Improving Distributed Application Reliability with End-to-End Datapath Tracing

George Porter

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-68
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-68.html

May 22, 2008

Copyright © 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Improving Distributed Application Reliability with End-to-End Datapath Tracing

by

George Manning Porter

B.S. (University of Texas at Austin) 2001
M.S. (University of California, Berkeley) 2003

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 Randy H. Katz, Chair
Professor Scott Shenker
Professor Charles Stone

Spring 2008

The dissertation of George Manning Porter is approved:

Randy H Katz 14 May 2008

Chair Date

16 May 2008

Date

Charles Stone 20 May 2008

Date

University of California, Berkeley

Spring 2008

Improving Distributed Application Reliability with End-to-End Datapath Tracing

Copyright 2008

by

George Manning Porter

Abstract

Improving Distributed Application Reliability with End-to-End Datapath Tracing

by

George Manning Porter

Doctor of Philosophy in Computer Sciences

University of California, Berkeley

Professor Randy H. Katz, Chair

Modern Internet applications are both powerful and highly interactive, but creating and operating them requires new approaches to the software development, deployment, and debugging processes. Applications like Google Maps, Facebook, and Map/Reduce must run continuously, handle millions of concurrent requests, and scale by taking advantage of the massive amounts of parallelism available in large-scale Internet datacenters. When a failure occurs, discovering the exact set of machines and resources responsible, as well as the location among those resources where the failure occurred, is a daunting task. The lack of visibility into these distributed systems prevents their reliable operation.

To improve distributed system visibility, we have developed an integrated tracing framework called X-Trace. A user or operator invokes X-Trace when initiating an application task (e.g., a web request), by inserting X-Trace metadata with a task identifier in the resulting request. This metadata is then propagated down to lower layers through protocol

interfaces (which may need to be modified to carry X-Trace metadata), and also along all recursive requests that result from the original task (by modified software stacks). The X-Trace infrastructure makes use of this metadata to build a "task graph", which represents a trace of the execution of the distributed application. Using these recovered task graphs, we have been able to identify correctness and performance bugs in a wide variety of distributed applications, from web and overlay applications, to the Hadoop Map/Reduce system. In this work, we present the design and implementation of X-Trace, show its application to the 802.1X network authentication protocol, and then present an API and software tool for manipulating large-scale network traces in a scalable manner.

Professor Randy H. Katz
Dissertation Committee Chair

Dedicated to

Fr. J. B. Leininger, S.J.

who taught me how to learn.

Ad Majorem Dei Gloriam

Contents

List of Figures **vi**

List of Tables **x**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 New challenges . . . 6
1.3 Approach and summary of contributions . . . 7
1.4 Structure of the dissertation . . . 11

2 Background and Related Work **13**
2.1 Network measurement and traffic classification . . . 14
2.1.1 Network-wide measurements . . . 14
2.1.2 Traffic classification . . . 16
2.1.3 Feature extraction for security . . . 17
2.1.4 Traffic matrix estimation . . . 18
2.1.5 Peer-to-peer traffic detection . . . 19
2.1.6 Application-aware network devices . . . 19
2.1.7 Network Processors . . . 20
2.2 State Exchange . . . 21
2.2.1 Packet annotations . . . 21
2.2.2 Internet extensions utilizing packet annotations . . . 22
2.3 Tracing . . . 23
2.4 Middleware . . . 26
2.4.1 Middleware architectures . . . 26
2.4.2 Performance analysis of middleware systems . . . 27
2.5 Machine learning . . . 28
2.5.1 For network routing . . . 28
2.5.2 For application identification . . . 29
2.5.3 For service failure detection . . . 29
2.6 Protecting services under load . . . 30
2.6.1 Network quality-of-service (QoS) . . . 30

2.6.2 Classical control theory (CCT) . . . 31
2.6.3 Denial-of-service (DoS) protection . . . 32
2.6.4 Application failure detection . . . 33

3 X-Trace: A framework for collecting end-to-end datapath traces 34
3.1 Motivating scenario: Wikipedia . . . 35
3.2 Design . . . 40
3.2.1 Design principles . . . 41
3.2.2 System architecture . . . 42
3.2.3 Task graph . . . 44
3.2.4 X-Trace metadata . . . 45
3.2.5 Metadata propagation through network protocols . . . 45
3.2.6 Metadata propagation through applications . . . 47
3.2.7 X-Trace reports and report collection . . . 49
3.3 Implementation . . . 54
3.3.1 X-Trace metadata . . . 55
3.3.2 Protocol extensions . . . 55
3.3.3 Propagation libraries . . . 56
3.3.4 Reports . . . 59
3.3.5 Reporting libraries and agents . . . 59
3.3.6 Report collection . . . 60
3.3.7 Inter-AS reporting . . . 63
3.3.8 In-core graph reconstruction . . . 63
3.3.9 Performance . . . 64

4 X-Trace: Evaluation 66
4.1 Web request and recursive DNS queries . . . 66
4.1.1 Overview . . . 66
4.1.2 X-Trace support . . . 68
4.1.3 Fault isolation . . . 69
4.2 A web hosting site . . . 70
4.2.1 Overview . . . 70
4.2.2 Tracing a request through the scenario . . . 71
4.2.3 Using X-Trace . . . 72
4.3 An overlay network . . . 74
4.3.1 Tracing a message through the scenario . . . 75
4.3.2 Using X-Trace . . . 77
4.4 Additional X-Trace Uses . . . 79
4.4.1 Tunnels: IPv6 and VPNs . . . 79
4.4.2 ISP Connectivity Troubleshooting . . . 79
4.4.3 Link layer tracing . . . 80
4.4.4 Development . . . 80
4.5 Discussion . . . 81
4.5.1 Report loss . . . 81
4.5.2 Managing report traffic . . . 81

4.5.3 Partial deployment . . . 82
4.5.4 Security Considerations . . . 83
4.6 Conclusions . . . 84

5 Exposing network service failures with application-level datapath traces 86
5.1 Motivation . . . 87
5.2 Approach . . . 89
5.3 Case Study: IEEE 802.1X . . . 92
5.3.1 Improving 802.1X observability: Motivations and challenges . . . 92
5.3.2 802.1X operation . . . 94
5.4 Design . . . 98
5.4.1 Problem statement . . . 99
5.4.2 Approach . . . 99
5.4.3 Collecting application traces using X-Trace . . . 100
5.4.4 Fault occurrence detection . . . 100
5.4.5 Fault localization . . . 101
5.4.6 Individual root cause determination . . . 101
5.4.7 Reporting faults to the operator . . . 102
5.5 Implementation . . . 102
5.5.1 Integrating X-Trace into 802.1X . . . 102
5.6 Evaluation . . . 110
5.6.1 Evaluation plan . . . 112
5.6.2 Experimental setup . . . 112
5.6.3 Results . . . 113
5.6.4 Feasibility . . . 120
5.6.5 Limitations . . . 121
5.7 Conclusions . . . 122

6 TraceOn: Scalable trace graph traversal 123
6.1 Motivation . . . 123
6.2 Design . . . 127
6.2.1 Graph reconstruction procedure . . . 128
6.2.2 Programming model . . . 129
6.2.3 Operational model . . . 130
6.2.4 Graph reconstruction procedure . . . 131
6.3 TETRA: TraceOn External Trace Reconstruction Algorithm . . . 132
6.3.1 Dataflow overview . . . 133
6.3.2 Input processing . . . 135
6.3.3 Offset index construction . . . 136
6.3.4 Root extraction . . . 137
6.3.5 Child map creation and `listChildren()` . . . 137
6.3.6 First topological sort and Finish Table 1 . . . 138
6.3.7 Second topological sort and Finish Table 2 . . . 142
6.3.8 Database initialization and layout . . . 143
6.4 Implementation . . . 144

6.5 Demonstration Application: an NSF award search engine 146
6.6 Evaluation 150
6.6.1 Input file processing 151
6.6.2 TETRA external stack 152
6.6.3 Depth first search graph iteration 153
6.6.4 Database construction 154
6.6.5 Ad-hoc query evaluation 154
6.6.6 Comparison to in-core approaches 155
6.7 Discussion and future work 158
6.8 Conclusion 159

7 Future Work and Conclusions **161**
7.1 Future directions 162
7.2 Contributions of the dissertation 163

Bibliography **165**

A X-Trace Enterprise Edition user manual **184**

B X-Trace metadata specification version 2.0 **201**

List of Figures

1.1 The Manchester Small-Scale Experimental Machine (SSEM) or "Baby", University of Manchester, June 21, 1948. [Courtesy of The University of Manchester] 2
1.2 The "Kilburn Highest Factor Routine", which was the first program executed on the SSEM on June 21, 1948. [Courtesy of The University of Manchester] 4
1.3 Overhead view of Google's Oregon Datacenter. [Courtesy of the Google Earth Mapping Service] 5

3.1 The structure of the Wikipedia open-source encyclopedia. 36
3.2 A proxied HTTP request and the logical causal relations among network elements visited. 38
3.3 An overview of X-Trace, including the instrumented datapath, reporting infrastructure, and post-processing components. 43
3.4 An X-Trace Task Graph representing an HTTP client connecting to a web server through a web proxy. This task demonstrates the multi-layer capability of X-Trace, touching on the TCP and IP layers. Each vertex in the graph represents an event in the transaction, and each edge captures a causal connection between events. Every event shares the same task identifier (not shown), yet has its own unique operation id, shown as the letters 'a' through 'n'. The boxed item represents an X-Trace report issued by the proxy. . . . 44
3.5 Radius packet format extended with X-Trace metadata. 46
3.6 The two aspects of metadata propagation: across protocols, and through the application's runtime environment. 47
3.7 Propagation through a multi-threaded system. X-Trace metadata is stored in per-thread storage. 49
3.8 Multiple X-Trace enabled applications send reports to a per-node reporting daemon, or proxy. This proxy is responsible for storing the reports and forwarding them to the backend. 50
3.9 Services along the datapath forward reports to the backend server. This server hosts visualization software, as well as report query processing applications that developers and network operators use to identify failures. . . 51

3.10 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. 52
3.11 Support for adding metadata to some protocols. We have implementations for the protocols in *italics*. 57
3.12 X-Trace reporting proxy architecture. 61
3.13 An X-Trace backend designed for enterprise applications. Reports arrive via the message queue (pictured top-left), and configuration is centralized in the LDAP directory (pictured top-right). 62

4.1 The complete HTTP and recursive DNS graph recovered by the X-Trace tool 67
4.2 Architecture of the web hosting scenario 69
4.3 An HTTP request fault, annotated with user input 73
4.4 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. 74
4.5 (a)X-Trace graph 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. 76

5.1 Sample log entries from the Apache webserver. Each entry contains the client IP address, date and time, request string, and status information about the response, including the time taken to serve the content. 87
5.2 Message Sequence Chart (MSC) describing the structure of the 802.1X protocol. 95
5.3 The format of a modified RADIUS packet, which includes an X-Trace VSA. 96
5.4 X-Trace graph of a typical 802.1X authentication operation. The authenticator is represented by the "JRadius" nodes, the authentication server by the "OpenRadius" nodes, and the identity store by the "Sun LDAP" nodes. Each server appears twice: once on the forward path (the three nodes on the left side of the graph), and once on the reverse path (the three nodes on the right side). The solid lines represent the path the messages took, while the dotted line represents the delay at that component. 105
5.5 Root cause determination. 106
5.6 We analyze individual X-Trace graphs using this decision table. The top portion of the table defines an exhaustive set of conditions that occur in a given graph. The bottom portion of the graph encapsulates root cause diagnoses. As an example, given a graph G, we compute a vector of boolean values based on each of the conditions listed (e.g., "B present"). We find the column matching this vector, and identify the set of diagnoses based on the "X" values in that column. Note that a given set of conditions can lead to multiple diagnoses, in which case we notify the operator of multiple possible root causes. 111

5.7 Root cause test 1: Miscalibrated Timeout Values. At $t = 2.034$ the client times out, only to have the data it needed arrive at time $t = 3.012$. Note that the timestamps on different machines are out of sync with each other. In this work, all latency measurements are considered from the point of view of a single machine, and we never compare timestamps across different machines. 114
5.8 Root cause test 2: Authenticator-Side Packet Loss (Reverse path). Packets are lost on the reverse path, meaning that the entire datapath is transited, excepting for the final link between the RADIUS server and the authenticator. Since the authenticator does not receive this response, it retries, and eventually times out. 115
5.9 Root cause test 3: Detecting RADIUS server overload. Since the modified RADIUS server signals overload via X-Trace reports, we can deterministically detect overload by looking for "OVERLOAD_DROP" events. 116
5.10 Root cause test 5: Slow LDAP performance. This can easily be detected by examining the edge labeled '0.152s'. Note that this latency measurement is accurate, since the LDAP PREOP and POSTOP events occurred on the same machine, and so clock drift is not a problem. 118
5.11 Root cause test 6: Firewall Interference. In this figure the firewall was *not* instrumented with X-Trace, and so the firewall report was emulated. 119

6.1 The "real edge" iterator visits edges along the direct causal route from source to destination. In this HTTP example, the emphasized edges are part of the real edge path. 130
6.2 Dataflow description of the TETRA algorithm. 134
6.3 Field format for the offset index. Each record consists of an 8-byte operation id followed by a 4-byte length representing the offset into the original input file. 137
6.4 The NSF Award search application results screen. Users can search based on the P.I., the title, the award number, the award amount, and a free text search which matches any field in the award. The right column indicates the score, or how well the document matches the query. 147
6.5 The NSF Award grant display screen. This screen shows the entire NSF award. 148
6.6 An overview of the NSF award search engine, hosted in the Amazon EC2 virtual datacenter environment. The grant source data is stored in S3, and is accessed by a horizontally scaled set of Thrift-enabled server processes, called ThriftS3. The index servers (ThriftyCene) store their indices on local disks. Trying ThriftyS3 and ThriftyCene together is the NSF Search Servlet, a Java-based servlet that powers the user interface. 149
6.7 The TETRA import file processing phase performance. Extracting the root nodes, building the unsorted child map, and building the offset map into the original report file results in a processing efficiency approximately 36% below optimal (at 20,715 reports processed per second). To put that performance in context, the current X-Trace backend server can handle approximately 30,000 reports per second. 151

6.8 The TETRA external stack performance. Using Java's unmodified `RandomAccessFile` for the stack takes about 15 seconds to handle 100,000 `push()` operations, followed by 100,000 `pop()` operations. Adding a small amount of buffering reduces this time to about 12 seconds. Finally, adding a modest-sized, 16 KByte buffer reduces this considerably, to about 500 ms. Increasing the buffer size beyond 16 KBytes does not result in significant additional gain. . 152
6.9 TETRA graph reconstruction relies on two depth first search evaluations of the graph. By making use of the paging and buffering external stack data structure, DFS-1 and DFS-2 complete in 36.57, and 43.21 seconds, respectively. Their total memory footprint is under 100 MB., meaning that multiple DFS operations can be executed concurrently if multi-core environments are available. 153
6.10 Once the TETRA topological sorts are complete, the reports must be stored on-disk in real-edge visit order. This process took 43.68 seconds for the 1.3 million node dataset, resulting in an on-disk size of 281 MB. 154
6.11 The TETRA pre-processing step allows for rapid query evaluation against the optimized, on-disk trace layout. This query plots Amazon S3 `put` latency over time. Its source data, a 1.3 million event trace, is processed in just 36.08 seconds, using a small constant amount of memory. This enables the user to rapidly adjust and refine the offered queries based on observing the output of previous queries. 155
6.12 S3 put latency during the NSF Award database initialization. This figure was produced using the real-edge iterator programming interface exported by TraceOn. 156
6.13 S3 put latency during the NSF Award database initialization, shown as a cumulative distribution. 157

List of Tables

3.1 Performance of an X-Trace enabled Apache website with a Postgres-backed reporting infrastructure 64

Acknowledgments

I am grateful for all of the warm, generous, and profound support that so many people have given me during my education. Without my family, friends, and colleagues, I would certainly not be where I am today. And while I cannot list everyone who deserves my gratitude here, I want to acknowledge those whose support I have most counted on.

First, I would like to thank my research advisor and mentor, Randy H. Katz. I was incredibly lucky to share his support, encouragement, wisdom, experience–and humor! Scott Shenker has been nothing but supportive, opening doors to the research community that have been critical to the success of my research. Ion Stoica, while not on my committee, has been a great source of support and encouragement. I would also like to thank Charles Stone for his time and effort in serving as a committee member.

I have been fortunate to meet outstanding students while at Berkeley. I would especially like to acknowledge Rodrigo Fonseca, who made developing X-Trace exciting and fun. I would like to thank Saul Edwards, who worked with me on the 802.1X chapter and provided necessary moral support during graduate school. Andy Konwinski provided invaluable collaboration in designing TraceOn. I am deeply grateful for the students I've been lucky enough to work with as part of the Sahara, Oasis, and RAD Lab projects. I would also like to thank the staff of UC Berkeley, especially Mike Howard, Jon Kuroda, and Keith Sklower.

My work would not have been possible without the generous financial support of the State of Texas and the State of California. I would like to also thank the RAD Lab sponsors, who enabled my work on X-Trace.

In addition to my dissertation committee, several mentors have provided me with guidance, and have taught me what it means to be a scientist and an engineer. Christoph Schuba took me under his wing at Sun Microsystems, and supervised a very productive and rewarding internship. Minwen Ji hosted me at the Systems Research Center (first at Compaq, then at HP). My presence in graduate school itself would definitely not have been possible without the tireless dedication of J Strother Moore and Bob Boyer. I am grateful for their presence in my life. Before college, Karl Lehenbauer and Ellyn Jones gave me a chance to turn my love of technology, which was then a hobby and interest, into a lifelong passion. And in so many ways, I wish to thank Fr. J. B. Leininger, S.J. His dedication, passion, and love of teaching had a deep impact on me, and his quiet and relentless pursuit of learning shaped the lives of thousands of students, of which I am one.

I could never have reached this point without the love, humor, and care of my friends. From a fabulous send-off breakfast in Austin (thanks Joe), to long talks about science and teaching (thanks Hal), to lifesaving weekend trips to LA (thanks Todd and Helena), to timeless friends (thanks Josh). With special thanks I would also like to acknowledge Mr. and Mrs. Stufft, who have been very supportive of Monica and I during this process.

My family has been there from the beginning. Carolyn has kept me creative, made me laugh, and been there as a true friend. My parents, Misty and George, have served as an example of how to live, and their unceasing support and focus of education has made this accomplishment possible. Finally, I would like to thank my wife Monica. We met as students, and supported and encouraged each other. As we start our next journey, I couldn't imagine a better partner to go exploring with.

Chapter 1

Introduction

> *"The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair"*
>
> — Douglas Adams, Mostly Harmless, pg. 103.

1.1 Motivation

On June 21, 1947, the Manchester Small Scale Experimental Machine, or SSEM, was brought online (shown in Figure 1.1). Built at the University of Manchester by Tom Kilburn and F. C. Williams, its first task was to calculate the highest factor of a given number. Unlike the computers that came before it, such as the ENIAC[39], this task was not programmed into the SSEM using patch cables and rewiring its internal connections, which often took days to complete. Rather, the instructions needed to complete the task were stored in memory, just like data. The SSEM was the first stored program computer, and Killburn's Factor Routine was the first computer program.

The move from patch cables to stored programs meant that the computer could be

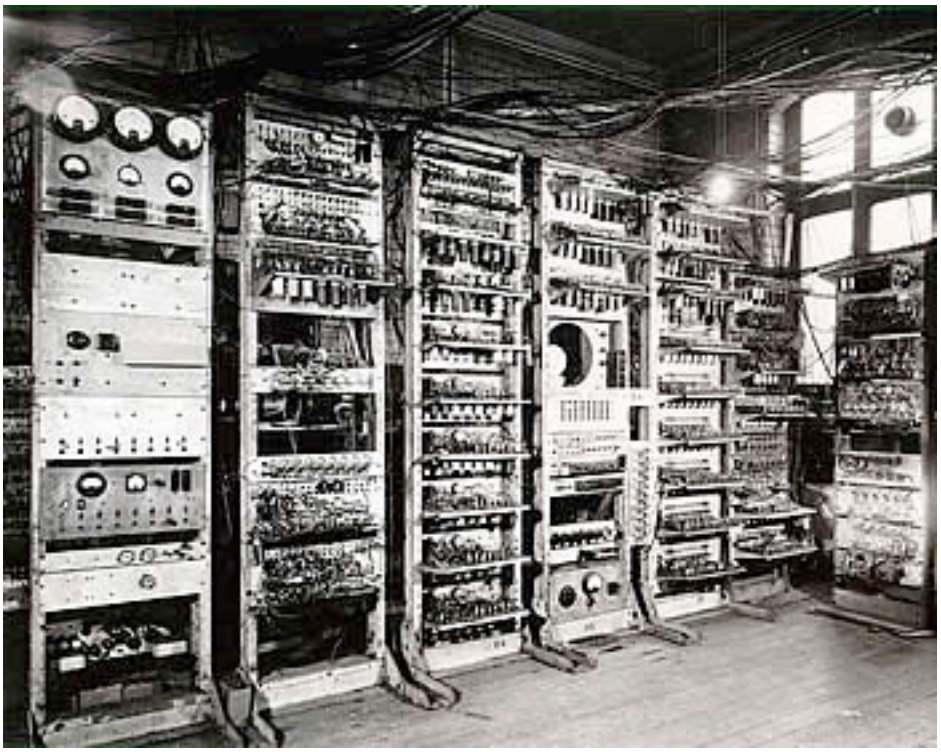


Figure 1.1: The Manchester Small-Scale Experimental Machine (SSEM) or “Baby”, University of Manchester, June 21, 1948. [Courtesy of The University of Manchester]

programmed directly from an attached keyboard, which allowed for rapid reconfiguration. It was the birth of software. Yet even this simple program, when it was first ran, resulted in unexpected behavior. F. C. Williams later remarked:

> "A program was laboriously inserted and the start switch pressed. Immediately the spots on the display tube entered a mad dance. In early trials it was a dance of death leading to no useful result, and what was even worse, without yielding any clue as to what was wrong[133]."

