, like activity and power state changes, from tracking the events. We currently record a log of the events for offline processing. The cost of logging is divided in two parts, one synchronous and one asynchronous. Recording the time and energy for each event has to be done synchronously, as close to the event as possible. Dealing with the recorded information can be done asynchronously.

It is very important to minimize the cost of synchronously recording each sample, as this both limits the rate at which we can capture successive events, and delays operations which must be processed quickly. Our current implementation records a 12-byte log entry for each event, described in Figure 8.8. We measured the cost of logging to RAM to be 101.7 μs, using the same technique as in [44]. At 1MHz, this translates to 102 cycles. This time includes 24 μs to read the iCount value, and 19 μs to read a timer value.

```
typedef struct entry_t {
  uint8_t type;     // type of the entry
  uint8_t res_id;   // hardware resource for entry
  uint32_t time;    // local time of the node
  uint32_t ic;      // icount: cumulative energy
  union {
    uint16_t act;         //for ctx changes
    uint16_t powerstate;  //for powerstate changes
  };
} entry_t;
```

Figure 8.8: The structure for the activity and powerstate log entry.

Buffer Size	800 samples
Sample Size	12 bytes
Cost of Logging	102 cycles @ 1MHz
Call Overhead	41 cycles
Read Timer	19 cycles
Read iCount	24 cycles
Others	18 cycles

Table 8.3: Costs associated with logging to RAM.

Because Quanto uses the CPU to keep track of state and to log changes to state, using it incurs a cost by delaying operations on the CPU, and spending more energy. For the run of Blink in Section 8.2, we logged 597 messages over 48 seconds. The total time spent on the logging itself was 60.71 ms, corresponding to 71.05% of the *active* CPU time, but only 0.12% of the total CPU time. The total energy spent with logging, assuming that logging is using the CPU and the *Constant* terms in the regression results, was 0.41 mJ, or 0.08% of the total energy spent. Although the 71% number is high, the majority of applications in these sensor network platforms strive to reduce the CPU duty cycle to save energy, and we expect the same trend of long idle periods to amortize the cost of logging.

The above numbers only concern the synchronous part. We still have to get the data out of the node for the current approach of offline analysis. We have two implementations for this. The first

records messages to a fixed buffer in RAM that holds 800 log entries, periodically stops the logging, and dumps the information to the serial port or to the radio. The advantage of this is that the cost of logging, during the period being monitored, is only the cost of the synchronous part.

The second approach allows continuous logging. The processor still collects entries to the memory buffer, and schedules a low priority task to empty the log. This happens only when the CPU would otherwise be idle. Messages are written directly to an output port of the microprocessor, which drives an external synchronous serial interface. Like the Unix `top` application, Quanto can account for its own logging in this mode as its own activity. For the applications we instrumented, it used between 4 and 15% of the CPU time.

The rate of generated data from Quanto largely depends on the nature of the workload of the application. For the classes of applications that are common in embedded sensor networks, of low data-rate and duty cycle, we believe the overheads are acceptable.

8.4.2 Instrumentation costs

Finally, we look at the burden to instrument a system like TinyOS to allow tracking and propagation of activities and power states. Table 8.4 lists the main abstractions we had to instrument in TinyOS to achieve propagation of activity labels in our platform, and shows that the changes are highly localized and relatively small in number of lines of code.

The complexity of the instrumentation task varies, and some device drivers with shadowed state that represents volatile state in peripherals can be more challenging to instrument. The CC2420 radio is a good example, as it has several internal power states and does some processing without the CPU intervention. Other devices, like the LEDs and simple sensors, are quite easier. We found that once the system is instrumented, the burden to the application programmer is small, since all

	Files	Diff LOC	
Modified Code			
Tasks	2	25	Concurrency
Timers	2	16	Deferral
Arbiter	5	34	Locks
Interrupts	11	88	
Active Msg.	2	8	Link Layer
LEDs	2	33	Device Driver
CC2420 Radio	11	105	Device Driver
SHT11	3	10	Sensor
New code	28	1275	Infrastructure

Table 8.4: The cost of instrumenting most core primitives for activity and power tracking in TinyOS, as well as some representative device drivers, is low in terms of lines of code. New code represents the infrastructure code for keeping track of and logging activities and power states.

that needs to be done is marking the beginning of relevant activities, which will be tracked and logged automatically.

8.5 Discussion

We now discuss some of the the design tradeoffs and limitations of the approach, and point to some research directions enabled by this work.

8.5.1 Design Tradeoffs

Our design decisions in Quanto reflect some tradeoffs, motivated by the resource constraints of the platform and by the set of questions that the framework tries to address.

Activity model. The activity model in Quanto is simpler than the general concurrent event model from X-Trace, described in Chapter 3. In Quanto we do not distinguish individual events that compose an activity, but group the resource consumption of *all* events that comprise an activity.

This can still directly answer questions about the aggregate resource consumption of an activity, and allows for much less overhead in the instrumentation.

Another important design decision in Quanto is that activities are not hierarchical, as opposed, for example, to the model used for profiling in tools like gprof [72]. While giving more flexibility, representing hierarchies would mean that the system would propagate stacks of activity labels instead of single labels, a significant increase in overhead and complexity. If a module C does work on behalf of two activities, A and B, the instrumenter has two options: to give C its own activity, or to have C's operations assume the activity of the caller.

Logging vs. counting. Quanto currently logs every power state and activity context change which can result in large volume of trace data. The data are useful for reconstructing a fine-grained timeline and tracing causal connections, but this level of detail may be unnecessary in many cases. The design, however, clearly separates the event generation from the event consumption. An alternative would be to maintain a set counters on the nodes, accumulating time and energy spent per activity. In our initial exploration we decided to examine the full dataset offline, and leave as future work to explore performing the regression and accounting of resources online, which would make the memory overhead fixed and practically eliminate the logging overhead.

Platform hardware support. All the data in this chapter were collected using the HydroWatch platform, but our experiences suggested that a more tailored design would be useful. In particular, we had the options of storing the logs in RAM, which has little impact but limited space, or logging to a processor port, which has a slightly higher cost and can be intrusive at very high loads. It is possible to explore different logging options with hardware support, like using hardware FIFO chips to offload logging from the processor.

8.5.2 Limitations

While Quanto provides enhanced visibility into distributed energy usage, it has several limitations. These arise from the assumptions that we make about its operating environment, which we briefly sketch below.

Constant per-state power draws. The regression techniques used to estimate per-component energy usage assume the power draw of a hardware component is approximately constant in each power state. Fortunately, we verified that this assumption largely holds for the platform we instrumented, by looking at different length sampling intervals for each state. The regression may not work well when this assumption fails, but we leave quantifying this for future work.

Linear independence. The regression techniques also assume that tracking power states over time produces a set of linearly independent equations. If this is not the case, for example if unrelated actions always occur together, then regression is unlikely to disambiguate their energy usage. As a work around, custom routines can be written to exercise different power states independently.

Modifications to systems. Quanto requires the OS, including device drivers, and applications, to be modified to perform tracking. The modifications to the system, however, can be shared among all applications, and the modifications to applications are, in most cases, simple. Device drivers have to be modified so that they expose the power states of the underlying hardware components. If hardware power states are not observable, estimation errors may occur.

Energy usage visibility. Our approach may not generalize to systems with sophisticated power supply filtering (e.g. power factor correction or large capacitors) because these elements introduce a potentially non-linear phase delay between real and observed energy usage over short time scales, making it difficult to correlate short-lived activities with their energy usage.

Hardware energy metering. Our proposed approach requires hardware support for energy metering, which may not be available on some platforms. Fortunately, the energy meter design we use may be feasible on many systems that use pulse-frequency modulated switching regulators. However, even if hardware-based energy metering is not available, a software-based approach using hardware power models may still provide adequate visibility for some applications.

8.5.3 Enabled Research

Finding energy leaks. A situation familiar to many developers is discovering that an application draws too much power but not knowing why. Using Quanto, developers can visualize energy usage over time by hardware component, allowing one to work backward to find the offending code that caused the energy leak.

Tracking butterfly effects. In many distributed applications, an action at one node can have network-wide effects. For example, advertising a new version of a code image or initiating a flood will cause significant network-wide action and energy usage. Even minor local actions, like a routing update, can ripple through the entire network. Quanto can trace the causal chain from small, local cause to large, network-wide effect.

Real time tracking. An extension of the framework can include performing the regression online, and replacing the logging with accumulators for time and energy usage per activity. This approach would have significantly reduced bandwidth and storage requirements, and could be used as an always on, network-wide energy profiler analogous to `top`.

Energy-Aware Scheduling. Since Quanto already tracks energy usage by activity, an extension to the operating system scheduler would enable energy-aware policies like equal-energy scheduling for threads, rather than equal-time scheduling.

Continuous Profiling. Quanto log entries are lightweight enough that continuous profiling is possible with even a modest speed logging back-channel [10].

8.6 Related Work

Our techniques borrow heavily from the literature on energy-aware operating system, power simulation tools, power/energy metering, power profiling, resource containers, and distributed tracing. In this Chapter 2 we explored the design space for distributed visibility tools, and in this section we focus more specifically on work related to Quanto in the area of power profiling in systems.

ECOSystem [180] proposes the Currentcy model which treats energy as a first class resource that cuts across all existing system resources, such as CPU, disk, memory, and the network in a unified manner. Quanto leverages many of the ideas developed in ECOSystem, like tracking power states to allocate energy usage or employing resource containers as the principal to which resource usage is charged. But there are important differences as well. ECOSystem uses offline profiling to relate power state and power draw, and uses a model for runtime operation. In contrast, Quanto tracks the actual energy used at runtime, which is useful when environmental factors can affect energy availability and usage. While ECOSystem tracks energy usage on a single node, Quanto transparently tracks energy usage across the network, which allows network-wide effects to be measured. Finally, the focus of the two efforts is different although similar techniques are used in both system.

Eon is a programming language and runtime system that allows paths or flows through the program to be annotated with different energy states [149]. Eon's runtime then chooses flows to execute, and their rates of execution, to maximize the quality of service under available energy

constraints. Eon, like Quanto, uses real-time energy metering but attributes energy usage to flows, which are similar to the activities that Quanto uses.

Several power simulation tools exist that use empirically-generated models of hardware behavior. PowerTOSSIM [147] uses same-code simulation of TinyOS applications with power state tracking, combined with a power model of the different peripheral states, to create a log of energy usage. PowerTOSSIM provides visibility into the power draw based on its model of the hardware, but it does not capture the variability common in real hardware or operating environments, or simulate a device's interactions with the real world. Quanto also addresses a different problem than PowerTOSSIM: tracing the energy usage of logical activities rather than the time spent in software modules.

The challenge in taking measurements in low-power, embedded systems that exhibit bursty operation is that until recently, the performance of available metering options was simply too poor, and the power cost was simply too high, to use in actual deployments. Traditional instrument-based power measurements are useful for design-time laboratory testing but impractical for everyday run-time use due to the cost of instruments, their physical size, and their poor system integration [55, 50, 174]. Dedicated power metering hardware can enable run-time energy metering but they too come with the expense of increased hardware costs and power draws [30, 87]. Using hardware performance counters as a proxy power meter is possible on high-performance microprocessors like the Intel Pentium Pro [89] and embedded microprocessors like the Intel PXA255 [35]. Quanto addresses these challenges with a new design based on a switching regulator [44].

Of course, if a system employs only one switching regulator, then the energy usage can be measured only in the aggregate, rather than by hardware component. This aggregated view of energy

usage can present some tracking challenges as well. One way to track the distinct power draws of the hardware components is to instrument their individual power supply lines [168, 153]. These approaches, however, are best suited to bench-scale investigations since they require extensive per-system calibration and the latter requires considerable additional hardware which would dominate the system power budget in our applications.

The Rialto operating system [88] introduced *activities* as the abstraction to which resources are allocated and charged. Resource Containers [18] use a similar notion, and acknowledge that there is a mismatch between traditional OS resource principals, namely threads and processes, and independent activities, especially in high performance network servers. Quanto borrows the concept of activities and extends them across all hardware components and across the nodes in a network.

Quanto borrows from earlier approaches on profiling and tracing distributed systems, described in Chapter 2, and applies them to the problem of tracking network-wide energy usage in embedded systems, where resource constraints and a energy consumption by hardware devices raise a number of different design tradeoffs.

8.7 Summary

The techniques developed and evaluated in this and the previous chapters – breaking down the aggregate energy usage of a system by hardware component, causally tracking the energy usage of programmer-defined activities, and tracking the network-wide energy usage due to node-local actions – collectively provide visibility into when, where, and why energy is consumed both within a single node and across the network. While we showed some example scenarios in which Quanto can help identify anomalies and understand energy consumption across the platform, we believe this

unprecedented visibility into energy usage will enable empirical evaluation of the energy-efficiency claims in the literature, provide ground truth for lightweight approximation techniques like counters, and enable energy-aware operating systems research.

Chapter 9

Conclusion

We are building distributed systems of unprecedented scale and complexity, and are increasingly dependent on them. This exacerbates the importance of tools and techniques that improve our ability to understand how these systems work, and especially how they fail. The two frameworks that we presented in this dissertation, X-Trace and Quanto, and the underlying principle of following the execution of distributed systems in a general and lightweight way, are steps in this direction. In this concluding chapter we present a summary of our contributions, the main limitations of our techniques, and some future avenues of research that our work opens up.