The SSEM could process seven instructions, had only 32 memory locations, and its entire state, including the program, all variables, and instruction pointer, was observable by examining an attached display tube. The program Williams and Kilburn wrote fit on a single page (shown in Figure 1.2). Yet the lack of visibility into the system's execution prevented Williams and Kilburn from correctly implementing their program, causing them to resort to trial and error until the program worked as expected. Their approach eventually worked, and was successfully able to calculate the highest factor of 2^{18} in just 52 minutes. But already, the lack of necessary tools to understand the software they were creating was an ominous sign for dependable and reliable software.

Over the next sixty years, computers grew powerful, taking advantage of generation after generation of innovations, from transistors to integrated circuits to multi-core processors. One particularly cost-effective way to build a powerful computer is to cluster many commodity computers together. This approach is embodied by Google's Oregon datacenter, shown in Figure 1.3. It spans multiple building full of thousands of racks, each outfitted with dozens of individual computers. Dissipating the enormous amount of heat generated by these racks are four-story tall water chilling plans. The entire facility is powered by its own power substation, which draws its energy from the Columbia River. This

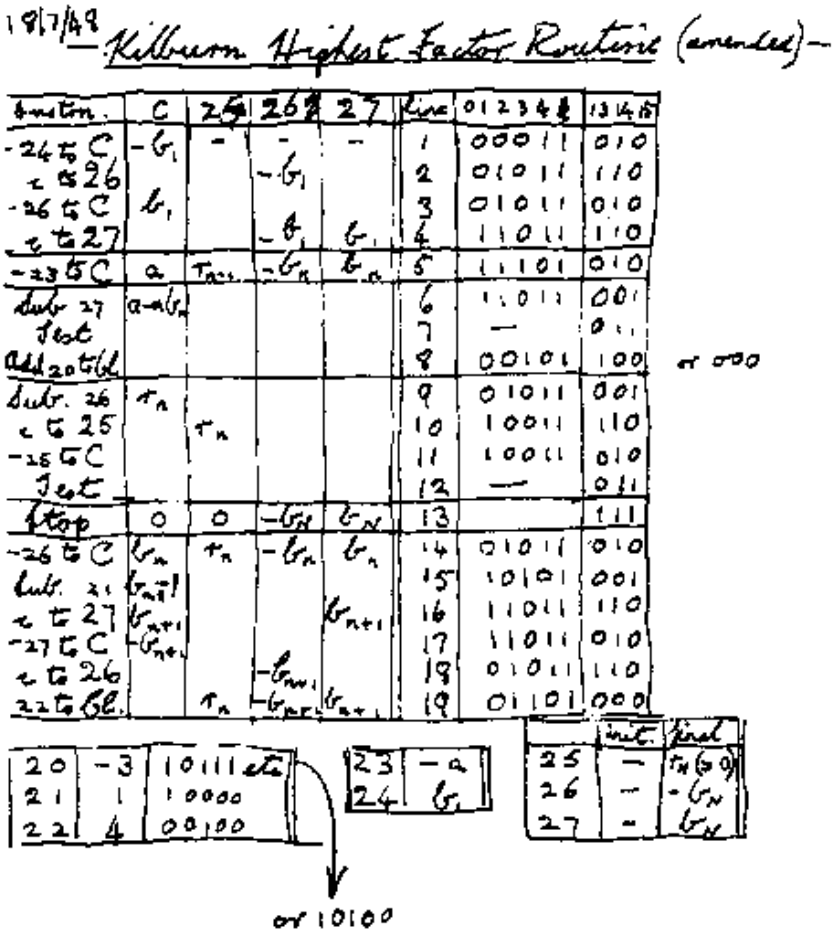


Figure 1.2: The "Kilburn Highest Factor Routine", which was the first program executed on the SSEM on June 21, 1948. [Courtesy of The University of Manchester]

dramatic demonstration of their size and scale provides for programs that are not simple computational problems like Kilburn's factor routine, but rather are globally spanning interactive applications like YouTube, Google Maps, and Google Search. Instead of 1024 bits of memory like in the SSEM, these applications operate over petabytes of data. Yet just as computers have become more powerful, that very complexity has exacerbated the lack of visibility. In order to continue this pace of innovation and expansion, we need better tools that provide visibility into complex software systems.

Figure 1.3: Overhead view of Google's Oregon Datacenter. [Courtesy of the Google Earth Mapping Service]

We now describe these new challenges to application visibility in detail. We then present X-Trace, which is our approach to addressing this problem. We continue by outlining the contributions of this work, and finally outline the structure of the rest of

dissertation.

1.2 New challenges

Modern Internet applications are characterized by:

1. **Unprecedented size and scale**. At Google, applications are assigned individual computational building blocks called "cells". Each cell consists of approximately 4,000 nodes, storage systems, and networking equipment[75].

2. **A "web services" composition model**, in which software components are loosely coupled together with open communications protocols. These couplings are established dynamically during runtime, complicating the process of capacity provisioning and planning.

3. **New failure modes**. The versatility of providing software "services" by loosely coupled components spread over the network results in unexpected behavior when failures occur. Because of the layers of abstraction the software service is built on top of, small failures at lower layers can disrupt the high-level service. For example, a misconfigured firewall can result in authentication failures for users connecting to the network. We explore this particular example in Chapter 4.

4. **Layers of virtualization**. This could include virtual disks located in storage area networks, or the entire datacenter could be virtualized. Several vendors, including Amazon and Sun Microsystems, are offering this "hardware as a service" on a pay-per-use basis.

The presence of applications like Google Search show that their implementation is feasible. However, the lack of pervasive, cross-cutting visibility into their operation results in bugs, reductions in performance, unacceptable variance in response time, and hindrance to growth. Without better tools to gain visibility into their systems, software developers and network operators will not be able to continue to grow their systems to meet the demands of their users.

1.3 Approach and summary of contributions

The goal of this work is to provide software developers and network operators with the tools they need to gain visibility into their software. The primary vehicle for this visibility is X-Trace, a cross-layer, multi-system tracing tool. X-Trace is a tracing methodology, programming convention, and software artifact. On top of the X-Trace foundation, we built TraceOn, a trace processing system that provides increased scalability for managing large-scale traces. We now describe this system, first by outlining our conceptual contributions, then by describing the software artifacts that arose from this project, and lastly by describing the real-world deployments in which we evaluated X-Trace.

We begin by outlining our conceptual contributions, which are:

1. **X-Trace methodology:** The foundation of our work is X-Trace. X-Trace is a set of software libraries and tools that developers make use of to collect end-to-end traces of the execution behavior of their systems. X-Trace works at a different layers, including the network, transport, middleware, and application. The X-Trace libraries modify protocol messages to include tracing information as well, meaning that traces can

span different devices.

2. **Use in diagnosing real-world faults:** During the development of X-Trace, we have embedded it into a variety of real-world applications, from a small web-based photo sharing site to an Enterprise-capable network authentication system. These deployments not only validated our design, but also provided valuable experience that we used to evolve our design to its current form. In fact, the design of X-Trace has undergone a variety of changes, each based on experienced gained deploying it in real-world systems. This process of iterative design has sharpened X-Trace's Application Programmer Interface (API), making the final product more powerful, while remaining easy to use.

3. **Approach to analyzing traces at large scales:** Because of the ease of X-Trace's adoption, it has been included in increasingly large distributed systems, such as Hadoop Map/Reduce. This has resulted in ever increasing trace datasets. We have had to scale our trace analysis tools to accommodate the increase in trace complexity, which has resulted in TraceOn, a large-scale trace processor.

We have realized these conceptual contributions in a variety of software artifacts, which include:

1. **X-Trace library:** The X-Trace library is linked into the software that is to be traced. Programs written in C, C++, and the Java languages are currently supported. As the software executes, trace data is generated and stored locally on each node.

2. **X-Trace backend (Research edition):** The purpose of the X-Trace backend is to

collect the trace data generated at each instrumented node. Additionally, the backend contains trace analysis tools for processing the traces after they are collected. The "Research edition" of the backend is deployed entirely as a stand-alone server, is written in Java, and is suitable for an individual or small group of researchers to get started quickly with X-Trace. The Research Edition does not require any support from the network administrators or operators.

3. **X-Trace backend (Enterprise edition):** Like the Research Edition, the X-Trace Enterprise Edition backend collects trace data and contains trace analysis tools. It differs from the Research Edition in that it is built around infrastructure services to perform its trace collection operation. While this requires support and setup by the network administrators, it is ideal for environments where X-Trace is a supported feature of the network, freeing software developers from managing the X-Trace backend. This allows a large number of software developers to share a centralized X-Trace infrastructure.

4. **TraceOn Server:** At its core, the X-Trace backend stores traces. The set of analysis tools included with it reconstruct those traces in memory, creating graphs of the execution that are presented to the user through a web interface. For very large traces, that in-memory reconstruction is no longer adequate, since the memory requirements quickly overcome most systems. TraceOn is a stand-alone server that interfaces to the X-Trace backend and includes an alternative analysis engine based on external data structures and pre-processing to avoid this memory requirement. As a result, large traces can be analyzed on standard server platforms, or even laptop computers.

We have demonstrated the effectiveness of our ideas coupled with their realization in software by deploying several real-world applications in both testbeds and datacenter environments. These deployments include:

1. **HTTP proxy:** This was the first evaluation of X-Trace, and proved the viability of encoding metadata in high-level protocols, in this case HTTP.

2. **2-tier web service:** This web application displays user-submitted photographs. Our deployment of X-Trace in this system evaluated the HTTP layer (through the Apache web server), as well as the database layer (through Postgres).

3. **I3/Chord overlay network:** Stressing X-Trace's cross-layer capability, the I3 and Chord instrumentation demonstrated its usefulness in a deeply multi-layered overlay network.

4. **802.1X network authentication protocol:** The 802.1X network authentication protocol is rapidly growing in importance as enterprise and campus networks need to better control access to their systems. This deployment demonstrated X-Trace's ability to bridge different protocols, written by different vendors, and controlled by different administrative domains.

5. **National Science Foundation grant award search engine:** X-Trace was designed to work in environments requiring a high degree of scale. In this deployment, we built a web search service hosted in a virtual datacenter, run by Amazon. The process of initializing this dataset resulted in a very large trace (with approximately 1.3 million events), providing a compelling evaluation of X-Trace's scaling properties.

Now that we have presented an overview of the X-Trace system, its conceptual contributions, software components, and our experience deploying it in real-world environments, we now present the structure of the rest of this dissertation.

1.4 Structure of the dissertation

In Chapter 2, we present the related work that this work is built on. The design of X-Trace cuts across network protocol design, active networking, quality of service, black-box analysis approaches, programming language design, and networking and operating systems research. In Chapter 3, we present the X-Trace framework. We begin by outlining the design decisions that led us to its current instantiation. We then describe its implementation, including modifications to software, protocols, and the construction of the backend. We then evaluate this implementation in the context of three deployed scenarios.

In Chapter 4, we present a use case of X-Trace in an Enterprise network environment, focusing on the 802.1X network authentication protocol. By analyzing the possible set of failures affecting 802.1X, we build a decision table that maps traces produced by X-Trace of unsuccessful authentications to a set of candidate root causes of failure. We then discuss how this approach of network failure detection from application layer traces can be applied to environments with limited network visibility, such as virtual datacenters and large enterprise organizations.

In Chapter 5, we present TraceOn, a trace analysis framework. TraceOn provides two primary benefits to users of X-Trace. First, it raises the level of abstraction at which developers write queries against X-Trace trace data, leading to easier-to-use interfaces for

the user. Second, it processes trace data in a more scalable way, making use of custom external data structures that operate efficiently by taking advantage of the semantics and structure of X-Trace graphs. We conclude the dissertation in Chapter 6. Lastly, we include two appendices of X-Trace specifications and user documentation.

Before we delve into X-Trace, we first present the work that inspired our design, and which we build upon.

Chapter 2

Background and Related Work

In this chapter, we present the background upon which this dissertation is built. We start our discussion with methods of collecting information about the network, and networked applications. These methods are aimed at capturing more holistic information than the fine-grained traces collected by X-Trace. We then describe work on facilitating state exchange. X-Trace relies on communicating state across network protocols, and relies on a variety of underlying state transportation mechanisms. We then compare X-Trace to related tracing projects. We further outline the growth of middleware systems, which are key building blocks to Internet-based applications. In the work of this dissertation, we make use of X-Trace instrumented middleware layers to inject trace instrumentation into applications without requiring extensive modifications by the developer. Next, we describe various uses for trace data, including its application to machine learning. Lastly, we describe related work on protecting services under load, and improving the user experience. In the next three chapters, we present X-Trace, a use case demonstrating its usefulness in enterprise-

capable applications, and finally an approach to improving the scalability of trace analysis techniques.

2.1 Network measurement and traffic classification

In this section, we describe related work in the area of network measurement and traffic classification. X-Trace tracks the execution of distributed applications as well as lower-layer network protocols. It could be used to identify and differentiate traffic flows, and as such, shares much in common with network measurement tools as well as traffic identification and classification systems. In this section, we describe these connections.

2.1.1 Network-wide measurements

A number of tools focus on monitoring network status, aggregating data from many devices and layers. X-Trace differs from these tools in that it traces, across devices and layers, the actual paths taken by data messages, rather than trying to get snapshots of the network infrastructure as a whole. Analyzing holistic information about the network is an integral part of running and troubleshooting networks. One such fundamental tool is `traceroute`, which traces IP network paths. A very common tool for capturing and displaying features of the traffic is TCP-dump[122]. This tool operates on endhosts, and its visibility is restricted to links that are adjacent to it.

Network operators must monitor hundreds or thousands of endhosts, routers, and network devices to discover and recover from faults. The distributed nature of network monitoring, and the large amount of data available poses certain challenges. One of the

most common protocols for extracting state from network devices is the Simple Network Management Protocol, SNMP[107, 108]. SNMP supports the remote examination of statistics collected in network devices. These statistics include traffic flow rates, exceptional event counters, and in some cases, protocol-specific counters. This information is stored in Management Information Base records, or MIBs[85]. MIBs vary from device to device, but typically include per-port and per-interface counters, records or traffic flow rates, counters of exceptional events, and other aggregate information.

The SNMP protocol is used to collect this MIB data. Commercial products that make use of SNMP include HP's OpenView[93]. Although SNMP can be used to instrument endhosts, it is less commonly used for that purpose. A tool designed to distribute systems-level metrics throughout a network for management purposes is Ganglia[50]. Ganglia-enabled hosts periodically announce systems metrics via XML messages on a shared, multicast channel. Commercial tools such as IPMonitor[63] can identify when network devices have failed or are unreachable. Cisco's NetFlow[90] system goes a step further and stores state on each flow transiting the router. Flow start time, packet size distributions, and flow rate data can be accessed via NetFlow. In practice, MIBs are queried remotely from management applications, while NetFlow data is transferred for analysis offline.

The need for exchanging state across the network was captured in the proposal of a Knowledge Plane (KP)[34]. The KP would serve as an information layer, orthogonal to traditional network layers. The KP would be distributed both geographically as well as administratively, and by querying it, network operators and end users would be able to pinpoint network faults and correct them.

2.1.2 Traffic classification

Routers forward traffic to its ultimate destination based on network-level addresses and fields encoded into the packet. This process is called packet classification. Typically in the Internet, routers extract a 32-bit IPv4 destination address, look it up in a table using a "longest prefix" matching rule, and finally the packet is sent out the appropriate network link. A variety of algorithms can be utilized, including linear search and binary search. Given a particular link speed, it is important to classify packets within a certain deadline time to maintain full link capacity. One such technique, proposed by Waldvogel, et al.[128], makes use of the Trie data structure, which allows binary search over a set of hash tables. As the authors show, an inexpensive hardware implementation of their scheme can forward hundreds of gigabits of data per second.

Traffic patterns and networked applications have become more sophisticated, and network operators often wish to specify complex forwarding policies for their networks. These policies specify forwarding behavior based on the origins of packets, the type of protocol the packet carries, and other features of the packet that require extracting more than just the destination address. This process is called "multi-field", or "multi-match" classification. An approach based on tuple space search[113] takes advantage of the fixed-length of most network fields. By reducing the space of match criteria to a smaller space of tuples, linear search for classification can be performed much faster. Gupta and McKeown define a multi-stage classification algorithm called Recursive Flow Classification (RFC)[56]. This algorithm consists of multiple phases of parallel memory lookups. The result is that nearly 30 million packets can be examined per second.

Cohen and Lund[72] analyze packet classification in the context of decision tree classifiers based on data taken from Tier-1 ISPs. Their proposal benefits from observed traffic characteristics, namely a Zipf-distribution of triggered rules based on typical rule-sets. There are commercial, multi-layer packet classification network devices available. Packeteer has a product called PacketSeeker[94] that can identify over 450 different protocols based on features extracted from all layers of the flows.

2.1.3 Feature extraction for security

The techniques described above typically assume that the fields used during classification are properly set. Given the rise in network attacks, automated worms, and viruses, network operators need to classify packets in which fields might intentionally contain false or misleading data. For example, an attacker might try to hide their identity by mis-setting the source address in packets they send. Firewalls are devices that test packets for conformity with network policies. Those packets not conforming are dropped. Probably the most common network firewall is CheckPoint[27]. Checkpoint specifies a set of rules and classification policies designed to prevent attackers from reaching critical network services. Cisco's PIX platform[33] has similar goals. In addition to router-based firewalls, endhosts often restrict network traffic based on classification rules. A common endhost-based system is Netfilter (IP Tables)[89]. The Snort system[109] consists of hundreds of rules that identify malicious traffic based on features from the network-level as well as the application-level.

In addition to attacking hosts, malicious attackers can make use of protocol ambiguities to gain information about the software running on endhosts. This information can be

used to tailor an attack. Protocol Scrubbing[130] is the act of sanitizing protocol exchanges such that ambiguities in the protocol are removed. Traffic that has been scrubbed thus gives no information to the attacker about the endhost–all traffic looks the same. Scrubbing involves extracting fields from traffic at a variety of levels and replacing those fields with semantically equivalent, but standardized, values.

2.1.4 Traffic matrix estimation

A common distributed network measurement that is of interest to network operators is the traffic matrix. A traffic matrix is a snapshot of the rate of flow on each of the major links in the network. Because of the high speed of traffic, as well as the distributed nature of the observation points, obtaining the traffic matrix is non-trivial. Direct measurements of each link necessary for exact results are considered too prohibitive to be feasible. A survey of existing techniques in estimating the traffic matrix is presented in [86]. The authors of [110] propose a model of origin-destination pairs based on observing disruptions to traffic caused by intentional link weight adjustments. Gunnar, et al.[55] compare the performance of a variety of traffic matrix estimation algorithms by utilizing data collected from a commercial ISP. In [96], the authors propose a distributed scheme that relies on the stability of traffic matrix observations that lets the observation points run periodically, rather than all the time. Lastly, in [142], the authors propose inferring the traffic matrix based on commonly available link-load data that is collected in each router.

2.1.5 Peer-to-peer traffic detection

Peer-to-peer (P2P) applications defy typical communications patterns in that endhost clients both source and sink data for other endhosts. Unlike client-server applications, which are amenable to firewall and proxy protection, client-to-client traffic is harder to detect, much less control. Servers tend to be centralized and easily protected, whereas clients tend to be distributed. Typically used for sharing content, many network operators prefer to restrict or eliminate P2P traffic. P2P software is designed to avoid restriction, and thus classification is non-trivial. A variety of P2P analysis tools have been proposed. Ennis, et al.[43] describe open-source approaches to detecting P2P traffic. The authors of [106] make use of application signatures, rather than network-level features such as TCP ports. Their approach was successful at detecting certain P2P applications. An alternative approach studied in [71] disregards the payload, and takes advantage of two heuristics based on the transport layer–the use of both TCP and UDP on the same port number, and the number of connected peers from a host.

P2P applications will become more sophisticated at hiding their identities, the deeper and more complex the classification will be necessary to detect them. For this task to succeed, deeper and more complex packet feature selection will also be needed.

2.1.6 Application-aware network devices

Based on the ability to perform multi-match classification, several network devices are now processing traffic based on features from all levels of the packet, including the application. One such device, the Nortel Alteon web switch[4], includes specialized

hardware that is able to process web traffic. It can apply load-balancing and encryption policies to web traffic using web URLs and cookies. This enables web service operators to give priority to certain type of requests, or to partition their servers to handle different content. Sun's N2000[121] switch operates in a manner similar to the Alteon. F5 Network's Big-IP blade switch[44] performs web load balancing between servers in the IBM BladeCenter platform. F5 has also published the programmatic interface[45] to its device to facilitate its integration with other network components. Cisco has a line of storage switches[32] that classify the network-based storage protocol iSCSI. In a similar manner to the web switches above, Cisco's product allows operators to impose policy on storage, rather than web, traffic.

2.1.7 Network Processors

An enabling technology to multi-match packet classification is the availability of high-speed network processors. The most widely available such processor is Intel's IXP line[64, 65]. This processor features multiple micro-computing engines that enable highly parallel processing of network traffic. Network processors have been used to build firewalls[124]. Software-based routers, which are endhosts that forward and optionally process network traffic, have been built around network processors[111]. Network processors have been used to accelerate the classification of application-level protocols in [87].

2.2 State Exchange

X-Trace transmits tracing metadata along the datapath so that distributed computations can be connected together into a single execution graph. This need for state exchange is not unique to X-Trace, and in this section, we describe other tools, and techniques, used to achieve this goal.

2.2.1 Packet annotations

Extensions to the Internet architecture have been proposed that add to, or enhance, typical packet forwarding. Often these enhancements require state about the connection, and that state is coupled with each individual packet. We call this state an annotation. Space for packet annotations is included in the Internet Protocol version 4[98]. IPv4 includes support for up to 40 bytes of "options". This option space is divided into one or more variable length entries each consisting of a type, and possibly length and data fields. IPv6[40] generalizes the use of options to include two distinct sets of headers: hop-by-hop headers and end-to-end headers. Hop-by-hop headers are intended for intermediate routers, and thus are the only ones that affect the core of the network.

Another way to bind annotations to a packet without modifying its contents is to tunnel the packet along with the annotation in a larger packet. The tunnel endpoint must decapsulate the packet, recover the annotation, and send the packet to the original destination. A common tunneling protocol is Generic Routing Encapsulation (GRE)[58]. A major drawback to tunneling is that the semantics of the protocol are violated and the destination address in the packet is changed. This means that tunneled packets may travel on

different network paths than untunneled packets. Often, TCP packets are encapsulated into UDP packets, which interferes with congestion control. Additionally, the communicating parties must include tunnel entry and exit points into their communication, which results in additional possible points of failure.

2.2.2 Internet extensions utilizing packet annotations

A variety of Internet extensions and enhancements have utilized packet annotations. XCP[74] addresses TCP instability and inefficiency that arises in networks with high bandwidth-delay products. It does this by applying control theory to provide both efficiency and fairness by taking more information into account than TCP does. XCP extends the information that each packet conveys by adding the sender's TCP congestion window and round-trip time estimate, as well as a third field that intermediate routers modify to affect the sender's congestion window to an XCP header stored in each packet. This data allows routers to regulate the flow of multiple senders using them. Quickstart[67] proposes that the sender annotate SYN packets it sends out with its desired sending rate. As routers along the network path see this SYN packet, they might adjust this rate downward if they are unable to support a new connection at the desired speed. When the packet reaches the destination, the initial rate field annotation determines the TCP parameters for the connection.

IP traceback is an enhancement of the standard network forwarding model that allows autonomous systems (ASes) to locate the source of packet floods during denial-of-service (DoS) attacks. Conceptually, routers along an Internet path keep enough state to reconstruct the set of links to a DoS source. To facilitate this, support at the network-level has been proposed in [104] to mark packets with identifiers that represent sampled network

path edge pairs. These edge pairs, which consist of two router IP addresses and a distance measurement, make their way to the victim network. With sufficient such edge pairs, the victim can reconstruct the correct ordering of links to the source of the attack. SCORE is a scalable network architecture that allows one to provide per-flow services such isolation and QoS without requiring routers to maintain per-flow state [115]. The main idea is to have packets carry this state, instead of routers maintaining it. By using this technique, called dynamic packet state (DPS), it is possible to provide a variety of services such as approximating per-flow fair queuing [119], guaranteed services [115], and per-flow load balancing [114].

2.3 Tracing

The most direct related work to X-Trace are other distributed trace tools. In this section, we describe those efforts.

Splunk [112] reverse-engineers end-to-end datapaths by mining application logs, allowing for certain path-oriented observation. While our approach for datapath tracing is more invasive, it is also more deterministic. The authors of [37] evaluate several algorithms that operate on end-to-end bandwidth measurements collected from active network traces to detect problems. Their analysis is especially focused on comparing different algorithms to analyze data with seasonal variations.

Hussain et al. [62] present a system for performing high-speed network traces at a large scale. The purpose of their work is to collect the data, process it according to anonymization policies, and make it available for multiple users. This work focuses on

traffic in the network, and not on capturing causal connections between requests at different layers. Kompella et al. [76] present a service for collecting "cross-layer information". The focus of this work is on collecting control path state at different layers. Using the information their system collects, one could identify how failures at one layer impact other layers. X-trace differs from this work in that we require widening the APIs at each layer, and focus on the datapath, rather than the control path.

The Application Response Measurement (ARM) [10] project annotates transactional protocols in corporate enterprises with identifiers. Devices in this 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. Aguilera et al., in [3], find anomalous behavior in distributed systems by treating each component as a black box, and inferring the operation paths by only looking at message traces. They present heuristics to recover the path given the timing relations among messages. A follow-up work, Pip [99] 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 AD, and does not capture cross-layer correlations. X-Trace is complementary to Pip in this sense. We believe that some of Pip's analysis can be performed on X-Trace's task trees.

Pinpoint [29] detects faults in large, distributed systems. The authors modified J2EE middleware to capture the paths that component-based Java systems took through

that middleware. They can mine collections of these paths to infer which components are responsible for causing faults. Our work focuses on recovering the task trees associated with multi-layer protocols, rather than the analysis of those recovered paths.

Magpie [13] is a toolchain that works with events generated by operating system, middleware, and application instrumentation, correlates them, and produces representations of paths through a system by inferring causal relations from a total ordering of events. Like X-Trace, they correlate lower level events with a higher level task, but their focus is mostly on a single system or in distributed systems that are tightly coupled and instrumented in a compatible way.

Causeway [25] offers support for propagating metadata in distributed applications in a mostly automated way. The focus is not on cross-layer tracing, and metadata is only encoded on the IP path. Each message can only have one metadata associated with it. In X-Trace, each layer may be carrying X-Trace metadata meaningful at that specific layer. Causeway could be used as a mechanism for transferring X-Trace metadata at the IP layer.

Liblog [51] is designed to find faults in a single distributed application. Each node of the application is modified to locally log all system calls. By aggregating these logs together, the entire system state can be reconstructed. Network messages are modified to include Lamport Clocks so that the message exchange can be synchronized. While it does not require changes in the application, it is not clear that higher level causal relations can be correctly inferred from the combined log.

2.4 Middleware

One "sweet spot" for introducing tracing into distributed systems is through instrumented middleware. Middleware systems bridge distributed pieces of code together, provide for an easy-to-use layer of abstraction that programmers can rely on, and allow legacy programs to interface to newer code. In this section, we describe these systems, and provide context for our work in adding X-Trace support to middleware systems.

2.4.1 Middleware architectures

Recent large Internet sites such as Amazon.com[5] and eBay.com[42] have adopted a three-tier approach to designing their sites. Three-tier systems are built out of middleware components. The tiers generally fall into web/presentation, business logic and application, and persistent storage and database. Three-tier systems allow for so-called "horizontal scaling", meaning that the system can grow through the replication of components in one of the three tiers. Examples of middleware layers include Sun's J2EE system[66] and BEA Logic's WebLogic[14]. Several open source web middleware systems have also been developed, such as JONAS[69] and JBoss[68] which typically couple Java-based middleware with the open-source web server Apache[9].

Our previous work in modeling web-services has been based on the Java-based RUBiS system[103], a publicly available workload generator patterned on eBay. The authors of RUBiS have analyzed the performance of it under a variety of conditions, and have showed that the mixture of requests–the workload–plays a large role in determining system bottlenecks[7, 23, 24]. Schmidt[105] shows that middleware-based systems have been

used for real-time and embedded applications, not just e-commerce and enterprise business. For interactive users of web services, Bhatti et al.[15] have shown that users' perception of web quality is non-linear. Namely, after a roughly eight second delay threshold, most users perceive that the system is non-operational.

2.4.2 Performance analysis of middleware systems

As middleware systems become more complex, the effects of requests given to them become difficult to statically analyze. Several efforts, both in the literature and in industrial best practices, have attempted to measure or predict the effect of individual requests on the various tiers of middleware systems. The SLIC project at HP Labs[35] attempts to identify which components are responsible for web service performance violations by fine-grained monitoring and instrumentation[141, 36]. They have shown that in practice, these SLO (Service level objective) violations can only be predicted by utilizing multiple systems metrics; it is not enough to find correlations between request features and a single metric such as CPU load.

The Performance Management project at IBM has explored using control theory and statistical monitoring to detect and adapt to unexpected traffic surges[82, 41]. In the Magpie project[13], request effects were uncovered by utilizing detailed instrumentation data taken from multiple layers of the system: device drivers, O/S calls, and middleware layers. Given such data, they build a causal model of the effect of requests given observations of the system.

2.5 Machine learning

The amount of trace data generated by X-Trace is sometimes very large. For long-running or complex applications, trace graphs with millions of nodes are not uncommon. In these cases, direct examination of the graphs is not feasible, and so more sophisticated techniques must be employed. In this section, we describe a relatively new branch of Statistics, Machine Learning, that can be brought to bear on analyzing trace data.

2.5.1 For network routing

Typically, network paths are chosen based on the metric of shortest path (in terms of the number of hops). An area of research called "Q-Routing" (similar to Q-Learning) uses reinforcement learning theory to choose network paths. The idea is that under high loads, static shortest path routing is suboptimal. The original paper on the subject by Boyan and Littman[19] proposes that each node maintain for each destination the estimate of packet latency via each neighbor. Given that this is load-sensitive routing, precautions have to be taken to ensure that oscillations do not occur. Boyan and Littman's work was extended[31, 79] to address the situation where network load decreases, and optimal paths must be chosen again. In the original Q-Routing work, longer paths were sometimes preferred over shorter paths since the Q-quantity that route selection is based on prefers stable, but slower paths verses more unstable, but sometimes faster ones. Chen and Druschel present a survey and case study of reinforcement learning in [28].

2.5.2 For application identification

In [88], Moore and Zuev address the problem of classifying traffic into application classes, such as Web, peer-to-peer, and FTP. The authors' approach is to utilize a naïve Bayesian classifier. That classifier was trained with manually labeled data. The features of the protocol that were relevant were header fields in the IP and TCP protocols, packet inter-arrival rate, payload size, and the effective bandwidth of the flow. After training, their Bayesian classifier was approximately 20% effective. They then applied a feature reduction technique called Fast Correlation-based Filtering (FCBF)[139]. This technique makes use of a quantity called the symmetric uncertainty to identify features that should be used in the model (the features chosen by the algorithm are not highly correlated with each other). The symmetric uncertainty between two variables is the quotient of the information gain and the product of the entropies of the two variables. After utilizing FCBF, their Bayesian classifier's accuracy jumped to over 90%. Lastly, they got rid of the Gaussian assumption of their model and instead made use of kernel functions. This increased accuracy a couple of percent.

2.5.3 For service failure detection

In [30], the authors study the interconnection of middleware components. By observing sufficient inter-middleware connections, they are able to apply statistical analysis to uncover errors by noting deviations from typical component communication patterns. Their approach relies on instrumenting the interfaces between Java components, which does not require a priori semantic knowledge of the application itself. Bodik et al.[18] make use

of machine learning to detect service failures in an e-commerce site. Their approach is based on detecting when page hit distributions deviate from what is expected. They detect this change via a χ^2 test, and couple that statistic with a visualization tool that operators can use to increase their confidence in the automated failure detection system.

2.6 Protecting services under load

While the bulk of this work studies the use of X-Trace to improve the reliability of individual software applications, it could also be used to protect the network as a whole. We first describe work on providing network quality of service, then describe other attempts to regulate and control network resources. The observation capabilities of X-Trace could be used to improve these techniques.

2.6.1 Network quality-of-service (QoS)

A considerable amount of research has investigated the problem of providing quality of service to network flows. Circuit-switched networks typically have a circuit-setup phase that reserves resources along the data path in advance (e.g., the telephone network). In packet-based data networks, this approach not directly applicable since each data packet is forwarded individually. Attempts at binding packets to certain paths have been made, including using label-switching protocols[101]. Two primary directions at providing network QoS include integrated services[20] and differentiated services[16]. Integrated services provide for fine-grained guarantees to be made to flows that are setup with the RSVP[21] protocol. Resources are reserved in advance, and each router on the path must

support integrated services. Contrasted with that is Diffserv, which provides for courser-grained prioritization of flows. In Diffserv, devices on the edges of the network aggregate possible many flows into a small number of categories. These categories are scheduled and buffered differently in the network, and the advantage of Diffserv is that the core of the network does not need to keep state on a per-flow granularity, since many flows are already aggregated into smaller Diffserv classes.

In SCORE, the amount of state that was previously needed in the core of the network is instead carried with the packets that require that state. In this way, routers in the network do not have to store state to provide QoS[115]. The ability for packets to carry QoS state is called Dynamic packet state[119, 114]. State is added in the edges of the network as packets enter the core, it is used in the core to provide fair queuing, guaranteed services, and per-flow load-balancing, and then it is removed before leaving the network.

2.6.2 Classical control theory (CCT)

Classical control theory has been used successfully in a variety of industries, including aviation, automobiles, manufacturing, and robotics. In CCT, a system is kept in equilibrium by adjusting some type of input signal to the system based on a model of how the system works. As the system moves out of equilibrium, the input signal is modified based on the system model. CCT has been applied to Internet systems as well. In [2], the authors apply CCT to a web server to provide for overload protection and service differentiation. Similarly, [41] focuses on differentiating service through the metric of constant factor differences in response time as seen by clients of the Apache web server.

CCT has also been applied to operating systems and three-tier systems. These ap-

proaches have tended to treat requests to the system as homogeneous (each request affecting the system in the same way). The SWIFT system[52, 53] is a systematic approach to introducing input/output interfaces to operating system components, which matches well with the well defined interfaces between middleware components. The ControlWare system[140] is a toolkit for automatically mapping QoS requirements into simple control loops in three-tier systems. The models these approaches have built as the central part of their controllers have not assumed, for example, the request differentiation work described in 2.4.2.

2.6.3 Denial-of-service (DoS) protection

DoS attacks are caused when an attacker, or group of attackers, issue a large number of requests to a service with the intent to deny access to legitimate users. DoS attacks are often easy to implement, since in the Internet, any host can usually send packets to any other host. When an attacker makes use of a large number of machines (often without the their owners' knowledge), then the attack becomes a distributed DoS attack (DDoS). DDoS attacks both increase the amount of attack traffic as well as hide the attacker's true identity through obscurity. Attempts at preventing such attacks have focused on a variety of avenues. In a technique called IP traceback[104], routers annotate packets with information about the paths those packets have taken. The victim can make use of that information to identify the source networks of the attack. The authors of SIFF[136] propose to add capabilities to packets which are granted by recipients. These capabilities are carried in each packet, and can be easily validated by core and edge routers. The advantage of the SIFF approach is that DDoS attacks can be prevented through the network–not just in the recipient's edge network.

Work on protecting web services and e-commerce sites is outlined in [73] and [22]. Router-mechanisms based on max-min fairness guarantees and other router-centric defenses are presented in [137] and [135]. An intriguing vulnerability is that of the "shrew" flows[80]. Typically, DoS attacks can be detected by looking for spikes, or surges, of traffic. Shrew flows make intelligent use of the dynamics of TCP to prevent applications from receiving adequate throughput. Because shrew flows do not themselves use a large amount of bandwidth on average, they are difficult to detect.

2.6.4 Application failure detection

The authors of [12] propose including monitoring components into a distributed web service. They outline two mechanisms: using the **Xlinkit** library, or linking directly into the web service. They look for service timeouts, "external errors", which are errors in the specific service instance, as well as violations of the BPEL spec.

Chapter 3

X-Trace: A framework for collecting end-to-end datapath traces

In this Chapter, we describe X-Trace, a cross-layer, multi-application trace tool. Software developers modify their software to include X-Trace instrumentation points by using a provided software library which implements the X-Trace interface. Once instrumented, X-Trace enabled software will generate trace data that is used to reconstruct the execution behavior after the fact. We begin this Chapter by motivating the types of applications that would benefit from X-Trace. We then describe its design and resulting implementation. In the next Chapter, we evaluate this implementation in a variety of real-world scenarios.

3.1 Motivating scenario: Wikipedia

Improving visibility into application behavior exposes otherwise difficult to detect bugs, underlying system failures, and generally unexpected application behavior. This should not be surprising, since software developers have come to rely on traditional software debuggers to debug their software. Applications are becoming more distributed and interconnected. This development leads to so-called "software-as-a-service" systems. The need to debug these compositions is critical to their reliability and made more complex because of their multi-system and dynamic nature.

Let us consider a concrete example of such a system: the Wikipedia on-line encyclopedia[132]. Imagine that a student in San Francisco updates the Wikipedia entry for their favorite baseball team. They tell their friend in Washington, D.C. about the change, but when their friend looks at the page, they see the old information. What conditions might lead to this failure? How can operators, developers, and users detect and correct this problem? In this Chapter, we introduce X-Trace, an end-to-end tracing framework designed to provide visibility into large-scale distributed systems such as Wikipedia.

As Figure 3.1 shows, Wikipedia is organized around a multi-tiered architecture of web load balancers, caches, application servers, and databases. The particular path through these components taken by one user might involve different components than another user. Since there is currently no way for the two users of Wikipedia to communicate the exact set of components they interacted with, it is not possible to narrow down the set of possible components that led to the observed failure highlighted above. One typical approach to addressing this problem is the use of per-component logs. Each component in the system

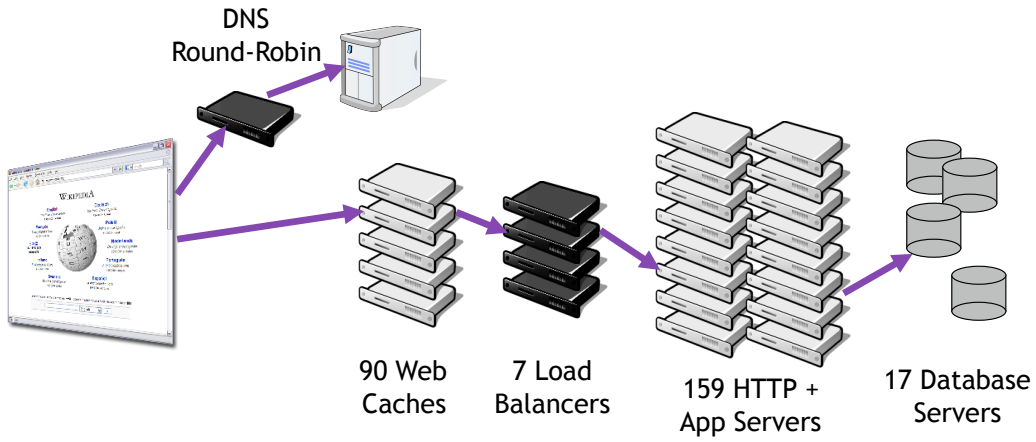


Figure 3.1: The structure of the Wikipedia open-source encyclopedia.

stores logs that are mined after the failure is detected. Given the huge scale that modern datacenters operate within, there is no common mechanism to determine which logs to examine out of the numerous possible logs deployed. There is also no direct way to correlate entries across multiple logs. It may not be possible for administrators to reproduce the problem, since their requests would most likely take a different path through the system. Thus, they will not be able to identify the defective component that led to the failure.

Various enterprise and datacenter diagnostic tools exist, but many of them are limited to a particular protocol. For instance, `traceroute` is useful for locating IP connectivity problems, but can not reveal proxy or DNS failures. Similarly, there are numerous alarm and monitoring suites for HTTP, but they cannot diagnose routing problems. While these tools are undoubtedly useful, they are also typically unable to diagnose subtle interactions between protocols or provide a comprehensive view of the system's behavior.