9.1 Summary of Contributions

Both X-Trace and Quanto share the same underlying principle, of attaching metadata to the paths that a given execution follows in a system. We demonstrate that it is feasible to retrofit this traceability in a number of existing systems, and that the visibility we gain into the execution can be useful for programmers, operators, and users.

X-Trace provides a simple model for deterministically capturing the causal relation among arbitrary events in a distributed system. Because it places few assumptions on the traced system, other than that it be able to propagate constant-sized metadata and log messages to a reporting system outside of the datapath, it can be integrated into multiple components and layers of a system, even in an incremental way and by different parties. X-Trace has a simple and uniform model for representing the execution as a partial order graph, and provides a unified interface for instrumenting components, in both thread- and event-based systems.

With X-Trace we demonstrated examples of fault isolation in the sample scenarios of Chapter 5, and in the Coral Content Distribution Network, in Chapter 6. In the same chapter, we demonstrated the use of X-Trace to disambiguate the causes of performance problems with similar symptoms, and to identify a number of long-standing bugs both in Coral and in the Oasis Anyscast service.

Quanto, which we presented in Chapters 7 and 8, has three main contributions: first, the ability to track the execution among nodes in a wireless embedded network of sensors to identify causally related events as part of meaningful activities. Second, a lightweight energy tracking framework that can be widely deployed and can separate the energy consumed by each "energy sink" in the system from an aggregate measure of energy. Third, the combination of the activity and energy tracking, which allows for a detailed, multidimensional breakdown of the energy in a way that is meaningful to the programmer.

We have demonstrated the feasibility of Quanto on simple applications on TinyOS. We demonstrated that Quanto can be calibrated to a high level of precision, that it can produce meaningful breakdowns of energy usage by activity and peripheral, and that the tracking of activities works across nodes. We also found interesting behaviors, in three case studies, that were surprising to the

developers.

Lastly, we have released both X-Trace and Quanto's source code on SourceForge,[1] and hope that other researchers and practitioners can use them to instrument their own systems.

9.2 Limitations

Of course, our techniques are not without their limitations. One important limitation is the need to modify existing system components to retrofit the traceability we need for following the execution.

In X-Trace, for example, adding metadata to some protocols, like UDP or Ethernet, is difficult. Another example is that even though we demonstrated that this can be done for DNS, there is a great amount of DNS code, including code that is part of the firmware of routers, switches, and load balancers, for example, that is very hard or impossible to modify.

In some cases, we can still trace 'around' an unmodified component, treating it as a black box. For example, if a web server is being traced and has to call an external database that has not been instrumented, we can still record the sending of the request and link it with the receiving of the response.

Another limitation of X-Trace reports, as proposed, is that the report format may be too heavy-weight if the granularity of tracing is too fine. For example, if we add tracing to IP packets and have a router report on annotated packets, the information generated may be larger than the packet itself if we use the text-based report format. In cases like these specialized reports would have to be generated in a compact binary form, and later converted to a compatible format for the interesting

[1] X-Trace's implementation for Java and C++ can be downloaded from `http://sourceforge.net/projects/xtraceprj`, and Quanto can be downloaded from the TinyOS project in SourceForge, at `http://sourceforge.net/projects/tinyos`, in the `tinyos-2.x-contrib/berkeley/quanto/` directory.

cases.

In the case of Quanto, for some devices the power states may not be visible to the CPU, making tracking the power states difficult. Also, although registering events in Quanto is fast, this time can be on the same order of magnitude as some interrupt service routines. If an activity has sustained interrupt activity for a long period of time, the probe-effect of Quanto can become significant. Lastly, we chose to use a model of activities that is not hierarchical. The effect of this decision is that we have to choose a priori how to treat delegated operations: either as their own activity or as doing work on behalf of the calling activity. This also makes it difficult to deal with re-entrant interrupts, which don't happen in the platform we instrumented.

Our approach may not be necessary when aggregate or approximate answers may be sufficient. Systems like Orion [32], which provide a set of dependencies of a given service, may be enough to guide operators in finding problems with running services. Also, it may present too much of a burden for some questions that can be answered by only looking at a single point in a system, or only at a single layer, for example. The number of questions that cannot easily be answered by these methods, however, is bound to grow as the complexity and scale of the distributed systems we deploy continue to increase.

9.3 Future Work

In spite of these limitations, there are many situations in which both X-Trace and Quanto can be very helpful. Both frameworks open some interesting possibilities of future work. We can view them as research that enables further research, by providing instruments with which new questions, about new systems, can be asked and answered.

For both X-Trace and Quanto, there are similarities, as we mentioned in Chapter 2, between tracking causality and label propagation in distributed information flow control systems such as Flume [96], Asbestos [167], and D-Star [179]. We plan to explore these similarities and the possibility of integrating the two approaches. We now discuss some possibilities of future work more specific to the two frameworks.

9.3.1 X-Trace

In the case of X-Trace, the task-graphs produced are richly annotated histories of specific executions, that can seamlessly integrate events from different layers, components, machines. We have only scratched the surface on how to process them, and to use X-Trace at scale in large systems requires that we do this efficiently. There is the possibility of new processing algorithms to answer specific queries on these graphs efficiently. We also need to be able to compare, cluster, and summarize them efficiently.

The visualization of X-Trace data, especially of large and/or many traces is important. X-Trace has the native ability to represent hierarchical structures, as redundant edges can naturally abstract entire subgraphs. There is plenty of room for the development of interactive interfaces that allow the efficient exploration of very large amounts of X-Trace information.

Another interesting problem is how to automatically detect graphs that represent normal executions versus those that represent anomalies. Ideally this detection would be unsupervised, and as close to the source of the events as possible. Pip's expectation checking [139], and Pinpoint's root cause identification [31] are good starting points. If detected early enough, normal executions would not need to be stored for long periods of time. Chun et al. [33] reported some initial progress on D3, a system that can answer declarative queries on X-Trace task graphs in a distributed way.

There are also many algorithms that have been developed for partial orders (e.g., see [171] for a survey), and can be applied to X-Trace graphs once we have derived comparable timestamps for the events in the graph (*cf.* Section 4.4.2).

Lastly, one important area of further research is on how to automate the instrumentation of systems. Hybrid approaches like using the protocol processors of BorderPatrol [95] are a good way to deal with yet uninstrumented components. Automatic code injection, either pre- or post-compilation, can also allow more automated instrumentation of existing systems.

9.3.2 Quanto

Quanto also gives rise to many new questions and challenges. We are in the process of instrumenting larger applications, at larger network scales, and we hope that Quanto will allow us to ask, and answer, interesting new questions.

One challenge here is how to cope with the data generated by Quanto, given the resource limitations of the nodes. For offline analysis, our current implementation requires 12-bytes per each event registered by Quanto, including activity and power state changes. Our initial results of applying very lightweight compression to these entries has reduced this to an average of three bytes per entry.

Another direction, however, is to do the analysis online on the nodes, and keep cumulative counts of profile results. This requires performing the regression on the nodes, which can be challenging given their limited computational power. Also, in large networks with many activities (recall that activities are labeled with the originating node), a node may need to choose about which activities to keep information, a problem similar to that of estimating the k most important flows in a network switch [97]. Once this online profiling is in place, we can think about performing energy budget management on the nodes at the level of activities, by changing scheduling policies and employing

admission control, based on the resources that different activities have consumed. These policies would be more intuitive to users, as activities represent meaningful groupings of operations across different peripherals and nodes.

9.4 Concluding Remarks

The two frameworks we described in this dissertation provide visibility into a wide range of distributed systems, and also open very interesting avenues for future work. When viewed as instruments, we hope that there are many questions that they will help answer. The scale and complexity of distributed systems is constantly increasing, and the relevance of tools such as these is bound to grow as well. Better tools that improve our ability to understand how the systems we build work in practice will allow us to build bigger and better systems. These will, in turn, pose new challenges for their instrumentation and understanding, an exciting co-evolution process.

Ultimately, the success of the frameworks and techniques presented here will be measured by their use and integration with systems. We hope to have provided a compelling case for new and existing systems to incorporate traceability as a first class concept, and to have shown that the value of the gained visibility will be even greater when this traceability is made to span multiple systems as they get integrated.

Bibliography

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 Usenix Annual Technical Conference*, June 2002.

[2] Anurag Agarwal and Vijay K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 19–28, New York, NY, USA, 2005. ACM.

[3] U.S. Environmental Protection Agency. Report to congress on server and data center energy efficiency: Public law 109-431. `http://energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf`, August 2007. Accessed on July, 2008.

[4] 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 '03*, New York, NY, USA, 2003. ACM Press.

[5] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[6] Alexa. `http://www.alexa.com`. Accessed on July, 2008.

[7] Amazon elastic compute cloud (ec2). `http://aws.amazon.com/ec2`. Accessed on July, 2008.

[8] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001. ACM Press.

[9] D.P. Anderson. Boinc: a system for public-resource computing and storage. In *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Nov. 2004.

[10] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *ACM Trans. Comput. Syst.*, 15(4):357–390, 1997.

[11] A. Aranya, C. P. Wright, and E. Zadok. Tracefs: A file system to trace them all. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 129–143, San Francisco, CA, March/April 2004. USENIX Association.

[12] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4033: DNS Security Introduction and Requirements, March 2005.

[13] Application Response Measurement, http://www.opengroup.org/tech/management/arm/.

[14] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. *SIGCOMM Comput. Commun. Rev.*, 37(4):13–24, 2007.

[15] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[16] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. In *Proc. SIGCOMM '96*, pages 256–269, New York, NY, USA, 1996. ACM Press.

[17] Thomas Ball and James Larus. Optimally Profiling and Tracing Programs. *ACM Transactions on Programmmg Languages and Systems*, 16(4):1319–1360, July 1994.

[18] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[19] Paul Barford and Mark Crovella. Critical Path Analysis of TCP Transactions. *ACM Transactions on Networking*, 9(3):238–248, June 2001.

[20] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modeling. In *Proc. USENIX OSDI*, 2004.

[21] Elizabeth A. Basha, Sai Ravela, and Daniela Rus. Model-based monitoring for early warning flood detection. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 295–308, New York, NY, USA, 2008. ACM.

[22] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the First Symposium on Networked Systems Design and Implementation, NSDI'04*, May 2004.

[23] Andrew Birrell. An introduction to programming with threads. Technical Report 35, Digital Systems Research Center, Palo Alto, California, 1989.

[24] R. T. Braden. RFC 1122: Requirements for Internet hosts — communication layers, October 1989. Status: STANDARD.

[25] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[26] J. D. Case, M. Fedor, M. L. Schoffstall, and C. Davin. RFC 1157: Simple network management protocol (SNMP), May 1990.

[27] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *Proceedings of the European Conference on Computer Systems (EuroSys'07)*, Lisbon, Portugal, March 2007.

[28] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proc. Middleware 2005*, pages 42–59, November 2005.

[29] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike

Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[30] Fay Chang, Keith Farkas, and Parthasarathy Ranganathan. Energy-driven statistical profiling: Detecting software hotspots. In *Workshop on Power-Aware Computer Systems*, feb 2002.

[31] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proc. International Conference on Dependable Systems and Networks*, 2002.

[32] Xu Chen, Ming Zhang, Z. Morley Mao, and Paramvir Bahl. Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, December 2008.

[33] Byung-Gon Chun, Kuang Chen, Gunho Lee, Randy H. Katz, and Scott Shenker. D3: Declarative distributed debugging. Technical Report UCB/EECS-2008-27, University of California at Berkeley, Electrical Engineering and Computer Sciences Department, March 2008. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-27.html`.

[34] Edmund M. Clarke, Orna Grumberg, and David E. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency, Reflections and Perspectives, REX School/Symposium*, pages 124–175, London, UK, 1994. Springer-Verlag.

[35] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. In *ISLPED '05: Proceedings of the 2005 inter-*

national symposium on Low power electronics and design, pages 221–226, New York, NY, USA, 2005. ACM.

[36] Brian Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Phil Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!s Hosted Data Serving Platform. In *Proceedings of the 34th International Conference on Very Large Data Bases, VLDB 2008*, Auckland, NZ, August 2008.

[37] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[38] D. Crocker. RFC 822: Standard for the format of ARPA Internet text messages, August 1982.

[39] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 137–150, Berkeley, CA, USA, 2004. USENIX Association.

[40] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: hardware support for instruction-level profiling on out-of-order processors. In *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997. IEEE Computer Society.

[41] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-*

first ACM SIGOPS symposium on Operating systems principles, pages 205–220, New York, NY, USA, 2007. ACM.

[42] R. P. Dilworth. A decomposition theorem for partially ordered sets. *The Annals of Mathematics, Second Series*, 51(1):161–166, January 1950.

[43] Jean Dollimore, Tim Kindberg, and George Coulouris. *Distributed Systems: Concepts and Design*. Addison Wesley, 4 edition, May 2005.

[44] Prabal Dutta, Mark Feldmeier, Joseph Paradiso, and David Culler. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN'08: International Conference on Information Processing in Sensor Networks*, pages 283–294, 2008.

[45] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In *Proceedings of the Sixth ACM Conference on Embedded Networked Sensor Systems (SenSys'08)*, November 2008.

[46] J. Ellson, E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and dynagraph – static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003.

[47] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 147–163, New York, NY, USA, 2002. ACM.

[48] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, pages 59–69, January 2002.

[49] Juan-Carlos Herrera et al. Mobile Century, Using GPS Mobile Phones as Traffic Sensors: A Field Experiment, November 2008.