To this end, we have developed an integrated tracing framework called X-Trace. A user or operator invokes X-Trace when initiating an application task (e.g., a web request),

by inserting X-Trace metadata with a task identifier in the resulting request. This metadata is then propagated down to lower layers through protocol interfaces, which may need to be modified to carry X-Trace metadata, and also along all recursive requests that result from the original task. This is what makes X-Trace comprehensive; it tags all network operations resulting from a particular task with the same task identifier. We call the set of network operations connected with an initial task the resulting *task graph*.

Constructing the task graph requires understanding the causal paths in network protocols. While in general this may be difficult, in most of the situations we have considered so far this is relatively straightforward: for example, a recursive DNS query is clearly causally linked to the incoming request. X-Trace requires that network protocols be modified to propagate the X-Trace metadata into all actions causally related to the original task. This involves both understanding calls to lower-level protocols (e.g., HTTP invoking TCP) and initiating forwarded or recursive requests.

X-Trace-enabled devices log the relevant information connected with each tagged network operation, which can then be reported back. The trace information associated with a task graph gives the user or operator a comprehensive view of what network operations were executed as part of a task. This would include information about low-level IP packet transmissions, high-level protocol actions such as DNS queries, and actions from devices usually invisible to network troubleshooting, such as transparent caches as long as they are X-Trace enabled. To illustrate, Figure 3.2 shows an example of the task graph involved in a simple HTTP request through a proxy, showing the causal relations between operations in the HTTP, TCP, and IP layers. X-Trace task graphs are runtime traces of a task execution,

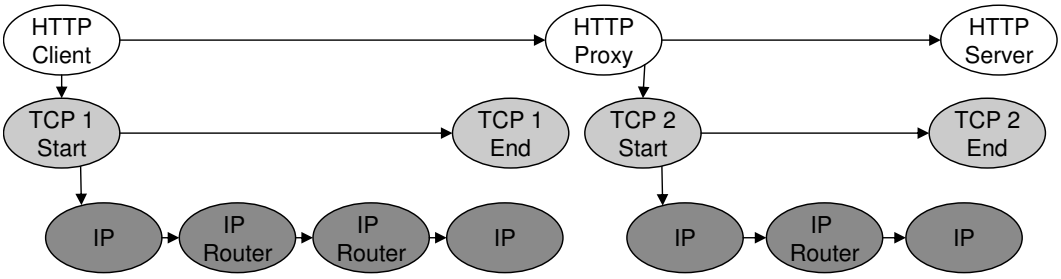


Figure 3.2: A proxied HTTP request and the logical causal relations among network elements visited.

and so long as individual components are integrated into the framework, there is no need for prior configuration of their dependencies. This is especially useful in systems that are large, dynamic, or administered by independent parties.

Diagnosing problems often requires tracing a task across different administrative domains, which we will call ADs. ADs may not wish to reveal internal information to each other, or to end users. Accordingly, X-Trace incorporates a clean separation between the client (user or operator) that invokes X-Trace, and the recipient of the trace information. For instance, when an end user notices a problem and invokes X-Trace, the trace information from her home network is delivered to her locally, the trace information from her ISP is delivered to the ISP support center, and the trace information from the web site she was accessing is sent to the web site operator. Each of these parties can then deal with the information as they see fit; sharing it with others, keeping it private, or even not collecting it at all. The fact that X-Trace gives them a common identifier for the task enables them to cooperate effectively if they so choose.

Realistically, we know all layers in the stack and different ADs will not deploy X-Trace-enabled protocols and devices simultaneously. However, individual protocols, ap-

plications, or ADs can benefit immediately from X-Trace if they support it. If a particular protocol or application gets instrumented alone, one gets horizontal slices of the task graph, which are useful for developers and users. If an AD alone deploys it on multiple layers within its network, it gets to internally visualize the portion of the task graph that happened inside of its domain. In addition, there is a "network effect" for adoption: as more protocols and networks integrate into the framework, X-Trace offers a common framework for their sharing of this information, increasing the value for all parties.

There has been much prior work on the study of application behavior, network monitoring, and request tracking. We discuss this related work in detail in Chapter 2, and only note here that the main differentiating aspect of X-Trace is its focus on tracing multiple applications, at different network layers, and across administrative boundaries. In the next chapter, we highlight these features in the context of three specific examples. However, X-Trace is applicable to a wide variety of other protocols, such as SIP, RPC, and email.

While we feel that X-Trace provides a valuable service, it certainly has significant limitations. They are discussed in detail at the end of this chapter, but we note them briefly here. First, implementing X-Trace requires modifications to clients, servers, and network devices; protocols that can not already do so must be altered to carry X-Trace metadata, and their implementations must log the relevant trace information. While these changes are conceptually simple, in practice retrofitting X-Trace into existing applications is a process of varying difficulty; our experiences in this regard ranged from trivial to quite challenging. Second, when X-Trace is only partially deployed, the ability to trace those parts of the network is impaired, sometimes entirely. Third, lost trace reports can limit reconstruction

of the request graph and can lead to false positives in diagnosing faults (i.e., the lack of trace data may be interpreted as a failure).

Because X-Trace only records paths that were taken, it is not a tool to assert global invariants about all possible paths. There are many problems for which X-Trace will not determine the cause, but will rather show the effect. While not an introspective debugger, it will point out the components involved in the operation, guiding the use of other tools to verify the cause. Examples of these cases are state corruptions that would cause a router to misroute packets, or an overloaded CPU that would cause a message to be dropped.

Despite these challenges, our experience with X-Trace has been very promising, shedding light on software correctness and performance bugs, as well as uncovering failures in networking and systems components. We now describe the overall design of X-Trace, followed by a detailed discussion of its implementation as an open-source software artifact. In the next chapter, we present three different real-world applications of X-Trace, highlighting its usefulness in these cases. We then close with a discussion of issues that arise during tracing.

3.2 Design

Now that we have motivated the need for pervasive and cross-layer tracing, we present X-Trace. We begin by outlining its design, and present an overview of its components. In the next section (Implementation), we will discuss how our X-Trace software artifact realizes our design, and how we are able to efficiently and reasonably add trace support to large-scale applications. But first, let us consider the principles that guided our

design.

3.2.1 Design principles

The purpose of X-Trace is to reconstruct a graph that captures key events making up a distributed system, and the causal connections between those events. In accomplishing this, our design follows these principles:

1. *The trace request should be sent in-band, rather than in a separate probe message.*

The first principle highlights our desire to probe what happens on the actual datapath we want to diagnose. Out-of-band probes might not end up following the same path as the original datapath. It follows that we need to *add metadata to the same datapath that we want to trace*. In X-Trace this metadata contains an identifier common to all operations in a task, which is added to messages and propagated by devices along the entire path.

2. *The collected trace results should be sent out-of-band, decoupled from the original datapath.*

This principle relates to gathering of trace information. If we appended trace information to the metadata encoded in the datapath, then we might lose this information in the event of network failure. Also, this would increase the overhead of messages. Obtaining trace data during periods of failure is especially critical to the debugging process. It follows that *we need an out-of-band, orthogonal mechanism to record and collect trace data*. Additionally, by decoupling trace reporting from the datapath, we lessen the impact of X-Trace on the datapath's latency, as well as provide more flexible options for collecting the generated data (e.g., sampling).

3. *The entity that requests tracing is decoupled from the entity that receives the trace results.*

As we discuss in the section on reporting, separating the user who inserts the X-Trace metadata in the datapath from the destination of the trace reports generated by components along the path allows for flexible disclosure policies of the trace information for each administrative domain (AD). Each AD keeps control of the information, while the common task identifier allows them to cooperate in exchanging information and solving problems if necessary.

X-Trace places the *minimal necessary mechanism* within the network, while still providing enough information to reconstruct the path. The data itself is not kept in the network path, but rather reported to specific places determined by the ADs. X-Trace metadata contains enough information for the ADs to communicate that trace information back to the user, if it so chooses.

3.2.2 System architecture

In this section, we describe architecture of X-Trace, which consists of the various supporting software needed to properly generate, collect, and process X-Trace traces. We based the design of X-Trace on the preceding principles, and the resulting architecture is shown in Figure 3.3. It is based on three conceptual pieces: an instrumented datapath, an offline reporting infrastructure, and a graph analysis engine.

The instrumented datapath is made up of X-Trace enabled hosts, servers, and network devices. These devices insert tracing metadata into protocol messages, and use

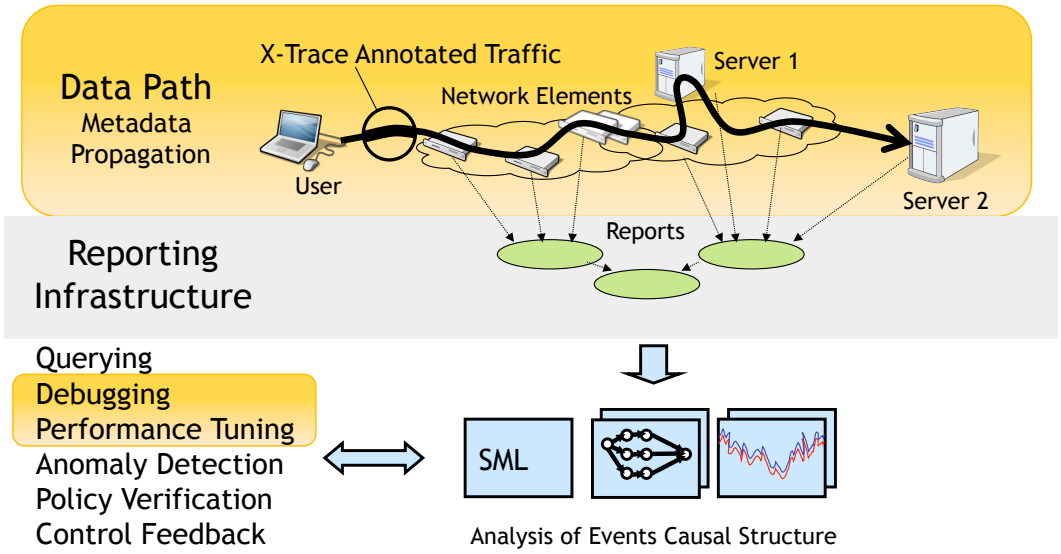


Figure 3.3: An overview of X-Trace, including the instrumented datapath, reporting infrastructure, and post-processing components.

that metadata to generate trace reports. A report is a text-based log entry that contains the X-Trace metadata, systems information such as the originating hostname and timestamp, as well as programmer-included annotations. The reporting infrastructure is a distributed collection system that periodically transfers reports from the hosts that generated them to a centralized backend server. Lastly, the backend server hosts the trace analysis component that developers, users, and operators use to identify failures, performance faults, anomalies, and other analytical actions on the trace data.

Before we describe the design of each of these components, we first present the concept of a task graph, which is the data structure that represents the execution of a distributed application. This is the data structure that we designed X-Trace to recover from distributed applications.

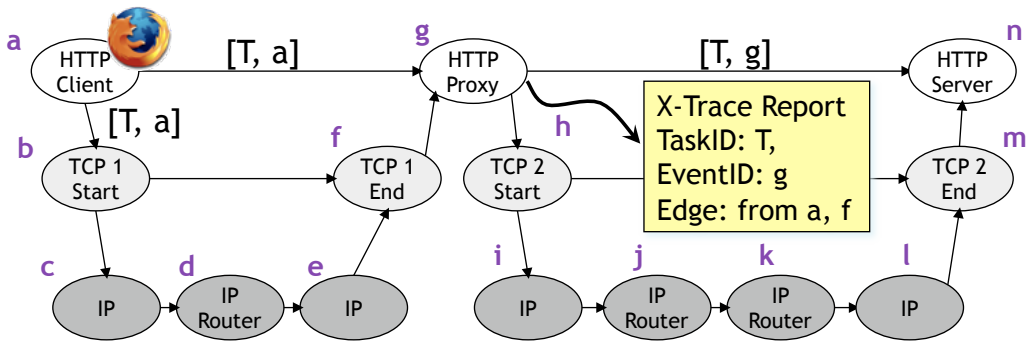


Figure 3.4: An X-Trace Task Graph representing an HTTP client connecting to a web server through a web proxy. This task demonstrates the multi-layer capability of X-Trace, touching on the TCP and IP layers. Each vertex in the graph represents an event in the transaction, and each edge captures a causal connection between events. Every event shares the same task identifier (not shown), yet has its own unique operation id, shown as the letters 'a' through 'n'. The boxed item represents an X-Trace report issued by the proxy.

3.2.3 Task graph

The key data structure in X-Trace is the task graph, and in this section, we describe its design. In X-Trace, a distributed system is made up of events defined by the software developer using the X-Trace API. A task graph is a graph data structure that records these events, as well as the causal connections between them. Figure 3.4 shows the task graph of an HTTP client issuing a request to a web server via a proxy. It demonstrates the multi-layer capability of X-Trace (touching on the TCP and IP layers). Each vertex in the graph represents an event in the transaction, and each edge captures a causal connection between events. Every event shares the same task identifier (not shown), yet has its own unique operation id, shown as the letters 'a' through 'n'. The boxed item represents an X-Trace report issued by the proxy. This report includes the task id, the operation id corresponding to that event, and a list of the operation ids that caused this event. In the figure, the report issued by the proxy was caused by the HTTP client that issued the HTTP request,

as well as by the underlying TCP layer that delivered the packets containing that request. Thus, the report indicates that events ‘a’ (the proxy) and ‘f’ (the TCP layer) caused event ‘g’ (the proxy).

3.2.4 X-Trace metadata

We now describe the design of the X-Trace metadata, which accompanies the execution of the application through the datapath. The metadata contains a task identifier, which must be unique among all of the reports accessed by an X-Trace user. In addition to the task identifier, the metadata also contains an *operation id*, also referred to as an *event id*. The operation id is unique within the context of a task id. Each event in the distributed system corresponds to an X-Trace report, and has its own unique operation id. Both the task and operation ids are unsigned integer types. Lastly, the metadata may optionally contain one or more options. An option is simply extended information that allows the metadata format to change and grow over time.

Tasks may be 4, 8, 12, or 20 bytes long, and operation ids may be either 4 or 8 bytes long. This means that within one enterprise, up to 2^{160} unique tasks may coexist, and that each task can consist of 2^{64} distinct events.

3.2.5 Metadata propagation through network protocols

Since X-Trace spans multiple applications, tracing metadata must be communicated between disparate software components. In this section, we describe the design of this capability. X-Trace metadata is carried by the extension, option, or annotation fields within each network protocol. Such fields include IP options, LDAP (Lightweight Directory Ac-

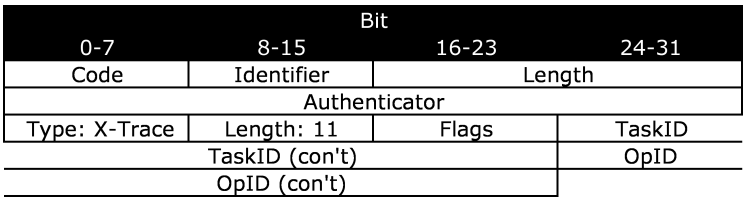


Figure 3.5: Radius packet format extended with X-Trace metadata.

cess Protocol) controls, and HTTP headers. Any protocol that allows for extension fields can work with X-Trace. The metadata is replicated across layers, ensuring that devices on the path can access it without having to violate layering. An example of X-Trace metadata included with an HTTP request is:

```
GET / HTTP/1.1
Host: www.cs.berkeley.edu
User-Agent: Mozilla/5.0
X-Trace: 01AABBCCDD01020304
Connection: close
```

In textual protocols, X-Trace metadata is encoded as a hexadecimal string. Here it is included as the "X-Trace" HTTP request header. A suitably X-Trace enabled web server would include the (modified) metadata along with the response, as shown here:

```
HTTP/1.1 200 OK
Server: IBM_HTTP_Server
Content-Type: text/html; charset=ISO-8859-1
Content-Language: en-US
Cache-Control: no-cache
Expires: Tue, 25 Mar 2008 20:49:54 GMT
Date: Tue, 25 Mar 2008 20:49:54 GMT
Content-Length: 18985
Connection: close
X-Trace: 01AABBCCDD3F78CAA0
```

Lastly, X-Trace metadata may also be included in binary protocols as well, as shown in the RADIUS (Remote Authentication Dial In User Service) protocol header (Fig-

ure 3.5).

3.2.6 Metadata propagation through applications

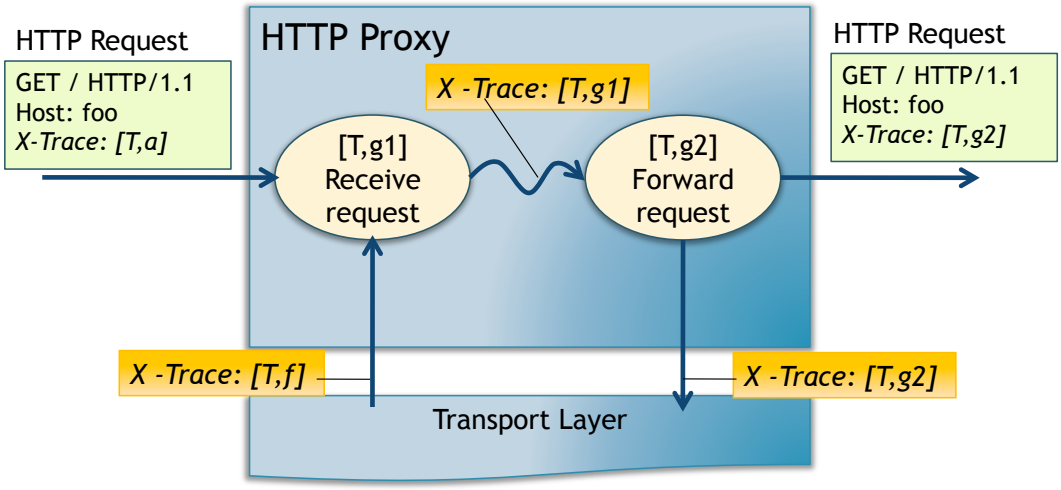


Figure 3.6: The two aspects of metadata propagation: across protocols, and through the application's runtime environment.

Devices and network elements on the path are responsible for propagating the X-Trace metadata along the path. Propagation ensures that X-Trace metadata stays with the datapath, as well as captures the causal relations between events in the application. Figure 3.4 shows in full detail the contents and the propagation of X-Trace metadata in part of the task graph from the HTTP example figure in the introduction to this chapter (Figure 3.2). In particular, the successive values of the task id and operation id fields must be passed along the datapath to recover the graph for this *TaskID*.

Figure 3.6 depicts the two primary requirements of application propagation: propagating through the software runtime, and propagating across the input/output of the program. Metadata is initially inserted into the network by the client. For legacy clients,

external devices in the network can add them. Network operators can also insert X-Trace metadata for operations traversing their AD. In the HTTP task graph (Figure 3.4), all of the vertical arrows represent cross-layer propagation.

As shown in the figure, the HTTP request pictured left contains metadata with Task ID 'T' and operation id 'a'. The first step to propagation is copying this data from the lower layer (TCP) and the adjacent layer (HTTP) into the application's address space. A new event is generated (labeled "Receive request"), and assigned a unique identifier, 'g1'. This identifier must be kept by the software runtime until the next event is generated, in this case "Forward request". This is so that the report corresponding to the "Forward request" event can indicate that it was caused by the preceding event. On the output side, the HTTP proxy has to copy the metadata into the newly generated transport connection, as well as the new HTTP request to the origin server. Likewise, the TCP process in the proxy has to copy this metadata down to the new IP path. Note that we do not make any *a priori* assumptions as to the number or ordering of layers in a protocol exchange: propagation works recursively, with each layer only naturally interacting with the layer immediately below. Metadata is placed into each layer of the application to support tracing, so that network devices do not have to access part of the packet outside of their layer. Since the X-Trace metadata is embedded into the messages at each layer, propagation happens at the same time as the messages are sent. In particular, if messages are stored for later forwarding, as is the case with email messages [38], the causal relations will still be preserved and recorded properly.

Multi-threaded applications are handled automatically by the X-Trace API. Con-

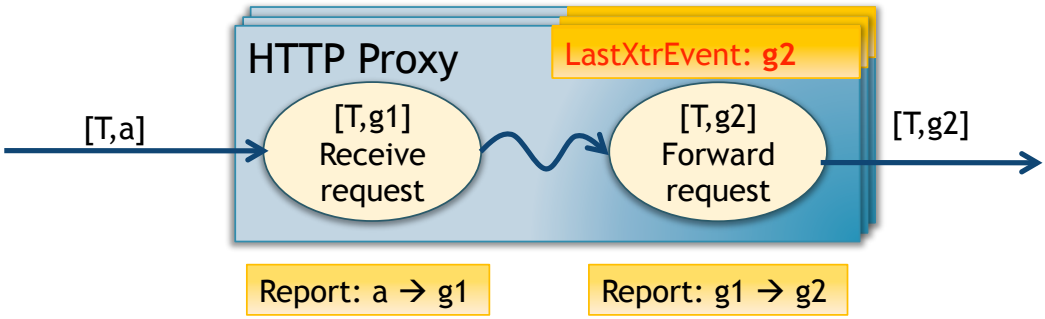


Figure 3.7: Propagation through a multi-threaded system. X-Trace metadata is stored in per-thread storage.

sider the HTTP proxy shown in Figure 3.7. In this case, the "last" operation id must be maintained on a per-thread basis. This is naturally performed by using per-thread variables in the case of the Java language.