[50] Keith I. Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M. Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. *SIGMETRICS Perform. Eval. Rev.*, 28(1):252–263, 2000.

[51] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley & Sons, New York, 3rd edition, 1970.

[52] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, Feb 1988.

[53] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[54] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles (SOSP'99)*, pages 48–63, 1999.

[55] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99: Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, page 2, Washington, DC, USA, 1999. IEEE Computer Society.

[56] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: Tracking energy in networked embedded systems. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, December 2008.

[57] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. IP options are not an option. Technical Report UCB/EECS-2005-24, EECS Department, UC Berkeley, December 9 2005.

[58] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *NSDI'07: Proceedings of the 4th USENIX/ACM Symposium on Networked Systems Design and Implementation*. USENIX, 2007.

[59] Rodrigo Fonseca, Sylvia Ratnasamy, Jerry Zhao, Cheng Tien Ee, David E. Culler, Scott Shenker, and Ion Stoica. Beacon vector routing: Scalable point-to-point routing in wireless sensornets. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, NSDI 2005*, Boston, Massachusetts, USA, May 2005.

[60] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[61] The Apache Software Foundation. Apache HTTP Server Project. `http://httpd.apache.org`. Accessed on July, 2008.

[62] Michael J. Freedman. *Democratizing Content Distribution*. PhD thesis, New York University, September 2007.

[63] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with Coral. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI 04)*, San Francisco, CA, March 2004.

[64] Michael J. Freedman, Karthik Lakshminarayanan, and David Mazières. OASIS: Anycast

for any service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI 06)*, San Jose, CA, May 2006.

[65] R. Freese, J. Jaroslav, and J. B. Nation. *Free Latice*. American Mathematical Society, 1996.

[66] Gallery photo album organizer. `http://gallery.menalto.com`. Accessed on July, 2008.

[67] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proc. USENIX Annual Technical Conference*, 2006.

[68] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[69] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM.

[70] Google app engine. `http://code.google.com/appengine/`. Accessed on July, 2008.

[71] Alla Goralčíková and Václav Koubek. A reduct-and-closure algorithm for graphs. In Jirí Becvár, editor, *MFCS*, volume 74 of *Lecture Notes in Computer Science*, pages 301–307. Springer, 1979.

[72] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, New York, NY, USA, 1982. ACM.

[73] Apache Hadoop Project. `http://hadoop.apache.org/`. Accessed on July, 2008.

[74] Robert J. Hall and Aaron J. Goldberg. Call path profiling of monotonic program resources in unix. In *Usenix-stc'93: Proceedings of the USENIX Summer 1993 Technical Conference on Summer technical conference*, pages 1–19, Berkeley, CA, USA, 1993. USENIX Association.

[75] Mark Hempstead, Nikhil Tripathi, Patrick Mauro, Gu-Yeon Wei, and David Brooks. An ultra low power system architecture for sensor network applications. In *ISCA'05: 32nd International Symposium on Computer Architecture*, 2005.

[76] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[77] Jason Hill and David E. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, nov/dec 2002.

[78] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[79] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic Program Instrumentation for Scalable Performance Tools. In *Proceedings of the Scalable High-Performance Computing Conference (SHPCC)*, pages 841–850, May 1994.

[80] Richard Honicky, Eric Paulos, Eric Brewer, and Richard White. N-SMARTS: Networked Suite of Mobile Atmospheric Real-Time Sensors. In *Proceedings of the Second ACM SIGCOMM Workshop on Networked Systems for Developing Regions - NSDR 2008*, Seattle, WA, USA, August 2008.

[81] Harry T. Hsu. An algorithm for finding a minimal equivalent graph of a digraph. *J. ACM*, 22(1):11–16, 1975.

[82] 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)*, 20(8), 2002.

[83] Alefiya Hussain, Genevieve Bartlett, Yuri Pryadkin, John Heidemann, Christos Papadopoulos, and Joseph Bannister. Experiences with a continuous network tracing infrastructure. In *Proc. MineNet '05*, New York, NY, USA, 2005. ACM Press.

[84] Selma Ikiz and Vijay K. Garg. Efficient incremental optimal chain partition of distributed program traces. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 18, Washington, DC, USA, 2006. IEEE Computer Society.

[85] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceeding of the European Conference on Computer Systems (EuroSys)*, Lisbon, Portugal, March 2007.

[86] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[87] Xiaofan Jiang, Prabal Dutta, David Culler, and Ion Stoica. Micro power meter for energy monitoring of wireless sensor networks at scale. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 186–195, New York, NY, USA, 2007. ACM Press.

[88] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the rialto operating system. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 12, Washington, DC, USA, 1995. IEEE Computer Society.

[89] Russ Joseph and Margaret Martonosi. Run-time power estimation in high performance microprocessors. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 135–140, 2001.

[90] Wagner Meira Jr. *Understanding Parallel Program Performance Using Cause-Effect Analysis*. PhD thesis, Dept. of Computer Science, University of Rochester, Rochester, NY, USA, July 1997. TR 663.

[91] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[92] Kevin Klues, Vlado Handziski, Chenyang Lu, Adam Wolisz, David Culler, David Gay, and Philip Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.

[93] Ramana Rao Kompella, Albert Greenberg, Jennifer Rexford, Alex C. Snoeren, and Jennifer Yates. Cross-layer visibility as a service. In *Proc. IV HotNets Workshop*, November 2005.

[94] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-Oriented (and Beyond) Network Architecture. In submission.

[95] Eric Koskinen and John Jannotti. Borderpatrol: isolating events for black-box tracing. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 191–203, New York, NY, USA, 2008. ACM.

[96] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2007. ACM.

[97] Abhishek Kumar, Minho Sung, Jun (Jim) Xu, and Jia Wang. Data streaming algorithms for efficient and accurate estimation of flow size distribution. *SIGMETRICS Perform. Eval. Rev.*, 32(1):177–188, 2004.

[98] Leslie Lamport. Email message sent to a DEC SRC bulletin board at 12:23:29 PDT on 28 May 87. `http://research.microsoft.com/users/lamport/pubs/distributed-system.txt`, Accessed on July 2008.

[99] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[100] H. Lauer and R. Needham. On the duality of operating system structures. In *Proc. Second International Symposium on Operating Systems*. IRIA, October 1978. Reprinted in Operating Systems Review, Vol. 12, Number 2, April 1979.

[101] Philip Levis. *Application Specic Virtual Machines: Operating System Support for User-Level Sensornet Programming*. PhD thesis, University of California, Berkeley, 2005.

[102] Apache log4j. `http://logging.apache.org/log4j/`. Accessed on July, 2008.

[103] C. Lonvick. RFC 3164: The BSD syslog Protocol, August 2001. Category: Informational.

[104] Keong Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *Communications Surveys & Tutorials, IEEE*, pages 72–93, 2005.

[105] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 131–146, New York, NY, USA, 2002. ACM.

[106] Mateusz Malinowski, Matthew Moskwa, Mark Feldmeier, Mathew Laibowitz, and Joseph A. Paradiso. Cargonet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asychronous monitoring of exceptional events. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 145–159, New York, NY, USA, 2007. ACM.

[107] John Markoff and Saul Hansell. Hiding in plain sight, google seeks more power. The New York Times, June 2006.

[108] Matthew L. Massie, Brent N. Chun, and David E. Culler. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing*, 30(7), July 2004.

[109] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the Int'l Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.

[110] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Peer-to-Peer Symposium (IPTPS 2002)*, pages 53–65, 2002.

[111] David Mazières. A toolkit for user-level file systems. In *USENIX Conference*, June 2001.

[112] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.

[113] 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*, pages 37–46, November 1995.

[114] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek see Lim, and Timothy Torzewski. Ips-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1:206–217, 1990.

[115] David L. Mills. Internet time synchronization: The network time protocol. *IEEE Transactions on Communications*, 39:1482–1493, 1991.

[116] P. V. Mockapetris. RFC 1035: Domain names — implementation and specification, November 1987.

[117] Răzvan Musăloiu-E., Chieh-Jan Mike Jiang, and Andreas Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *Proceedings of the 7th International Symposium on Information Processing in Sensor Networks (IPSN)*, 2008.

[118] Răzvan Musăloiu-E., Andreas Terzis, Katalin Szlavecz, Alex Szalay, Joshua Cogan, and Jim

Gray. Life under your feet: Wireless sensors in soil ecology. In *Proceedings of the Third Workshop on Embedded Networked Sensors (EmNets 2006)*, pages 51–55, Harvard University, Cambridge, Massachusetts, USA, May 2006.

[119] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *In Proceedings of the First Symposium on Networked Systems Design and Implementation*, pages 155–168, 2004.

[120] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[121] Nagios. `http://www.nagios.org`. Accessed on July, 2008.

[122] Cisco NetFlow Services and Applications White Paper, http://www.cisco.com/go/netflow.

[123] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS 05)*, February 2005.

[124] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996. `http://www.softpanorama.org/People/Ousterhout/Threads/`, Accessed on July, 2008.

[125] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 Annual Usenix Technical Conference*, June 1999.

[126] Shamim N. Pakzad, Gregory L. Fenves, Sukun Kim, and David E. Culler. Design and im-

plementation of scalable wireless sensor network for structural monitoring. *ASCE Journal of Infrastructure Engineering*, 14:89–101, March 2008.

[127] Insung Park and Ricky Buch. Event Tracing: Improve Debugging and Performance Tuning with ETW. *MSDN Magazine*, April 2007.

[128] Vern Paxson. On calibrating measurements of packet transit times. *SIGMETRICS Perform. Eval. Rev.*, 26(1):11–21, 1998.

[129] M. Petrova, J. Riihijarvi, P. Mahonen, and S. Labella. Performance study of ieee 802.15.4 using measurements and simulations. In *Wireless Communications and Networking Conference 2006 (WCNC 2006)*, volume 1, pages 487–492, April 2006.

[130] Emmanuel Pietriga. A toolkit for addressing hci issues in visual language environments. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 00:145–152, 2005.

[131] Planetlab. `http://www.planet-lab.org`. Accessed on July, 2008.

[132] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.

[133] Joseph Polastre, Jonathan Hui, Philip Levis, Jerry Zhao, David Culler, Scott Shenker, and Ion Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 76–89, New York, NY, USA, 2005. ACM.

[134] Geroge Manning Porter. *Improving Distributed Application Reliability with End-to-End Datapath Tracing*. PhD thesis, University of California at Berkeley, 2008.

[135] J. Postel. RFC 791: Internet Protocol, September 1981.

[136] Powerdns recursor. `http://www.powerdns.com/`. Accessed on July, 2008.

[137] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard M. Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172, San Diego, CA, USA, August 2001.

[138] John Reumann and Kang G. Shin. Stateful distributed interposition. *ACM Trans. Comput. Syst.*, 22(1):1–48, 2004.

[139] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[140] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. Wap5: black-box performance debugging for wide-area systems. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 347–356, New York, NY, USA, 2006. ACM.

[141] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. Opendht: a public dht service and its uses. In *SIGCOMM*

'05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications, pages 73–84, New York, NY, USA, 2005. ACM.

[142] D. Robinson and K. Coar. The Common Gateway Interface (CGI) Version 1.1, October 2004.

[143] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[144] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[145] Salesforce. `http://www.salesforce.com`. Accessed on July, 2008.

[146] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[147] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner-Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, 2004.

[148] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. In *International Colloquium on Automata, Languages and Programming on Automata, languages and programming*, pages 376–386, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[149] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *Sen-*

Sys '07: Proceedings of the 5th international conference on Embedded networked sensor systems, pages 161–174, 2007.

[150] Splunk, http://www.splunk.com.

[151] R. Srinivasan. RFC 1831: RPC: Remote procedure call protocol specification version 2, August 1995.

[152] S. M. Srinivashan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference*, 2004.

[153] Thanos Stathopoulos, Dustin McIntire, and William Kaiser. The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes. In *IPSN'08: International Conference on Information Processing in Sensor Networks*, pages 383–394, 2008.

[154] Pat Stephenson. Dapper: It's 11pm and do you know where your RPC is? Google Technical Talk. February 28th, 2008, Google Pittsburgh Office.

[155] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proc. SIGCOMM '02*, pages 73–86, New York, NY, USA, 2002. ACM Press.

[156] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM '01*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[157] Ion Stoica, Robert Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan.

Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160, San Diego, CA, USA, August 2001.

[158] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons From A Sensor Network Expedition. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN)*, 2004.

[159] Igor Talzi, Andreas Hasler, Stephan Gruber, and Christian Tschudin. PermaSense: Investigating Permafrost with a WSN in the Swiss Alps. In *Proceedings of the Fourth Workshop on Embedded Networked Sensors (EmNets)*, 2007.

[160] Jay Taneja, Jaein Jeong, and David Culler. Design, modeling, and capacity planning for micro-solar power sensor networks. *International Conference on Information Processing in Sensor Networks*, 0:407–418, 2008.

[161] Tcpdump. `http://www.tcpdump.org`. Accessed on July, 2008.

[162] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 3–14, New York, NY, USA, 2006. ACM.

[163] Gilman Tolle and David Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proceedings of the Second European Workshop of Wireless Sensor Netw orks (EWSN)*, 2005.

[164] Gilman Tolle, Joseph Polastre, Robert Szewczyk, Neil Turner, Kevin Tu, Stephen Burgess, David Gay, Phil Buonadonna, Wei Hong, Todd Dawson, and David Culler. A macroscope in the redwoods. In *SenSys*, November 2005.