3.2.7 X-Trace reports and report collection

X-Trace reports are the data structure that capture X-Trace trace data. In this section, we describe how these reports are generated, what information they contain, and how these reports are collected to a logically centralized place. When a node sees X-Trace metadata in a message at its particular layer, it generates a report, which is later used to reconstruct the datapath. This report generation operation is separate from propagating X-Trace metadata, and is specific to the graph reconstruction aspect of our application. Reports contain a local timestamp, the *TaskID* they refer to, the report's unique operation id, and any operation ids that caused this report to be generated, as well as programmer-inserted annotations. Data is stored in reports in the form of key-value pairs. Devices only report information accessible at their own network layer. For example, an HTTP cache may report on the URI and cookie of the request, and the action taken upon receiving the request.

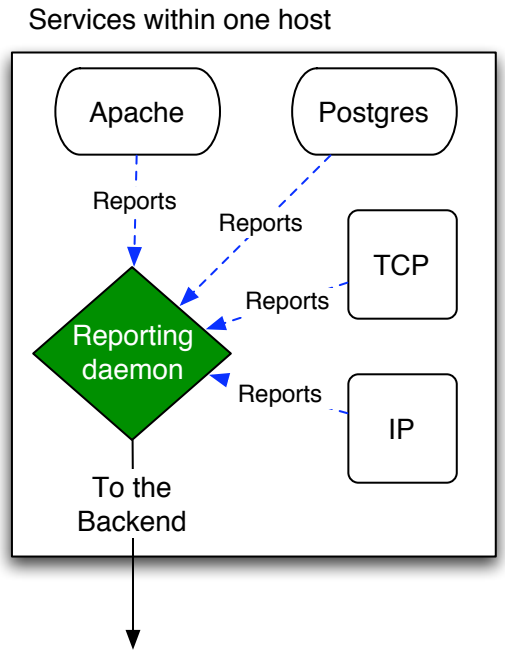


Figure 3.8: Multiple X-Trace enabled applications send reports to a per-node reporting daemon, or proxy. This proxy is responsible for storing the reports and forwarding them to the backend.

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.

The reports generated by devices within one AD are kept under the control of that AD, in local per-host storage. If multiple X-Trace enabled applications reside on the same host, they issue their reports to a local reporting daemon, which handles their storage and eventual transmission elsewhere in the network, as shown in Figure 3.8.

Periodically, reports must be transmitted to a central processing server that will complete the graph reconstruction process. This centralized approach ensures that reports generated in different parts of the network are visible from the analysis component of X-

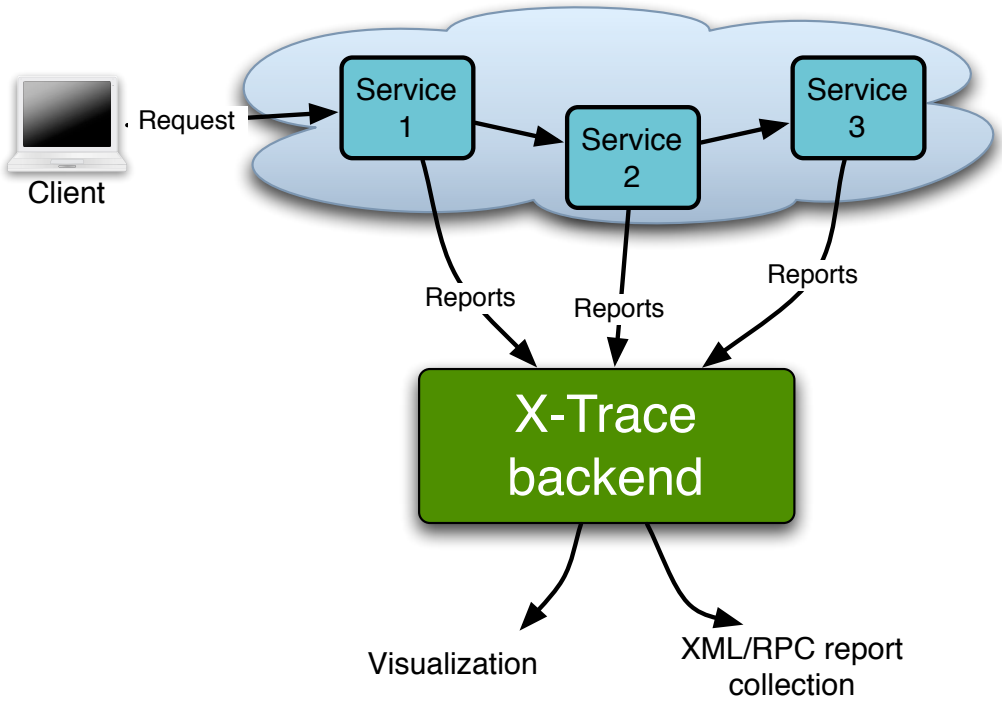


Figure 3.9: Services along the datapath forward reports to the backend server. This server hosts visualization software, as well as report query processing applications that developers and network operators use to identify failures.

Trace. The collection of reports can be organized as a *push*, *pull*, or a variation of *push* and *pull*. In the *push* model, each node in the system asynchronously sends reports to the backend server when they are generated. This minimizes the latency between the time trace data is generated and when it can be analyzed (see Figure 3.9). In the *pull* model, the trace data is buffered indefinitely on each of the instrumented nodes. Periodically, the backend will poll each node to collect the stored trace data. This increases the time until reports can be analyzed, but provides for finer-grained control of network overhead and prevents the backend from becoming overloaded. Lastly, a variation of the *push* and *pull* models would consist of periodic polling, with the ability for instrumented nodes to asynchronously transmit reports (e.g., in the event that the system is about to fail or leave the network). The

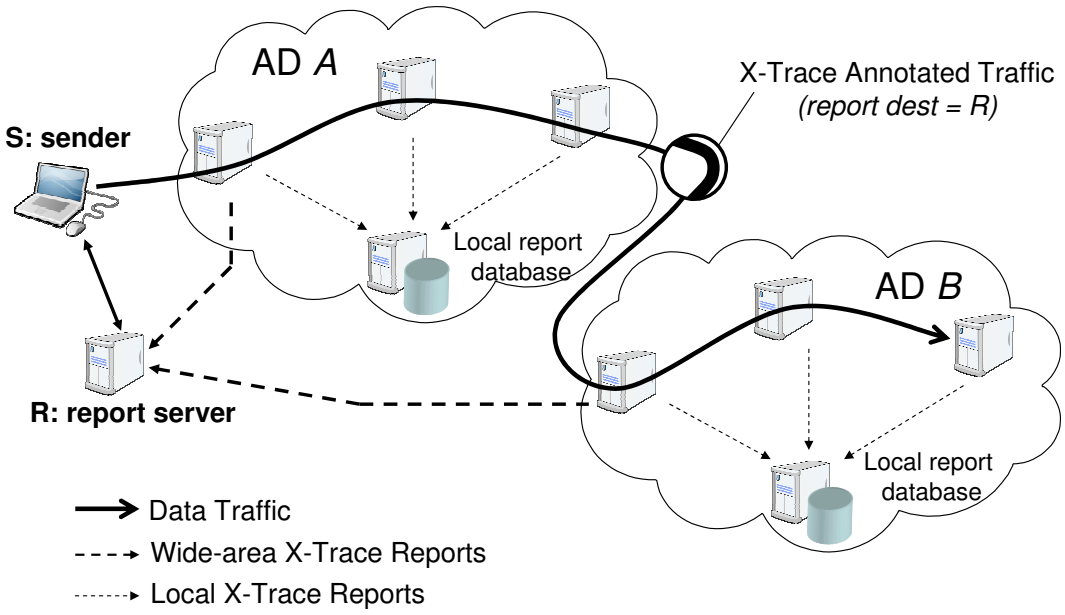


Figure 3.10: An example of wide-area reporting. The client embeds X-Trace metadata with a message, setting the report destination to R. Different ISPs collect reports locally, and send pointers to R so that the client can later request the detailed reports.

current X-Trace software artifact is based on the *push* model.

Deploying the reporting infrastructure required by X-Trace requires a level of expertise and on-going support that not all can provide. We imagine the emergence of an ecosystem of reporting service providers. These providers would maintain a reporting infrastructure, and could even provide visualization and diagnosis capabilities. ISPs could provide this service as a benefit for their users.

The X-Trace metadata has an optional *Destination* field. If present, this field signals that a user located at that destination is interested in receiving the trace data as well. This user might be the client, 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 the data. One possible mechanism uses indirection, and works as follows. The AD still collects all reports locally in a private database. It then sends a special report to the user, containing a pointer to the report data. The pointer could be the URL of a page containing the trace data. This 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. 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 3.10 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.

Graph Reconstruction

In this section, we describe the process of X-Trace graph reconstruction, which is the process of combining multiple distributed X-Trace reports together into a data structure that captures the execution behavior of a distributed application. Task graph reconstruction

is an offline process performed by the user that reconstructs the request path of the data connection. After the user collects reports from the reporting infrastructure, they examine them to reconstitute the datapath graph. Each of the reports is represented by a node in the task graph, identified by its unique operation id. A subset of the nodes, possibly empty, will have "edge" headers that represent causal edges in the graph. Each of these edge headers is represented by an edge in the graph. After reconstructing the graph, the client can examine the nodes and paths that the request took. For transitory errors, this 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.

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.

3.3 Implementation

Now that we have presented the design of X-Trace, we discuss its implementation. We first discuss the representation of the X-Trace metadata, then how we extended protocols to support X-Trace, the propagation libraries and API, the format of reports, the reporting libraries and collection infrastructure, how we carry out inter-AD reporting, and finally how graphs are rebuilt from individual reports.

3.3.1 X-Trace metadata

The complete specification of the X-Trace metadata is presented in Appendix B, however we now present a brief overview of its format.

Flags: The flags field contains bits that specify the metadata version number, whether any options are present, the size of the operation id field (either 4 or 8 bytes), and the length of the task id field (4, 8, 12, or 20 bytes).

TaskID: Our design supports 4, 8, 12, or 20 byte unsigned integer fields to represent the *TaskID*. The *TaskID* must be unique within 1) a window of time, and 2) a reporting domain. The window of time must be long enough so that no two tasks that overlap in time share the same ID.

Operation ID: The operation id field is a unique unsigned integer number within the context of a given task, and can be either 4 or 8 bytes long. We implement the `unique()` function as a random number generator.

Options: *(Optional)* To accommodate future extensions to the X-Trace identifier format, we include an options mechanism. The Options block, if present, consists of one or more individual options. Each consists of a type, a length, and then a variable length payload.

3.3.2 Protocol extensions

To support X-Trace, a layer or application must embed X-Trace metadata in the messages it exchanges with peers. The difficulty of this for existing protocols depends on

their specification. For example, it is simple for HTTP, because its specification [46] allows for extension headers, and dictates that unknown extensions be forwarded unmodified to next hops by proxies. Other protocols like SIP [102], e-mail [38], IP, TCP, and I3 [117] share this characteristic. For protocols without an extension mechanism, one has to resort to either changing the protocol or overloading some existing functionality. In the implementation of Chord that comes with I3 we had to create a new type of message. Table 3.11 gives details on adding metadata to these and some other protocols.

3.3.3 Propagation libraries

Applications must support two aspects of X-Trace identifier propagation: (a) carrying X-Trace metadata between incoming and outgoing messages, and (b) manipulating the metadata by updating operation ids to correctly record the causal relations. We implemented support in C/C++ and Java for easily manipulating X-Trace metadata, including performing the propagation operations, such that few lines of code need to be added to perform (b), once (a) is in place.

In our experience, we found that capturing the causal connections within the application presented the highest variability in difficulty, as it requires understanding how received messages relate to outgoing messages, and may require following long chains of calls within the implementation. If the implementation associates a context data structure with the processing of a message, it may be easy to add X-Trace metadata to the data type, which gets carried with the processing flow automatically. Apache and I3 fall into this category. In applications where a thread or process only handles one message at a time, it is possible to add the X-Trace metadata as a thread or process global variable. In event-based

Protocol	Metadata	Comment
HTTP, SIP, Email	Extension Header	Out-of-the box support for propagation. The only change is for causal relations.
IP	IP Option	Automatic propagation. Dropped by some ASs, wide-area support varies [52].
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	Mirrored augmented call path for new X-Trace data message.
DNS	EDNS0 OPT-RR	The EDNS0 [129] extension to DNS allows metadata to be added to messages.
SQL	SQL Comment	Possible to encode X-Trace metatada within a SQL comment.
Hadoop	RPC modification	Minor changes of the argument marshalling layer were needed to propagate metadata.
Facebook Thrift	RPC modification	X-Trace metadata stored in hidden RPC parameters.
libasync	RPC modification	X-Trace metadata stored in hidden RPC parameters.
UDP, Ethernet	No support	Must change protocol or use shim layer.

Figure 3.11: Support for adding metadata to some protocols. We have implementations for the protocols in *italics*.

systems, like libasync, it is necessary to associate the metadata with each callback. In more general cases, when messages for processing go through queues or shared memory, it is necessary to associate metadata with these structures as well. Other implementation structures require more work, as in the case of Chord: we had to create a parallel path of functions with an extra X-Trace metadata parameter following the call path from receiving the message until sending it. Instrumenting concurrency libraries and runtime environments may ease or automate this propagation [29, 99, 25]. We have already added X-Trace support for libasync [83], Facebook Thrift [123], and Hadoop [57].

The following program snippet shows a typical example of the calls that are needed for full identifier propagation in the forwarding function of an application. In this example, trace metadata is injected into the datapath with the *startTrace()* function. Each *logEvent()* function adds a new event to the datapath, propagating the metadata through the application using per-thread state.

```
public static void main(String[] args) {
     XTraceContext.startTrace("Document_Loader", "phase 1");
     ...
     XTraceContext.logEvent("S3", "store/start");
     S3.store(document);
     XTraceContext.logEvent("S3", "store/end");
}
```

The following is an example of the "Process" abstraction of the X-Trace propagation API. With it, developers specify the beginning of a semantic process using the *startProcess()* function. Successful completion of the operation is signaled via the *endProcess()* function. In the event of an exceptional condition, the Java exception raised is sent to the reporting infrastructure using *failProcess()*.

```
Private void storeDocument(String doc, String id) {
     XTraceProcess p =
          XTraceContext.startProcess("S3", "store/start");
     try {
          AmazonS3.put(id, doc);
          XTraceContext.endProcess(p);
     } catch (Exception e) {
          XTraceContext.failProcess(p, e);
     }
}
```

3.3.4 Reports

A report is a UTF-8 [138] encoded message consisting of a header section followed by a body section. The first line of the header identifies the layer issuing the report. The rest of the headers are specified as key-value pairs, according to RFC 822 [38]. The body of the report is free-form, and the content is set by the device issuing the report and other operator policy.

3.3.5 Reporting libraries and agents

Included with X-Trace is a reference implementation of a client library that can be linked into applications for issuing reports. This library is very thin, and simply relays reports to a locally running proxy.

The reporting proxy listens for reports on three different channels: a UDP socket, a TCP socket, and Facebook's Thrift RPC interface. One thread per channel listens for these reports, and places them on a shared queue. Another thread pulls reports off this queue, and sends them to the appropriate handler module. These handler modules, which run in separate threads, can forward the report to another report server, in the case of the reporting

proxy. Alternatively, in the case of the X-Trace backend server, the handler forwards reports to a report store, which commits them to local storage on the server. Metadata about available reports, tasks, and other bookkeeping information is kept in a local, embedded database.

We also implemented a packet sniffing application that can send reports on behalf of services and applications that cannot be modified to include the X-Trace library. This application snoops network traffic using the `libpcap` library, sending reports for any protocols that it supports. Currently, this application supports the IP and TCP protocols. Network switches can make use of port mirroring to mirror traffic to this agent.

The remaining task is to enable network elements – devices, protocol stacks, and applications – to issue reports of interesting events. For hardware devices like routers and appliances, one needs to modify the software running on the control processor. However, using the feature of port mirroring in switches, a network administrator can insert nodes that would report on traffic seen without slowing down the data path. The routers would still have to do the propagation, but not bother to call reporting functions. For software implementations, it is straightforward to integrate the reporting library, which is a variation of adding a logging subsystem to the application.

3.3.6 Report collection

An X-Trace proxy runs on each host in the system, and it is responsible for collecting reports from local applications. These reports are then forwarded to the backend, which stores them and makes them available for queries. In the current software artifact, there are two distinct implementations of this general design: one targeted for well-managed

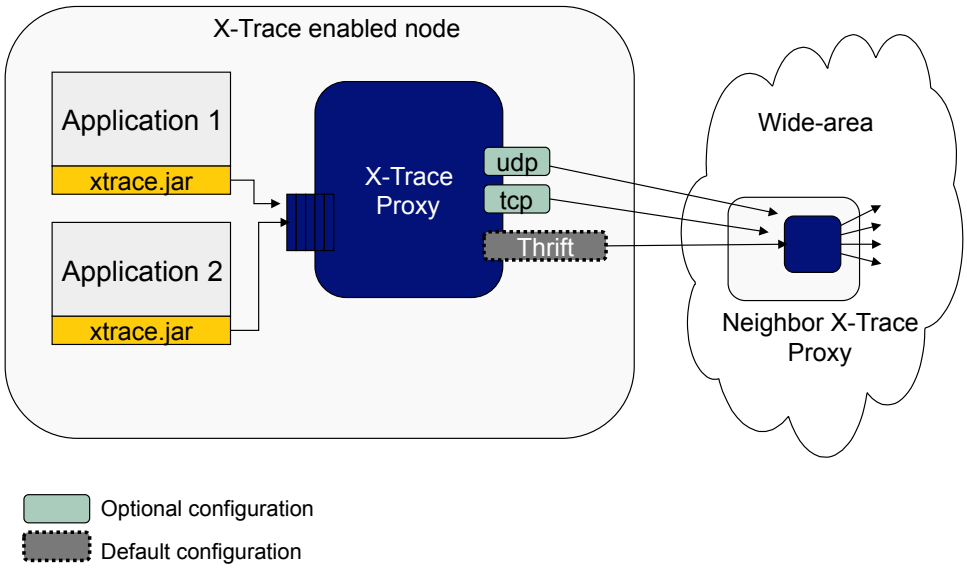


Figure 3.12: X-Trace reporting proxy architecture.

enterprise environments, and another targeted for "stand-alone" operation suitable for an individual user of X-Trace. In the former case, reports are collected and processed by central software infrastructure maintained by the network administrator. This leads to a highly scalable design requiring minimal setup by the user. In the second case, the user is responsible for running all of the components of X-Trace on their own server. However, they can setup and use X-Trace without administrator involvement.

Figure 3.13 shows the design of the enterprise-enabled reporting infrastructure. Sun's Message Queue, a Java Message Service provider, is responsible for communicating reports from the proxies to the backend. Commands can also be sent from the backend to the proxy with the message queue. Once reports arrive to the backend, they are committed to a Postgres SQL database. To visualize or query for these reports, users interact with an Apache-based web interface, which uses SQL to pull the reports from the database.

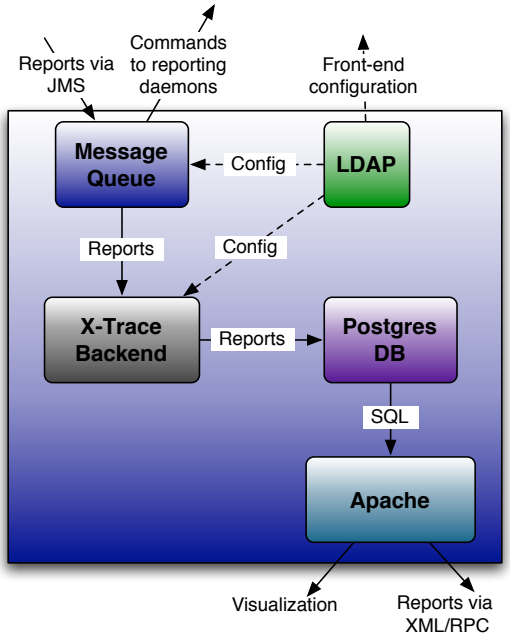


Figure 3.13: An X-Trace backend designed for enterprise applications. Reports arrive via the message queue (pictured top-left), and configuration is centralized in the LDAP directory (pictured top-right).

There are various parameters and configuration state that the clients must have to send reports to the backend. Rather than storing this state in the clients, it is stored in the backend, and made available to the clients via an LDAP interface. Thus, the clients must simply be programmed with the address of the backend, and upon initialization, they query the LDAP component to get their configuration state. This reduces the amount of state on end-hosts, and centralizes administration of the system to the backend server.

In contrast, the "stand-alone" backend is a single Java program that receives, stores, and serves out X-Trace reports. The visualization and query interface is provided by a Jetty embedded webserver. Rather than using a Postgres database to store reports, the stand-alone server stores the reports on disk in standard text files.

3.3.7 Inter-AS reporting

We have implemented a special case of Inter-AS reporting in the web hosting scenario described in the next chapter. The front end webservers included two HTTP headers in the response sent back to the client. The first contains a URL for collecting trace information about the request. The second is the X-Trace task identifier associated with the network operation. This is included to simplify handling at the client, as well as for environments in which the X-Trace metadata was added by the front-end webservers. We wrote a Firefox[47] extension that reads these HTTP headers, and provides the user with a visual indicator that the page they are visiting is "X-Trace enabled", as well as a button they can click to fetch the trace data from the provided URL, in the case of an error. This button is identified with the text "I'm feeling unlucky."

3.3.8 In-core graph reconstruction

Our initial implementation of the task graph reconstruction is quite simple, and can serve as the foundation for other, more complex, visualizations. We initially start by processing each report in the trace, adding child pointers to each node. Recall that X-Trace reports indicate causal edges in them, which are equivalent to parent pointers. To build the graph, we have to use these parents pointers to also generate the appropriate child pointers. Once the parent and child pointers are initialized, we output each node in the graph along with its child and parent pointers in the "dot" file format used by GraphViz. We then visualize the graph using a GraphViz viewer tool.

	Requests / sec	Mean latency
Baseline	764	1.309 ms
With X-Trace	647	1.544 ms

Table 3.1: Performance of an X-Trace enabled Apache website with a Postgres-backed reporting infrastructure

3.3.9 Performance

We tested the performance of the metadata propagation and the reporting aspects of our reference implementation of X-Trace. For the propagation, we measured the latency report generation. This operation is blocking, and if implemented in a router, would have to be performed on a per-packet basis on the forwarding path.

Our X-trace implementation generation was able to generate approximately 33,000 reports per second. Given that contemporary high-performance webservers handle about 10,000 requests per second, we feel confident that for most applications, reporting will not be the bottleneck. In fact, optimized libraries could be much faster. As shown in Chapter 6, we are also gaining experience with large-scale graphs consisting of millions of events.

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. Table 3.1 outlines these results, which show that reporting decreased the total system throughput by 15%.

In this chapter, we have described the design and implementation of X-Trace. In

the next chapter, we will present three evaluations of X-Trace, including deployments and micro-benchmarks. Then, we will discuss several issues that arise when integrating X-Trace into new and existing protocols and applications.

Chapter 4

X-Trace: Evaluation

In this chapter, we describe several scenarios where X-Trace could be used to help identify faults. We discuss three examples in detail–a simple web request and accompanying recursive DNS queries, a web hosting site, and an overlay network. We deployed these examples within one AD, and thus do not make use of the wide-area reporting mechanism. We follow these examples with a description of other scenarios.

4.1 Web request and recursive DNS queries

4.1.1 Overview

The first scenario that we consider is that of requesting a web page from a server. Figure 4.1 shows the graph corresponding to a simple web request. The user starts 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

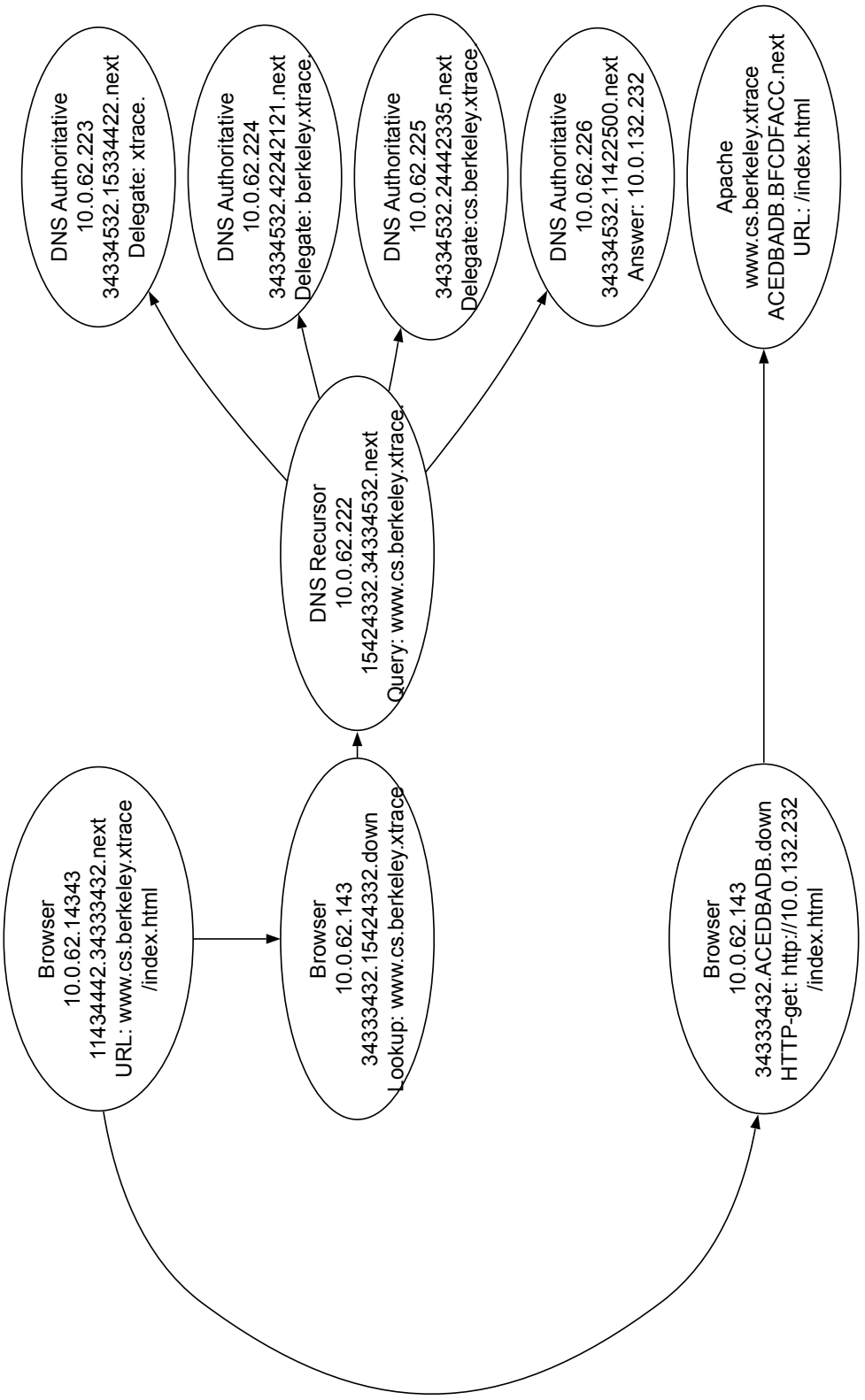


Figure 4.1: The complete HTTP and recursive DNS graph recovered by the X-Trace tool

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.

Tracing each of these "subtasks" is a challenge: 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. This can easily lead to misconfigurations and inconsistent views.

4.1.2 X-Trace support

We added support for X-Trace to the DNS protocol by using the EDNS0 [125] extension mechanism. This backwards-compatible mechanism allows metadata to be associated with DNS messages, and is increasingly supported in the wide area. We modified a DNS client library, an authoritative DNS server, as well as a recursive DNS resolver to support X-Trace metadata propagation and reporting.

We deployed this software in our local testbed, and created a parallel top-level domain (.xtrace). Figure 4.1 shows the final graph. In this example, the task has two subtasks, resolving the name, and fetching the page. A Java-based web browser issues the query to the DNS client library, which encapsulates the X-Trace metadata in an EDNS0 field of the query. This query is forwarded to the resolver on 10.0.62.222, which recursively looks up the address in other, authoritative nameservers, propagating the metadata at each step. Lastly, each of our authoritative nameservers issues reports when they receive queries with X-Trace/EDNS0 records in them. When the name resolution is complete, the browser issues an X-Trace enabled HTTP query.

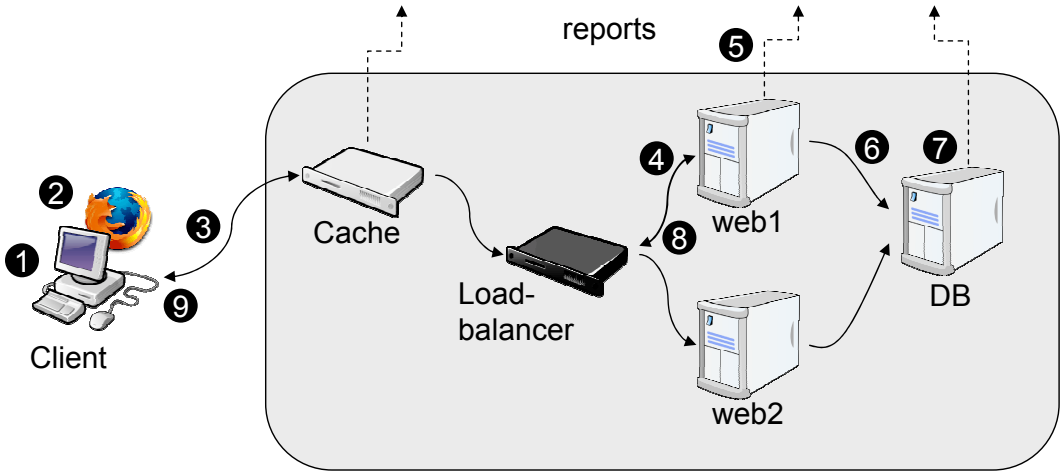


Figure 4.2: Architecture of the web hosting scenario

4.1.3 Fault isolation

An X-Trace enabled DNS 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 4.1 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.

4.2 A web hosting site

4.2.1 Overview

The second scenario that we consider is a web hosting service that allows users to post and share photographs (see Figure 4.2). We deployed an open-source photo application in our network on an IBM Bladecenter. The front-end webserver host Apache and PHP. The photos, metadata, and comments are stored in a Postgres database. Also included are a cache and load-balancer. The photo site has attracted approximately 200 visitors a day for a period of two months.

For this site to support X-Trace, we implemented a reporting module for Apache, and one for Postgres. To support legacy web clients, we implemented an "X-Trace headers" module that inserted X-Trace headers into requests from the legacy clients.

X-Trace can be invoked by either end users or by the operator. End users can invoke X-Trace in two ways: by using an X-Trace-enabled web browser, or an X-Trace-equipped web page. We implemented an X-Trace toolbar for the Firefox web browser that puts X-Trace metadata in requests. We also implemented a Javascript/PHP library that added a feature to selected webpages in the site that let the user report problems via an HTML form. These reports were internally coupled with the X-Trace metadata of the user's request, enabling the network operator to match their complaint with a trace of their session. This is a powerful mechanism to detect semantic faults that would appear normal from the web site's perspective, such as stale pages or logic errors in a well formed response. This is not necessary for all faults, since many requests might generate anomalous task graphs that can be analyzed with methods such as Pinpoint [29].

4.2.2 Tracing a request through the scenario

The client application (i.e., Firefox with our X-Trace extension) creates a new X-Trace metadata and initializes its task and operation id fields. It issues an annotated request to the front-end cache. This cache issues a report based on fields in the request and the X-Trace metadata. It forwards it on, possibly to other middleboxes such as load balancers that might also be on the path. When the Apache process on the front-end tier receives the request, it issues a report that includes the URL, status code, and time of the request.

The PHP-based photo software creates SQL statements to retrieve images and metadata from the backend database. We modified this code to retrieve and propagate the X-Trace metadata from the array of HTTP headers. The new metadata is propagated to the database by enclosing it in a SQL comment (i.e., `/* X-Trace:013A2E... */`). The query is sent to the database, which looks for embedded X-Trace metadata. It issues a report containing the query. When the webserver sends the response back to the client, it adds two headers to the response: one has the X-Trace metadata (in case it was generated by the webserver), and the other has a URL that the client can access to examine the trace of the request.

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

4.2.3 Using X-Trace

In this section we introduce several faults into the photo hosting site. These are based on our first-hand experience with the deployed system.

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. Examining Figure 4.3 shows immediately that the fault is the former–there are no reports from the database, pinpointing the problem to the PHP script. Figure 4.3 shows a square node that represents a problem report issued by the user, using the PHP/Javascript web problem reporting tool. 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 will include nodes up to and including the cache, but will 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 does not 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 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.

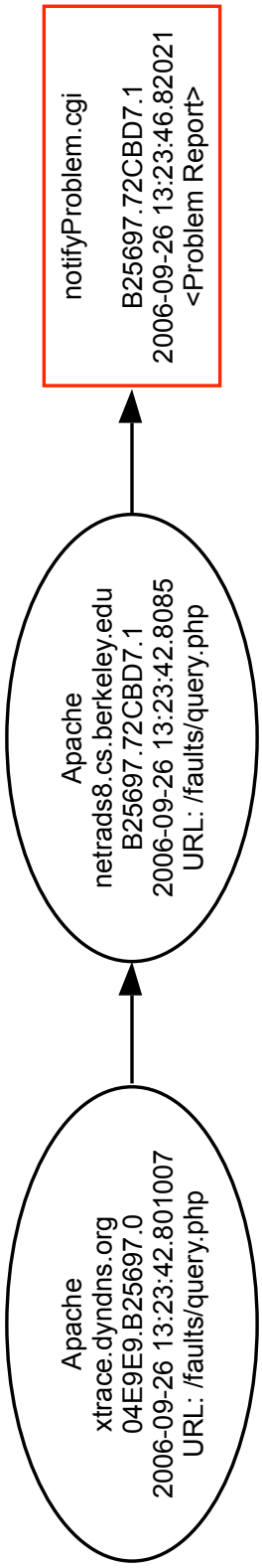


Figure 4.3: An HTTP request fault, annotated with user input

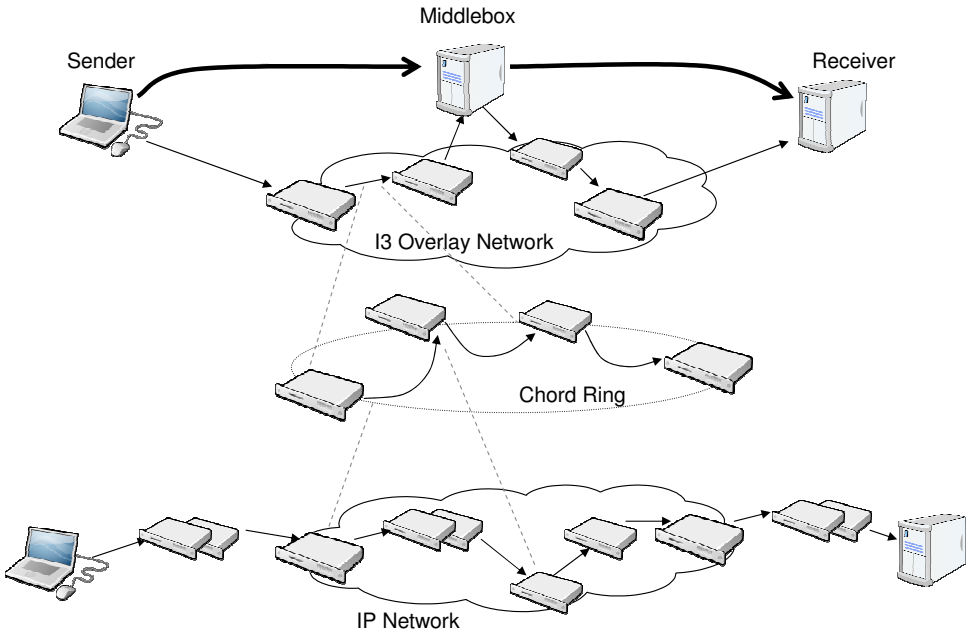


Figure 4.4: X-Trace on an I3 overlay scenario. A client and a server communicate over I3. Shown are the Chord network on top of which the I3 servers communicate, and the underlying IP network.

4.3 An overlay network

The third scenario we look at in some detail is an overlay network. 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 [61], reliability [8], SIP support [102], and data storage [118] 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 [116]. For our purposes, it suffices to say that 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

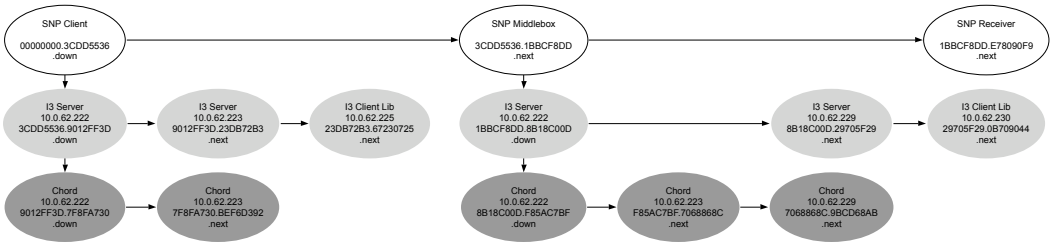
the Chord DHT [118], which provides efficient routing to flat identifiers and is an overlay network on its own.

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. The scenario topology is shown in Figure 4.4, 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. This situation is analogous to having a Web proxy in an HTTP request path, or a video transcoder. We used a toy protocol we called SNP – Simple Number Protocol – that is simply sending 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.

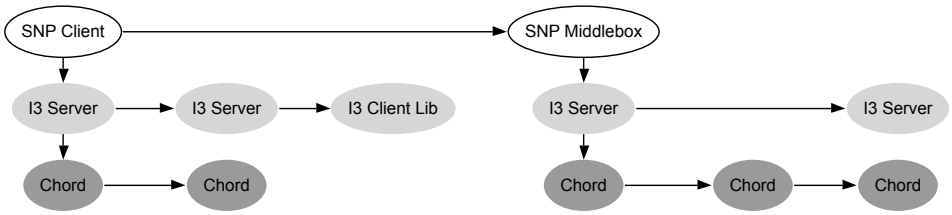
4.3.1 Tracing a message through the scenario

In Figure 4.5(a) we show the reconstructed graph of operations given by X-Trace in a sample run of the scenario. This graph was generated from X-Trace reports by the visualization tool we developed. We deployed an I3 network consisting of 3 machines, each of which was also Chord node. The SNP client, receiver, and middlebox are on separate machines. We omit the IP report messages: all IP paths are one hop, since the machines were all on a switched LAN.

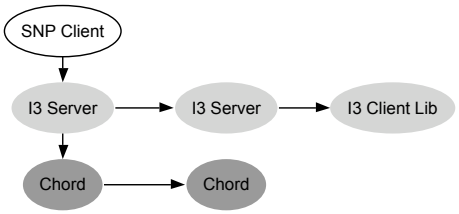
The SNP client sends a message to the the SNP receiver (see Figure 4.5), and it interposes the SNP middlebox on the path. The following is a detailed look at the transmis-



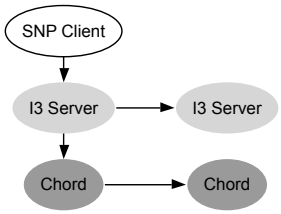
(a) Graph for normal operation



(b) Fault 1: The receiver host fails



(c) Fault 2: Middlebox process crash



(d) Fault 3: The middlebox host fails

Figure 4.5: (a)X-Trace graph 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.

sion of a message in this scenario.

The SNP client creates a message, chooses a *TaskID* and operation id, 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 copies the metadata into the I3 layer. Two more propagation operations copy it into the Chord and IP layers. 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 identifier to the Chord and IP layers. The next Chord node in the path, 10.0.62.223, receives the message. It sends a report, and then since there is no I3 layer, it simply forwards the message on. This 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.

4.3.2 Using X-Trace

In Figures 4.5(b), (c), and (d) we injected different types of faults and show how the resulting X-Trace 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. We chose these particular faults since they exhibit the same symptoms to the user, however they have different causes. These faults cover different parts of the datapath space, including node failures and application process failures.

Fault 1: The receiver host fails In Figure 4.5(b) we simulated a crash in the receiver. 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 4.5(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 graph 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 4.5(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 after the graph indicate that the entire node has failed.

4.4 Additional X-Trace Uses

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

4.4.1 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). In the first case, the lack of universal deployment of the IPv6 protocol can be overcome by bridging IPv6-enabled parts of the network with tunnels residing over IPv4. In the second case, enterprise networks can communicate with different parts of the network over secure, encrypted channels. 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. By propagating metadata across layers, the tunnel itself will contain the X-Trace identifier needed to send trace data about the tunnel to the sender. This helps diagnose faults in the tunnel, which are nontrivial to isolate otherwise.