[165] Ronald Tschal. HTTPClient Library. `http://www.innovation.ch/java/HTTPClient/`. Accessed on July, 2008.

[166] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12, New York, NY, USA, 1979. ACM.

[167] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007.

[168] Marc A. Viredaz and Deborah A. Wallach. Power evaluation of a handheld computer. *IEEE Micro*, 23(1):66–74, 2003.

[169] Paul Vixie. Extension Mechanisms for DSN (EDNS0). RFC 2671, August 1999.

[170] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM.

[171] Paul A. S. Ward. *A Scalable Partial-Order Data Structure for Distributed-System Observation*. PhD thesis, University of Waterloo, 2001.

[172] Brian Wellington. dnsjava library. `http://www.dnsjava.org/`. Accessed on July, 2008.

[173] Matt Welsh, David Culler, and Eric Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.

[174] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: a wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.

[175] Wikipedia Infrastructure, http://meta.wikimedia.org/wiki/Wikimedia_servers.

[176] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *INFOCOM'02: The 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, June 2002.

[177] Matei Zaharia, Andrew Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, San Diego, CA, December 2008.

[178] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[179] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[180] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 123–132, 2002.

[181] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, January 2004.

Appendix A

X-Trace Metadata Specification

X-Trace Specification Document: X-Trace Metadata Format

Authors: Rodrigo Fonseca, George Porter
Draft-Created: 18-Jul-2006
Last-Edited: 07-Apr-2008
Specification Version: 2.1.1
Revision: svn 53

> **Note**
>
> This version of this specification document is 2.1.1, and it specifies the X-Trace metadata version 1. The X-Trace metadata version is the version number that is encoded in the metadata itself, and is independent of the version (2.1.1) of this document.

Summary

This document describes the X-Trace metadata format version 1, and how it is encoded in both binary and textual form. It also describes the mandatory fields, and the format of optional fields. Separate documents describe individual options, including how they are represented, propagated, and how they relate to X-Trace reports.

1. Introduction

X-Trace [1] is a framework for tracing the execution of distributed systems. It provides a logging-like API for the programmer, allowing the recording of events during the execution. X-Trace records the causal relations among these events in a deterministic fashion, across threads, software layers, different machines, and potentially different network layers and administrative domains. X-Trace groups events (which we also call operations) in tasks, which are sets of causally related events with a definite start. Events in a task form a directed acyclic graph (DAG). Each task is given a unique task identifier, and each event within a task is given an identifier which is unique within the task.

X-Trace tracks the causal relation among events by propagating a constant-sized metadata along with the execution, both in messages exchanged by the different components and within the components themselves. The metadata contains the task identifier of the current task, and the last event recorded in the current execution sequence. Each event is reported to a separate reporting infrastructure, and each report records the identifier of the current event, as well as the identifier(s) of the event(s) that caused the current event. The reports contain other information in the form of key/value pairs. The format of these reports is described in the accompanying specification [2].

The X-Trace metadata has two mandatory components, and a set of optional ones, described in their own specifications. The first mandatory component is a **Task Identifier**, or TaskId, which uniquely identifies a task (in the context of a reporting infrastructure and during a finite window of time). The TaskId is the same across all operations comprising a task, and serves the purpose of identifying all such operations as belonging to the task. The second component is called the **Operation Identifier**, or OpID, and should be unique for each operation or event within the task. The OpID identifies the operation that generated the metadata, and is used by the subsequent operation(s) to record the causal relationship between the two operations by means of reporting.

There are also optional fields, which may or may not be present, and enhance the functionality of the X-Trace framework. At the time of this writing, there are three types of options defined, but more can be added in the future. They are the Destination option, for specifying where to send reports if not to a default location; the ChainId option, to explicitly indicate concurrent chains of events; and the Severity option, that indicates how important it is to record events related to a particular task.

This document specifies in detail the layout and meaning of the X-Trace Metadata how it is represented in binary and textual form, and how it is propagated in the same and between adjacent layers. It does not discuss reporting in detail, nor how to implement the propagation. It is also beyond the scope of this document how different protocols include X-Trace Metadata associated to their respective protocol data units.

2. X-Trace Metadata Binary Representation

The X-Trace Metadata (XTR-Md) is represented in a compact wire format by the following byte layout. This layout is used when the XTR-Md is encoded in binary protocols, as is the case of the encodings of XTR-Md in IP options and in TCP options. A hexadecimal textual representation of this binary layout is also used when XTR-Md is encoded in text-based protocols, as is the case of HTTP and SIP, for example.

Flags:	1 byte
TaskId:	4, 8, 12, or 20 bytes
OpId:	4 or 8 bytes
Options Block (optional):	
OptionsLen:	1 byte
Options:	OptionsLen bytes

The only mandatory fields are flags, TaskId, and OpId. The other fields are optional.

2.1 Flags

There are 8 bits (1 byte) allocated for flags. The bits are presented here with 7 being the most significant bit and 0 the least significant bit.

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Version				OpIdLen	Options	IdLen	

Bit	Name	Description
0-1	IdLen	Encodes the length of the TaskId
2	Options	Whether there are option fields present
3	OpIdLen	If 0, OpIdLen = 4, if 1, OpIdLen = 8
4-7	Version	Encodes the version of the X-Trace Metadata

Bits 0 and 1 encode the length of the TaskId as follows:

IdLen		Mask	Bytes for TaskId
1	**0**		

IdLen		Mask	Bytes for TaskId
1	**0**		
0	0	0x00	4
0	1	0x01	8
1	0	0x02	12
1	1	0x03	20

Bits 4 through 7 encode the version of the metadata as an integer, with 4 being the least significant bit. The value 15 (bits 4 through 7 set to 1) is reserved for future expansion of the version number, if necessary. **The current version id represented by this spec is 1**. It is backwards compatible with version 0. The difference between the two is that version 0 has bit 3 of the flags set to 0, and only allows OpIds of length 4.

Version				Mask	Version
7	**6**	**5**	**4**		
0	0	0	0	0x00	0
0	0	0	1	0x10	1
0	0	1	0	0x20	2
...					
1	1	1	1	0xF0	reserved

2.2 TaskId

The TaskId identifies a Task, or a higher level operation that generally corresponds to a user-initiated task. It can be composed of several operations or events that span different software components, nodes, and layers. As noted above, it can be 4, 8, 12, or 20 bytes long, with the length being selected by the IdLen bits in the flags field. The TaskId MUST be the same across all operations comprinsing a task, or else the operations will not be associated with the same task.

The set of TaskIds comprised of all 0's is reserved to mean **INVALID** TaskIds. An X-Trace Metadata with an **INVALID** TaskId is invalid, and MUST not be propagated or generate reports.

2.3 OpId

The OpId field identifies each operation or event that is part of a task and needs to be distinguished from the point of view of the X-Trace framework. It is a 4 or 8 bytes in length, depending on the setting of the flags bit 3. The value is interpreted as an opaque string of bytes, not as a number, and needs to be unique unique within the context of a task.