4.4.2 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 can help quickly identify the location of the problem, and thus reduce support costs.

4.4.3 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 [11]. Retrofitting X-Trace into Ethernet is not possible, due to its lack of extensibility. However, X-Trace metadata can 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.

4.4.4 Development

Tracing tasks is needed at one point or another in the development of distributed applications and protocols for debugging and verification. As with standard logging subsystems, developers can integrate X-Trace into their applications. It is actually being used by the team developing the Data-oriented Network Architecture (DONA) [26], a content-based routing scheme for the Internet.

4.5 Discussion

While X-Trace has many uses, it also has limitations. We discuss those here, as well as other relevant tracing issues.

4.5.1 Report loss

If the reporting infrastructure loses any reports, the effect to the graph will be the deletion of nodes and edges represented by that report. The loss of reports makes the graph reconstruction operation harder, leading to a corrupted graph. Depending on the severity and the specific report involved, report loss can make it impossible to reconstruct causal connections. However, even without being able to reconstruct these connections, the ability to group network operations by a common identifier remains quite useful. In scenarios with lost reports, rather than trying to reconstruct the task graph, a better choice might be to order the reports temporally. Then, the operator can get a picture of the linear progression of events related to a given request.

4.5.2 Managing report traffic

The structure and complexity of an application's task graphs 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. Several mechanisms have been put into place to limit the volume of this traffic, while maintaining its usefulness. *Sampling* can limit the number of requests that are tagged with X-Trace metadata to a rate specified by policy. A low sampling rate is ideal for "always-on" tracing used to get a picture of the behavior of

the network. Differently from independent sampling at each node, using X-Trace, each "sample" is a complete task graph. Since X-Trace reports are delivered out-of-band, they can be *batched* and *compressed* before transmission. Within our network we have observed a 10x compression factor for X-Trace generated reports. Finally, *scoping* can be used to limit report generation to certain network layers, devices, or parts of the network. Layers such as IP generate many reports per request, since reports are generated on a per-packet basis. By limiting the scope of reports to those layers above IP, a smaller volume of reports is generated. Of course, if a fault is suspected at the network layer, the scope of reports could be widened to include IP packets of interest, from a client or subnet experiencing the observed problem. Currently, support for scoping is statically configured into the reporting infrastructure.

4.5.3 Partial deployment

Thus far, our discussion has focused on a comprehensive deployment of X-Trace throughout the network. However, even when X-Trace is partially deployed within one particular application or network layer, it still provides useful tracing benefits. For example, by integrating X-Trace into the I3 and Chord overlay networks, users of those systems can track the mapping of I3 messages to Chord nodes. Alternatively, the developer of a middleware system could use X-Trace to follow requests from one node to another. In this spirit, researchers developing the DONA [26] project are making use of X-Trace to aid in the development of their new routing protocol.

Specific ADs can deploy X-Trace within their networks without requiring any cooperation or support from other ADs. For example, a service provider could deploy X-

Trace at strategic points within their datacenter. This provides the service provider with the task graph within their network. We see the adoption of X-Trace following this partial deployment strategy.

4.5.4 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: we expect ADs to generate very few wire-area reports with pointers to detailed, independent stores for local reports within each AD. Lastly, this problem is more prevalent

when the destination for reports are IP addresses. Using wire-area destinations like I3 or OpenDHT leverages these systems' denial of service prevention features. X-Trace keeps control of report generation rate and visibility with each report provider, which allows for defense mechanisms to be put in place.

4.6 Conclusions

Internet applications are becoming increasingly distributed and complex, taking advantage of new protocol layers and middlebox functionality. Current network diagnostic tools only focus on one particular protocol layer, and the insights they provide on the application cannot be shared between the user, service, and network operators. We propose X-Trace, a cross-layer, cross-application tracing framework designed to reconstruct the user's task graph. This framework enables X-Trace enabled nodes to encode causal connections necessary for rebuilding this graph. The trace data generated by X-Trace is published to a reporting infrastructure, ensuring that different parties can access it in a way that respects the visibility requirements of network and service operators.

We deployed and evaluated X-Trace in two concrete scenarios: a web hosting site and an overlay network. We found that with X-Trace, we were able to quickly identify the location of six injected faults. These faults were chosen because they are difficult to detect using current diagnostic tools.

The data generated by X-Trace instrumented systems can serve as the basis for more sophisticated analysis than the simple visualization and fault detection shown here. Using this data for new and existing algorithms [29, 99] is the object of our ongoing work.

Given that the provider of reports ultimately controls how much data is generated, we are also investigating strategies to push filters on what to report as close to the sources of data as possible. For example, an AD could push a filter to all of its reporting daemons to not send reports on the IP layer.

In the next Chapter, we present the 802.1X network authentication framework, which is an example of a large, distributed enterprise application that has been retrofitted with X-Trace. By making use of the graphs collected from its execution, we are able to identify failures in underlying network components, leading to a more robust and reliable application.

Chapter 5

Exposing network service failures with application-level datapath traces

Software developers use tracing and logging to detect failures in their applications. However, application failures are not always the result of logic or programming mistakes. Failures in the underlying network infrastructure and contention for network resources also affects applications. In the previous chapter, we described X-Trace, a distributed application tracing framework. With X-Trace, we gain insight into the series of events making up the execution behavior of instrumented applications. In this chapter, we will apply X-Trace to a the well-known distributed network authentication protocol IEEE 802.1X [1]. In addition to detecting failures in this system, we will use the traces gathered from X-Trace to infer failures in its network environment. In the next chapter, we will ex-

```
142.68.116.156 [24/May/2007:10:46:30 -0700]
    "GET /apps/ HTTP/1.1" 404 203
142.68.116.156 [24/May/2007:10:46:31 -0700]
    "POST / HTTP/1.0" 200 2243
142.68.116.156 [24/May/2007:10:46:31 -0700]
    "GET /ar/ HTTP/1.1" 404 201
142.68.116.156 [24/May/2007:10:46:31 -0700]
    "GET /archive/ HTTP/1.1" 404 206
```

Figure 5.1: Sample log entries from the Apache webserver. Each entry contains the client IP address, date and time, request string, and status information about the response, including the time taken to serve the content.

plore how to leverage trace data to reason about the performance of applications deployed in virtual datacenter environments. These environments are characterized by highly variable performance, and so we consider the implications for instrumenting applications and designing a trace collection system that for that environment.

5.1 Motivation

Software developers and network operators rely heavily on application-generated log files to debug Web 2.0 applications, since those logs give hints about the conditions under which it was operating. These log files consist of individual log lines, each containing portions of the system state that are captured via programmer-inserted instrumentation points. An example log file from the Apache webserver is shown in Figure 5.1. Examples of such state include information about user requests, the values of program variables, the existence and cause of exceptional conditions, and the success or failure of invocations into required application components. A single user interaction with a Web 2.0 service might result in thousands of log messages generated by dozens of software components, distributed

across multiple machines. Debugging begins by examining these logs after an error is discovered. The examination begins at the initial point of contact with the user, through the set of invocations from that point to the various components the application relies on, until an erroneous condition is discovered, or the trace diverges from what the operator intends and expects. In addition to bugs present in the application itself, failures in Web 2.0 systems can be the result of failures in the underlying network and required network services. Errors in configuration specifications, as well as unexpected configuration interactions can lead to unexpected traffic patterns, which could in turn lead to failures.

Before we describe our approach, we will define some terms. *The network* is the set of host network interfaces, network cables, switches and routers, and devices that reside on the datapath, yet are not typically addressed by end hosts. Such devices include firewalls, proxies, and network address translation (NAT) systems. Examples of network failures include link failures and packet loss, malfunctioning middleboxes, unexpected traffic surges that impact application traces, and failures in switching and routing protocols that result in a loss of connectivity between two end hosts. We define a *network service* to be a software component that is used by, but not part of, an application. Network services provide either data, or processing, or both to an application, are typically provided by the network operators, and are usually shared between multiple applications. Examples include the Domain Name System (DNS) and the Lightweight Directory Access Protocol (LDAP). Since Web 2.0 applications are distributed across the network and rely on network services, failures in those underlying components now directly affect the application's correctness. It is not enough to look for bugs just in the application's code. Failures in the application might also

be the result of network and network service conditions.

5.2 Approach

We argue that while application log files are primarily designed to detect failures in the application itself, we can also use them to infer failures in the underlying network and network services. In this chapter, we will enhance application log files with X-Trace trace data, and use the resulting dataset to infer underlying failures in the application's network environment. Since there is already a wealth of network monitoring tools available, one might wonder why one would want to use application-layer traces to infer lower-layer failures. There are several motivations for doing so:

1. The administrative division of responsibility often means that the application developer and the network infrastructure team are two separate entities. Thus an application developer might not have access to the underlying network hardware.

2. Failures observed in the network might only be detectable from the point of view of the application. For example, modifying a firewall rule may block an application from reaching one of its required components. The network operator has no reason to know that dependency exists, since the software is highly dynamic, and thus they would not have realized that the firewall configuration change poses a problem. As we will show in the evaluation section, this specific problem has shown up in the networks of customer deployments.

3. With the increasingly virtualized datacenter environments provided by such compa-

nies as VMWare [126] and Amazon [6], access to the "true" underlying network may be hidden from the application and application developer by design, making the management of co-located applications simpler, as well as decoupling network resources from the applications. This might be done to better handle traffic demands or to mask failures. These layers of virtualization pose a challenge for determining the status of the network itself, due to the virtualization layer. Furthermore, one part of the application might be unable to reach a required network service due to failures in the virtualization layer. For example, an "invisible" load balancer introduced by the virtualization layer may not properly load balance the traffic. These types of errors are very difficult to detect.

In this chapter, we will present a methodology for detecting network and network service failures from higher-level application traces. We use as a case study the IEEE 802.1X network authentication service, which is a representative example of a complex distributed application that authenticates end user devices such as laptops and desktops and authorizes their access to the network. 802.1X is built out of heterogeneous network protocols and relies on network services, in particular LDAP. Its function is critical to end-users, since their connectivity to the network depends on its proper function. It is also becoming widely deployed in enterprise and campus networks. To evaluate our approach, we will test our inference scheme on a deployed 802.1X system injected with a variety of network and network service failures. These failures were suggested to us by support technicians at a wireless LAN authentication network appliance vendor. They identified various real-world faults that have occurred "in the wild" in customer networks. We have

used their experience to generate a failure taxonomy that defines the set of failures that we evaluate against.

While our approach shows promise, its efficacy relies on a set of underlying assumptions:

1. Underlying failures are only exposed if the application trace and logs capture information relevant to the failure. The more detailed those sources of data, the more failures we can detect. We will show both failures observed by direct examination of the trace, as well as failures inferred from the trace.

2. Traces and logs show only the effects of underlying failures, not the causes. Since trace data reflects the application's datapath, it is able to capture deviations from known "good" behavior. While a valuable diagnostic tool, this differs from diagnoses obtained by directly measuring the underlying cause of failure. As a concrete example, an application trace might show that all traffic transiting a web load balancer is sent to only a single host. Direct underlying measurements might show that the cause of this behavior is an incorrectly configured web load balancer.

3. The latency between an instance of a failure and its detection is a function of the trace and log collection infrastructure. There is an inherent tradeoff between systems that detect failures in real time, but require a large amount of overhead to do so, and systems that have a lower impact on their network surroundings, but rely on techniques like buffering and compression, both of which increase the latency to failure detection.

4. Our approach relies on modifying applications to add tracing instrumentation, as well as installing trace collection software on each instrumented host in the system.

5.3 Case Study: IEEE 802.1X

We choose to examine 802.1X because of its increasing importance in enterprise environments. However, due to its structure and composition, tracing 802.1X is challenging. This challenge arises from the need to modify different binary protocols to carry metadata, as well as different software modifications to propagate this metadata through the 802.1X components, which are provided by different vendors.

In the remainder of this chapter, we will show how to leverage those traces to also detect and diagnose failures in the network surrounding the 802.1X distributed application application-layer traces to infer underlying failures is the second We now examine the motivations, and challenges, of adding trace support to 802.1X.

5.3.1 Improving 802.1X observability: Motivations and challenges

We now apply our approach to the IEEE 802.1X [1] network authentication system, an authentication protocol used by edge network devices to authenticate end-users. Although not a traditional application, 802.1X is a distributed system consisting of multiple tiers of software and hardware. As more and more critical business applications are made available on-line, controlling access to network resources ensures that these applications remain secure. In health care and financial networks, network access must be logged and auditable, providing a trail that leads back to registered network users. Thus, because

of its central importance to network administration, monitoring and tracing 802.1X is an important goal for network operators. Improved observability into 802.1X enables:

1. Detecting flash traffic and denial-of-service attacks on the authentication infrastructure

2. Detecting and diagnosing errors that prevent users from properly authenticating. For example, failures in the network might prevent client requests from being properly handled.

3. Improving the placement and provisioning of authentication components in the network. As we will show in Section 5.3.2, 802.1X components must be properly provisioned to handle expected, and sometimes unexpected, user loads.

Despite the importance of improving the observability of 802.1X, the addition of trace support must overcome a set of challenges. First, the authentication infrastructure is distributed for scalability and reliability purposes, and there is no single point for collecting end-to-end traces of individual requests. Second, 802.1X is a framework that marries multiple different protocols together, and as such, there is not an easy way to trace the composed path. Since that path is made up of multiple underlying IP paths, tools like "traceroute" are not usable for this purpose. Third, different 802.1X components are often in different administrative domains. For example, the network devices enforcing access control are typically part of the core infrastructure, whereas the databases and directories that contain user information are part of the application space. Tracing across different administrative domains can lead to problems of data privacy and access control. Fourth, single-vendor

solutions are unlikely to work in this area, since the components of 802.1X are typically made by different vendors. Lastly, any trace collection infrastructure and protocol modifications to support tracing must be backwards compatible. However, as our network grows, we would like to be able to take advantage of any trace functionality that is introduced.

5.3.2 802.1X operation

The complexity of an 802.1X deployment is not only driven by technical requirements, but also by business needs. In a typical deployment, a successful authentication request involves the cooperation of at least four independent systems:

- **Client:** a device that requests network access. Common examples include laptops, PDAs, and shared desktops.

- **Authenticator:** a device that provides network access; it also denies access to a network at the data link layer before authentication is complete or if authentication fails. Most 802.1X authenticators are wireless access points, wired Ethernet switches, or VPN concentrators.

- **Authentication server:** a server responsible for deciding if the client should be allowed on the secured network. It receives credentials from the client via the authenticator and collects other information about the client from identity stores as described below. It then uses authentication and identity information to make a decision, which is returned to the authenticator. Note that the decision and the enforcement of that decision are implemented on independent systems to avoid maintaining an organization's access policy on every authenticator in a potentially large and diverse network.

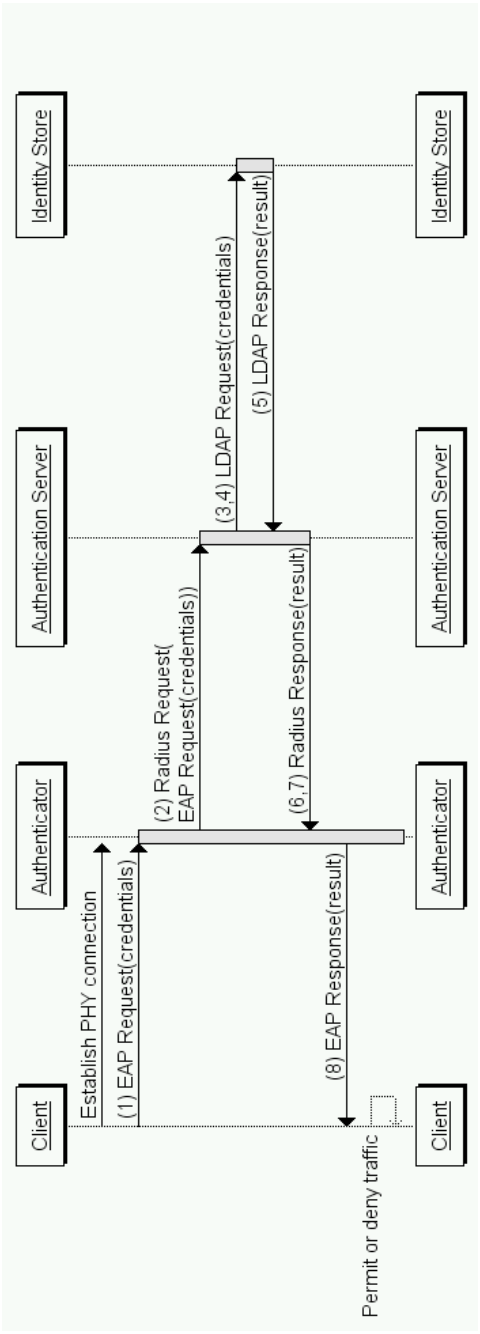


Figure 5.2: Message Sequence Chart (MSC) describing the structure of the 802.1X protocol.

Bit			
0-7	8-15	16-23	24-31
Code	Identifier	Length	
Authenticator			
Type: X-Trace	Length: 11	Flags	TaskID
TaskID (con't)			OpID
OpID (con't)			

Figure 5.3: The format of a modified RADIUS packet, which includes an X-Trace VSA.

In 802.1X, the authentication server is a Remote Authentication Dial In User Service (RADIUS) [100] server. RADIUS was historically used by modem dial-in banks to centralize authentication data. The RADIUS protocol uses UDP as an underlying transport. Figure 5.3 outlines the RADIUS protocol data unit.

- **Identity store:** a service that provides further information about the identity of an organization's users or devices. The identity information can take the form of credentials, group membership, or user attributes. Most identity stores used in 802.1X are LDAP [127] directories, but Kerberos, token servers, NIS, and various databases are also occasionally used.

To further complicate matters, both the authentication servers and identity stores are often implemented as redundant pairs because of the critical uptime requirements for network access. Also, multiple types of identity stores can be used to make a single access decision: for example, one store may contain user attributes while another may contain credentials.

To better understand the network path of a typical 802.1X authentication system, and thereby what a typical trace of 802.1X will look like, it is instructive to step through a simple access request:

1. After the **Client** and **Authenticator** establish a physical connection, they begin a conversation using the Extensible Authentication Protocol (EAP) [17]. This conversation occurs directly over the link layer because the two systems are directly connected and the client has not yet been provisioned on the network to which it is requesting access.

2. The **Authenticator** forwards the EAP conversation to the **Authentication Server** over RADIUS. The actual method of authentication contained inside the EAP tunnel is unimportant to the **Authenticator**: it continues to forward EAP messages between the **Client** and the **Authentication Server** until the server makes a decision. Common EAP authentication algorithms include shared key, one-time password, and mutual authentication using SSL certificates. [Event A]

3. The **Authentication Server** sends a request for credential validation to an **Identity Store**. In the common case of shared key authentication, a it will simply perform an LDAP bind. [Event B]

4. The **Identity Store** receives the request from the **Authentication Server**. [Event C]

5. The **Identity Store** responds to the **Authentication Server** with a success or failure result. [Event D]

   (Steps 3 and 4 may be repeated if the server must collect further user data from identity stores.)

6. The **Authentication Server** makes a decision and sends it in the form of a RADIUS accept or reject packet to the **Authenticator**. [Event E]

7. The **Authenticator** receives the success or failure response. [Event F]

8. The **Authenticator** returns an EAP success or failure packet to the **Client**. In the case of success, the **Authenticator** begins to forward non-802.1X traffic between the **Client** and the secured network. [Not shown]

Our reliance on in-datapath tracing also allows us to trace the exact conditions that the end-to-end request underwent. The alternative, separate trace requests, transits a different end-to-end path, or measures different network phenomenon. Additionally, using datapath tracing provides us with a very fine-grained trace, rather than a more aggregate or summary view. However, our approach is not without its drawbacks. Datapath tracing requires the insertion of trace metadata into protocol messages. Additionally, the software must be modified to propagate this metadata along the path. Lastly, a certain volume of generated traces must be collected and aggregated together for analysis. Fortunately, we were able to overcome these challenges with minimal changes to the software running on each machine. Additionally, the trace metadata encoding we use is backwards compatible to legacy devices, reducing the barrier to introduction in production networks. In the discussion at the end of this chapter, we present several recommendations to protocol designers on designing more observable and traceable distributed, composed protocols.

5.4 Design

So far we have described the 802.1X network authentication protocol–a complex, distributed application running within an enterprise network. This application relies on a variety of underlying network services, including network switches, RADIUS servers, and LDAP identity providers. We seek to identity failures within 802.1X components them-

selves, as well as failures due to the dynamics of the surrounding network environment. These dynamics result from underlying failures, misconfigurations, and co-located applications, which share the same components as 802.1X itself. We now turn our attention to extending 802.1X to gather instrumentation data about its execution behavior. Our primary tool in this effort is path-based tracing, which we now discuss after defining the problem that we want it to solve.