If the OpId length is 4 bytes, the version can be set to 0 or 1. The table below specifies how implementations of versions 0 and 1 of the X-Trace metadata specification treat the different settings of the OpId length field.

Version	OpIdLen	OpId	Version 0 Impl.	Version 1 Impl.
0	0	4 bytes	ok	ok

Version	OpIdLen	OpId	Version 0 Impl.	Version 1 Impl.
0	1	INVALID METADATA	fail	fail
1	0	4 bytes	fail	ok
1	1	8 bytes	fail	ok

If ChainsIds are being used as options to capture the concurrency structure of a task, then the OpId needs to be unique only within the context of a ChainId.

2.4 Metadata Length

Given these three mandatory fields, the smallest well-formed X-Trace metadata is 9 bytes long, comprising the flags field, a 4-byte TaskId, and a 4-byte OpId. As two examples, in hex representation, a well-formed and valid X-Trace metadata can be 00 01020304 03030303 (with spaces added between the fields for clarity). The smallest well-formed, invalid X-Trace metadata is 00 00000000 00000000. Note that if the OpId length is set to 4, the settings of version to 0 or 1 are both valid.

The maximum size is 1 + 20 + 8 + 1 + 255 = 285 bytes, but so far we have seen very little use of options, and no long options have been defined.

2.4 Optional / Extended fields

The option bit in the Flags field indicates the presence of optional or extension fields in the metadata.

2.4.1 Options Length

To make determining the size of the XTR-Md easier, there is a Length field that contains the length of all options combined, in bytes. This length **does not** include the length field itself. Thus, for determining the total length of an X-Trace metadata with options, one would add:

> 1 (flags) + (length of TaskId) + (length OpId) + 1 (OptionsLen byte) + OptionsLen.

If present, the options length field MUST NOT be set to 0. If there are no options, the O bit in the Flags field MUST be set to 0.

2.3.1 Option Format

Following the Options Length field, there may be one or more options. Options have a Type field, represented by one byte.

If the type is 0, it is a **no-option** type. The option of type 0 has a total length of 1 byte, and no body. Option type 0 MAY be used for padding, when it is more efficient or not possible to include arbitrary-length byte fields in protocols. It MUST only occur at the end of the options list. Implementations MUST NOT try to parse options past the first type 0 option.

No-option option format	
Type:	1 byte (0)

If the option type is not 0, i.e., between 1 and 255, then the option is a normal option, and follows the option

format below:

Normal option format	
Type:	1 byte (1-255)
Length:	1 byte (0-253)*
Payload:	Length bytes

The length only includes the size of the payload. Because of the total length of all options being limited to 255 bytes, the maximum length of each option can be at most 253 bytes (because of the type and length fields of the option itself. It there are more than one options, then the maximum length of each will be correspondingly smaller.

3. X-Trace Metadata Text Representation

For text-based protocols (ASCII, Unicode), like HTTP, SIP, or email, for example, X-Trace Metadata is represented as a the hexadecimal string of the successive bytes in the binary representation. The characters A to F SHOULD be represented in CAPITAL LETTERS, but implementations SHOULD be tolerant to non-capital letters. Each byte MUST be 0-padded and thus represented by two characters each.

4. Propagation

For the propagation of the X-Trace metadata, a node implementation will generate a unique OpId and replace the OpId of the previous operation(s) that happened before the current one. The TaskId MUST remain the same. So that the graph remains connected, every time a new OpId replaces a preceeding one, this binding MUST be registered in a report.

How specific options are propagated will depend on the semantics of each option. How each option is propagated is part of the option's specification. However, if an implementation doesn't know how to propagate a specific option, it MUST copy the option to any new metadata it generates based on the current one.

Figure 1 below shows an example of a simple HTTP transaction through a proxy that propagates X-Trace Metadata across layers and operations of the task. In the figure, (a) represents an OpId, and <T,a> represents metadata with TaskId T and OpId a.

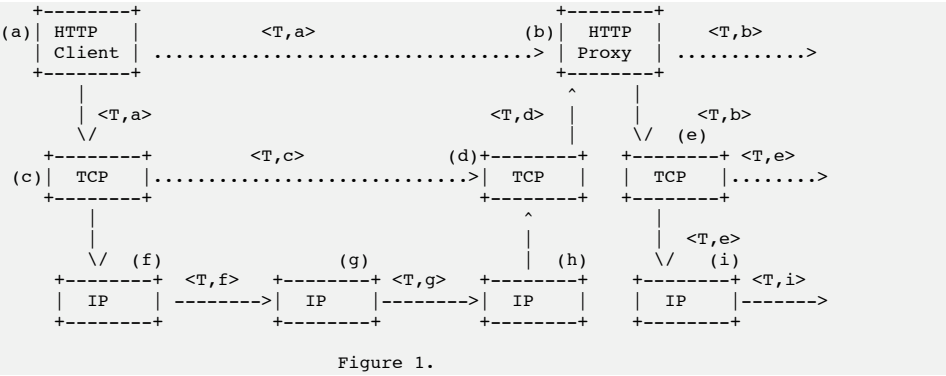


Figure 1.

5. Author's Address

Rodrigo Fonseca

465 Soda Hall
Berkeley, CA 94720-1776
USA

email - rfonseca at cs dot berkeley dot edu

George Porter
465 Soda Hall
Berkeley, CA 94720-1776
USA

email - gporter at cs dot berkeley dot edu

Citations

[1] Rodrigo Fonseca, George Manning Porter, Randy H. Katz, Scott Shenker and Ion Stoica, "X-Trace: A Pervasive Network Tracing Framework". In Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI'07). Cambridge, MA, USA, April 2007.

[2] X-Trace Specification Document - Report Format. http://www.x-trace.net

Appendix: Change Log

Changes marked with a '*' mean changes that have implementation implications. Otherwise changes just refer to the document (fixes and clarifications). The versioning reflects this: minor numbers will change with at least one '*' change, e.g., from 1.2.x to 1.3.0.

- 2.1.1 - minor changes and fixes
- 2.1.0 - * upgraded the Metadata version to 1, backwards compatible with version 0.
  This update introduces variable length OpId field (4 and 8 bytes).
- 2.0.0 - major revision of the X-Trace metadata format, simplifying the metadata contents and propagation.
- 1.3.1 - added change log. fixed section numbering
- 1.3.0 - ! added a length byte to the destination field
- 1.2.1 - fixed typo in the mask for the task id length of 20 bytes: was 0x0C, should be 0xC0
- 1.2.0
  - ! added invalid XTR id
  - updated the description example to have 4-byte tree info operation ids
  - Added sentence to cover propagation operations on metadata with no tree info.
  - fixed typo and clarified IdLen flags
  - added reference to NSDI paper