5.4.1 Problem statement

Using an application trace of an 802.1X deployment, detect the presence of, and likely cause of, faults in underlying network and network services.

5.4.2 Approach

Our approach is based on an observe-analyze-act approach. We first instrument the application to provide data on its distributed execution behavior. We then analyze this data, inferring the root causes of observed failures. Lastly, we act by notifying the operator of our findings. This approach consists of the following discrete steps:

1. Collect application traces using X-Trace
2. Determine when a fault is occurring
3. Localize the fault
4. Determine the root cause, if possible
5. Report problem and root cause (if known) to network operator

We now consider each of these steps in detail.

5.4.3 Collecting application traces using X-Trace

Our general approach to collecting application traces from distributed systems is outlined in the X-Trace Chapter and previous work [48]. In this effort, we deployed X-Trace in a testbed environment, and not in Identity Engine's product. However, their product has several features that facilitates integration with X-Trace. First, the product logs each 802.1X event in local storage using Syslog [84]. These logs are periodically (typically once an hour) collected and stored on a central server. By logging X-Trace reports to syslog, no additional daemon processes need to run on the node to handle reports, and the generated reports could easily be extracted at the central server. Second, their product includes a set of interfaces and operators control panels for managing and updating configuration state. X-Trace configuration could piggyback on that interface, obviating the need for a two-way control plane between the X-Trace backend and X-Trace itself. Lastly, the user interface tools provided by Identity Engines would be a natural site for visualizing X-Trace graphs and trace data.

5.4.4 Fault occurrence detection

In a large production network, determining when faults occur is non-trivial. In general, we can leverage three sources of information for indications of network service faults. First, we can use application-specific semantics to understand which parts of the datapath trace are faults. For example, failing over from one service to another might indicate a fault in the service. Second, we can look for patterns in well-known protocols as a

signal of error. For TCP, packet retransmissions indicate a network error. In HTTP, there are certain well-known error codes, e.g., “404” and “500”, that indicate missing documents or server-side errors. Lastly, we can make use of user-initiated trouble tickets and complaints to determine erroneous conditions.

5.4.5 Fault localization

The task of fault localization is important, since we wish to present a coherent set of information to the network operators. Thus, if a single node in the distributed service were to cause faults in the process of servicing multiple clients, we would prefer to inform the operator about a service failure rather than issue multiple alerts. Fault localization is a very difficult problem in general, and many attempts have been made in specific contexts. In this work, we assume that each end-to-end authentication operation suffers from at most one fault at a time. If this assumption does not hold, then our system diagnoses the first fault temporally along the datapath. This is a common assumption in distributed system design. Once that is fixed, and operations resumed, then it detects the next subsequent fault. This is due to the nature of our diagnosis, which is based on application traces, which only tell us information about components the application interacts with.

5.4.6 Individual root cause determination

Once we have detected a fault, we try to determine its root cause. Depending on the root cause, this involves analyzing the structure of a particular X-Trace trace or analyzing a set of multiple traces. For well-known protocols, we look for error conditions based on knowledge of the protocol semantics. To identify graph structures that indicate

faults in the network service itself, we must know more about how those errors appear in the trace.

5.4.7 Reporting faults to the operator

Lastly, we alert the operator with our findings. This alert is not delivered in real-time, but rather is delivered to the operator within minutes of its presence. The information that we provide includes the presence of the fault, localized to a particular part of the network, as well as any root causes that we were able to determine. Since our analysis is based on X-Trace graphs, we also return a handle to that graph to the operator. This provides a detailed history of the execution behavior of the misbehaving application, and a reference to this graph can accompany any bug reports that are issued by the network operators. This ability to refer to an execution graph by its handle provides a powerful mechanism for communicating the circumstances contributing to the failure.

5.5 Implementation

In this section, we describe how we captured our approach in a deployed network authentication system. We begin by describing how we were able to add X-Trace support to the 802.1X protocol.

5.5.1 Integrating X-Trace into 802.1X

Application tracing fits neatly on top of the 802.1X architecture: the X-Trace task represents a single authentication request. We chose not to include X-Trace metadata in the

EAP protocol, which spans the network between client and authenticator, since we did not want to modify end-system devices. EAP is very extensible, and so including additional metadata is straightforward. One intriguing possibility is including the X-Trace metadata info only on the return response from the authenticator to the end-system. For those clients that support X-Trace, they would be able to provide network operators with the trace ID of any improperly rejected access attempts, speeding up the process of failure diagnosis. In this work, we did not implement this response trace token functionality.

The two protocols that we modified are RADIUS and LDAP. RADIUS packets consist of a small set of operations, followed by a variable number of attributes. We encoded the metadata in a standardized, but unused, Vendor-specific Attribute (VSA). Depending on the size of unique X-Trace task ID used, this custom VSA is between 11 and 30 bytes long. For LDAP commands, we included the X-Trace metadata in an LDAP Control. This control was not marked as "critical", meaning that legacy LDAP servers can safely ignore the trace request. We used a custom Object Identifier (OID) taken from the OID space allocated to the X-Trace project (1.3.6.1.4.1.4995.1000.2). Therefore, the X-Trace task begins and ends with the authenticator. An example trace of an authentication request produced by X-Trace is shown in Figure 5.4. This figure was generated by the X-Trace tool using the GraphViz[54] library.

We limited our instrumentation to the set of components provided as part of the Identity Engines network appliance product. This reflects a realistic environment in which a vendor has complete control over a subset of devices in the network, but no access to the others. From the vantage point of such a vendor, they can only observe traffic transiting

their own components, and so must infer the behavior of the rest of the network.

Fault Occurrence Detection

When authorized clients attach to the network and invoke the 802.1X protocol against an authenticator, one of several outcomes are possible: 1) The client is correctly authorized or properly denied, 2) The client is authorized, but after an excessive time delay, 3) The client is improperly rejected, or 4) The client times out after waiting for the authenticator, which does not return a response. Client timeout issues can be detected by polling the authenticator (for example, via SNMP). The problem we face is detecting faults elsewhere in the end-to-end 802.1X datapath.

We determined the presence of faults primarily by using authenticator and authentication server timeouts. We needed to use both timeouts since RADIUS servers typically have a "fail-safe" policy that rejects clients when they timeout waiting for the decision points to return a response. In this case, the authenticator receives a rejection, and does not generate a timeout. Additionally, we used timing information to determine when components of the service were running too slowly, such as overloaded LDAP services. Lastly, we determined faults by looking for X-Trace datapath graphs that did not match the six-event structure exhibited by successful traces.

Fault Localization

In our system, we use a simple approach to fault localization that leverages the typically hierarchical geographic distribution of enforcements, decision, and data points throughout the network. We group faults both by time and by network location. We start by

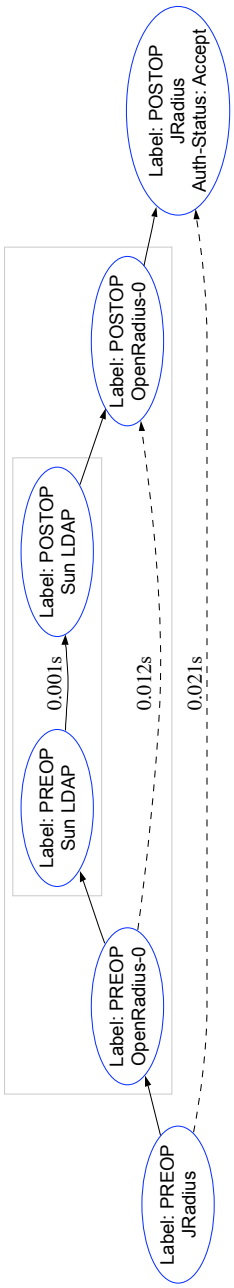


Figure 5.4: X-Trace graph of a typical 802.1X authentication operation. The authenticator is represented by the "JRadius" nodes, the authentication server by the "OpenRadius" nodes, and the identity store by the "Sun LDAP" nodes. Each server appears twice: once on the forward path (the three nodes on the left side of the graph), and once on the reverse path (the three nodes on the right side). The solid lines represent the path the messages took, while the dotted line represents the delay at that component.

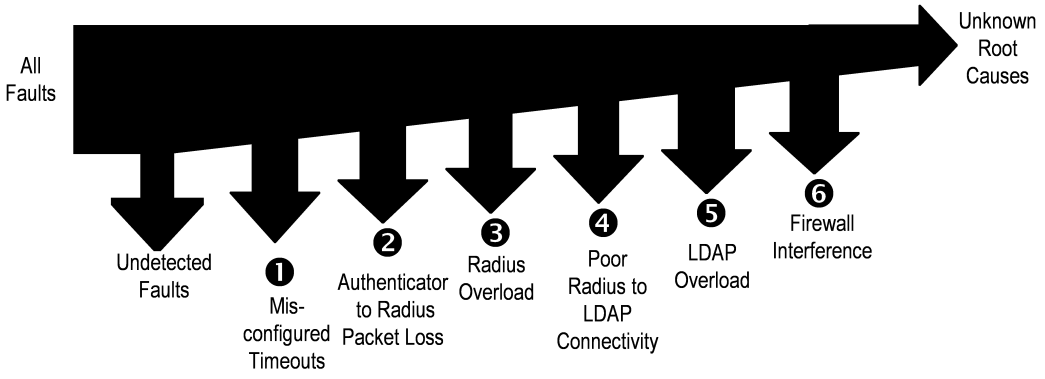


Figure 5.5: Root cause determination.

choosing a time window, T, of approximately five minutes. If there is only one authenticator or RADIUS fault during T, we report that fault as an independent event. If there are multiple authenticator faults during time T, then we group those faults that share the same RADIUS server and root cause. Likewise, we group multiple RADIUS servers that occur during T if they share the same LDAP server and root cause.

Individual Root Cause Determination

Figure 5.5 represents the set of root causes to those that we can detect (or, we indicate that we "don't know" the root cause). We now consider each step in our approach in turn.

1. Misconfigured Timeouts We infer that a device has a misconfigured timeout when it times out before it would have otherwise received information needed to properly make an authentication determination. With X-Trace, misconfigured timeouts can be detected by looking for an authenticator that times out at time T_1, only to have a RADIUS server issue a

report at $T_2 > T_1$. Likewise, a RADIUS server with a misconfigured timeout is detected by looking for an LDAP server that issues reports shortly after the RADIUS server times out.

2. Authenticator to RADIUS Packet Loss Since the RADIUS protocol resides on UDP, packet loss between the authenticator and the RADIUS server results in the loss of the entire RADIUS request. This type of packet loss can be detected by looking for X-Trace paths in which there is an authenticator report but no corresponding RADIUS report. To detect loss on the reverse path, we look for X-Trace paths in which the RADIUS server sends a response to the authenticator at time T_1, only to have the authenticator time out at time $T_2 > T_1 + \delta$, where δ is greater than the one-way latency between the RADIUS server and the authenticator.

3. RADIUS Overload Delay in the RADIUS server can result from background tasks that induce resource contention (for example, of a disk), resulting in a degradation of service quality. I/O contention is not related to a specific service path, but it impacts paths that transit that RADIUS server. Regardless of the underlying cause, an overloaded RADIUS server would be expected to reject RADIUS requests during overload. This can be achieved by managing the length and contents of an incoming work queue. When the server is overloaded, using any internal metric that might be server-specific, it would find RADIUS requests in its queue and remove them. By additionally issuing an X-Trace report whenever a RADIUS request is evicted from the incoming work queue, we can provide a deterministic signal that a failure in the 802.1X service path is due to overload at the RADIUS server. Determining if a RADIUS server is overloaded requires looking for an "overload" report

along the path.

4. Poor RADIUS to LDAP Connectivity For traffic using TCP, packet loss manifests itself as a decrease in throughput between the RADIUS server and LDAP server, as well as an increase in the variability and burstiness of that throughput [81, 78]. In general, background flash traffic, packet loss, switch failures, and network device firmware bugs can result in increases in end-to-end latency variance resulting in connection degradation. TCP was designed to accommodate increases in variance, however datacenters are tightly controlled and provisioned environments, and thus large variation in TCP performance are engineered to be uncommon. Factors such as lost packets which affect this performance should be minimized in datacenter environments. The heuristic of considering sudden increases in the variance of query latency as signals of packet loss is simple, but overly course. We do not add X-Trace instrumentation to the transport layer, and so are unable to definitively pinpoint TCP performance degradation due to packet loss. This limitation of X-Trace is discussed in Section 5.6.5.

5. LDAP Overload The performance of LDAP under load is a factor of the number of queries and bind operations it receives per unit time, the mixture of query types, and the quantity and organization of its data store, among other factors [129]. Thus, as the load on the LDAP server increases, we would expect that the latency of LDAP operations would increase as well. However, unlike some other network services such as HTTP, with user-perceived timing requirements, in this application LDAP must respond within a fixed amount of time. Thus, we apply a threshold test to the observed LDAP query latency,

determining an error condition whenever the query latency exceeds a predetermined value, in our case, 100 ms. In our experience, any deployment with latency greater than this value is exhibiting an error condition. Refining this threshold test to a more versatile way of differentiating between the "normal" increase in latency and "abnormal" increases in latency due to faults is the subject of the next chapter.

6. Firewall Interference Network operators often put LDAP servers, which are considered "application" layer technology, in a different part of the network than the core network infrastructure. Thus, it is sometimes the case that LDAP requests from RADIUS servers have to transit a firewall before they reach the identity store. Network operators frequently update firewall rules as the network grows, new applications are deployed, and as outside connectivity changes. It is possible that a change to the firewall configuration can inadvertently interrupt the connectivity between the RADIUS server and the LDAP server. Since the RADIUS server cannot access the identity data it needs, users are unable to gain access to the network. Since X-Trace can optionally extend to the network layer, it is possible to directly detect loss due to firewalls.

Analysis: Identifying Root Causes

Following our observe-analyze-act approach, we have shown how to use X-Trace to collect graphs representing individual authentication execution histories. We now describe how to analyze these graphs. The data structure we use is a decision table, shown in Figure 5.6. The top portion of the table defines an exhaustive set of conditions that occur in a given graph. The bottom portion of the graph encapsulates root cause diagnoses.

We begin by computing a vector *v* based on the presence of nodes in the X-Trace graph. The authenticator start and end nodes are labeled *A* and *A'*, the RADIUS server start and end nodes are labeled *B* and *B'*, and the LDAP server start and end nodes are *C* and *C'*. RADIUS overload reports are identified as *B"*. If timeouts are present in the graph, they are also captured in vector *v*. To use the decision table, we match *v* with its matching column in the top half of the table. We identify the set of diagnoses based on any "X" values in the lower half of the table. Note that a given set of conditions can lead to multiple diagnoses, in which case we notify the operator of multiple possible root causes.

Reporting Faults to the Operator

Currently in our system, reports of faults are simply determined statically from the X-Trace datapath traces using our X-Trace tools. There are many industrial tools for detecting and managing failures, and the output of our X-Trace analysis could be integrated into these systems to give network operators visibility into the failures in their network.

5.6 Evaluation

Now that we have covered the approach of our use case, and outlined the implementation of this approach in a deployed 802.1X software system, we now describe our evaluation of this system. We begin by presenting our plan for evaluation.

Conditions	A present	0	1	1	1	1	1	1	1	1	1	1	1
	B present	0	0	0	1	1	1	1	1	1	1	1	1
	C present	0	0	0	0	0	0	1	1	1	1	1	0
	C' present	0	0	0	0	0	0	1	1	1	1	1	1
	B' present	0	0	0	0	1	1	1	1	1	1	1	0
	A' present	0	1	1	1	1	1	1	1	1	1	1	1
	A ... A' timeout		0	1		0	1	0	0	1	0	0	0
	B ... B' timeout					0	0	1	0	0	1	0	1
	C ... C' timeout								0	0	0	1	0
	B" present	0	0	0	0	0	0	0	0	0	0	0	1
Diagnosis	Success					X	X		X				
	PHY failure	X											
	Client to Auth. pkt loss	X											
	General client failure	X	X										
	Misconfigured auth.			X	X								
	Auth. to Radius pkt loss *			X									
	Radius crash *				X								
	Misconfigured Radius server					X	X	X					
	Radius to LDAP pkt loss *							X					
	Misconfig. Auth. timeout *									X			
	Misconfig. Radius timeout *										X		
	LDAP Overload *											X	
	Radius Overload *												X
	Firewall Interference *	X		X				X					

Figure 5.6: We analyze individual X-Trace graphs using this decision table. The top portion of the table defines an exhaustive set of conditions that occur in a given graph. The bottom portion of the graph encapsulates root cause diagnoses. As an example, given a graph G, we compute a vector of boolean values based on each of the conditions listed (e.g., "B present"). We find the column matching this vector, and identify the set of diagnoses based on the "X" values in that column. Note that a given set of conditions can lead to multiple diagnoses, in which case we notify the operator of multiple possible root causes.

5.6.1 Evaluation plan

We evaluate our approach by setting up a complete network authentication environment within the DETER testbed [131]. This configurable platform allowed us to deploy our modified software in a controlled setting without any background traffic or contention for network resources. We could bring links up and down and modify link delay or loss rates. We utilize widely available, open-source software components for each tier of the 802.1X setup, modified to support X-Trace. Within this environment, clients are emulated by dedicated RADIUS load generators. We chose this evaluation approach since within the testbed, we were able to recreate each of the conditions experienced in customer deployments through fault injection, since DETER provides fine-grained control of network and server resources. The advantage of this approach is two-fold. First, we were able to create individual root causes on demand, and second, we were able to cover the entire set of conditions that we wished to test. In fact, although failures in the infrastructure have significant effects, they are typically rare events.

5.6.2 Experimental setup

We deployed the X-Trace framework API [134] into three open-source 802.1X components: the JRadius load generator tool [70], a RADIUS client written in Java that we used to emulate the authenticators, OpenRadius [91], a modular RADIUS server, and the Sun Directory Server 5.2 [120], which is an LDAP server we used as the identity store. We modified JRadius to include X-Trace metadata in an unused but standardized Cisco VSA. We modified OpenRadius to pass this metadata to the LDAP service via an LDAP control

in an X-Trace OID. To support X-Trace in the LDAP server, we wrote pre-operation and post-operation plug-ins that were loaded at run-time.

To explore overload in RADIUS servers, we modified the OpenRadius software to include multiple work queues. This allowed us to configure OpenRadius to selectively drop excessive work from those queues to emulate the effect of overload due to disk I/O contention. In our implementation, the choice to drop requests was statically determined by our code, rather than from resource contention or true overload. Similarly to emulate LDAP overload and the resulting increase in LDAP query latency, we modified the LDAP pre-operation plug-in to delay requests by a configurable amount, which enabled a direct injection of latency, rather than creating conditions that indirectly resulted in increased query latency.

5.6.3 Results

Root Cause Test 1: Miscalibrated Timeout Value Figure 5.7 shows the result of a client timeout resulting from excessive delay in the LDAP component of the path that was artificially injected via an LDAP delay plug-in. At time $t = 3.012$, the OpenRadius server completes its operation and returns a result to the client. However at $t = 2.034$, the client signified a timeout and instituted its default policy that is to reject the client.

Root Cause Test 2: Authenticator-Side Packet Loss If a packet is lost on the forward path, the RADIUS server does not receive it, so only the client knows that the request was issued. Thus, we see in the trace a set of timeouts issued by the client, without any other datapath elements. Figure 5.8 shows the result of packet loss on the reverse path, in

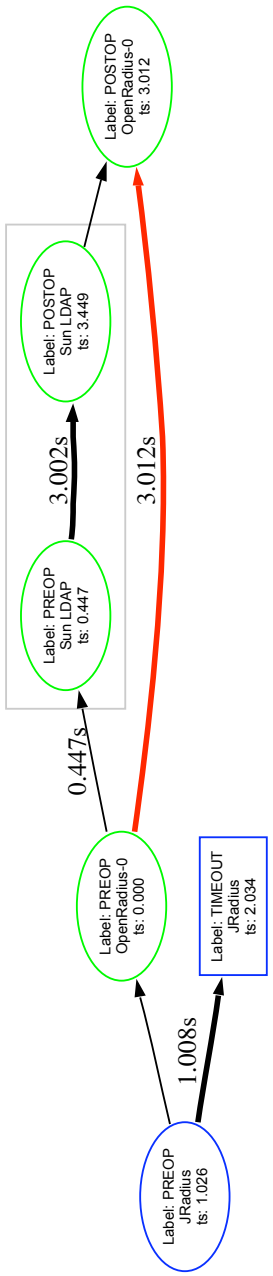


Figure 5.7: Root cause test 1: Miscalibrated Timeout Values. At $t = 2.034$ the client times out, only to have the data it needed arrive at time $t = 3.012$. Note that the timestamps on different machines are out of sync with each other. In this work, all latency measurements are considered from the point of view of a single machine, and we never compare timestamps across different machines.

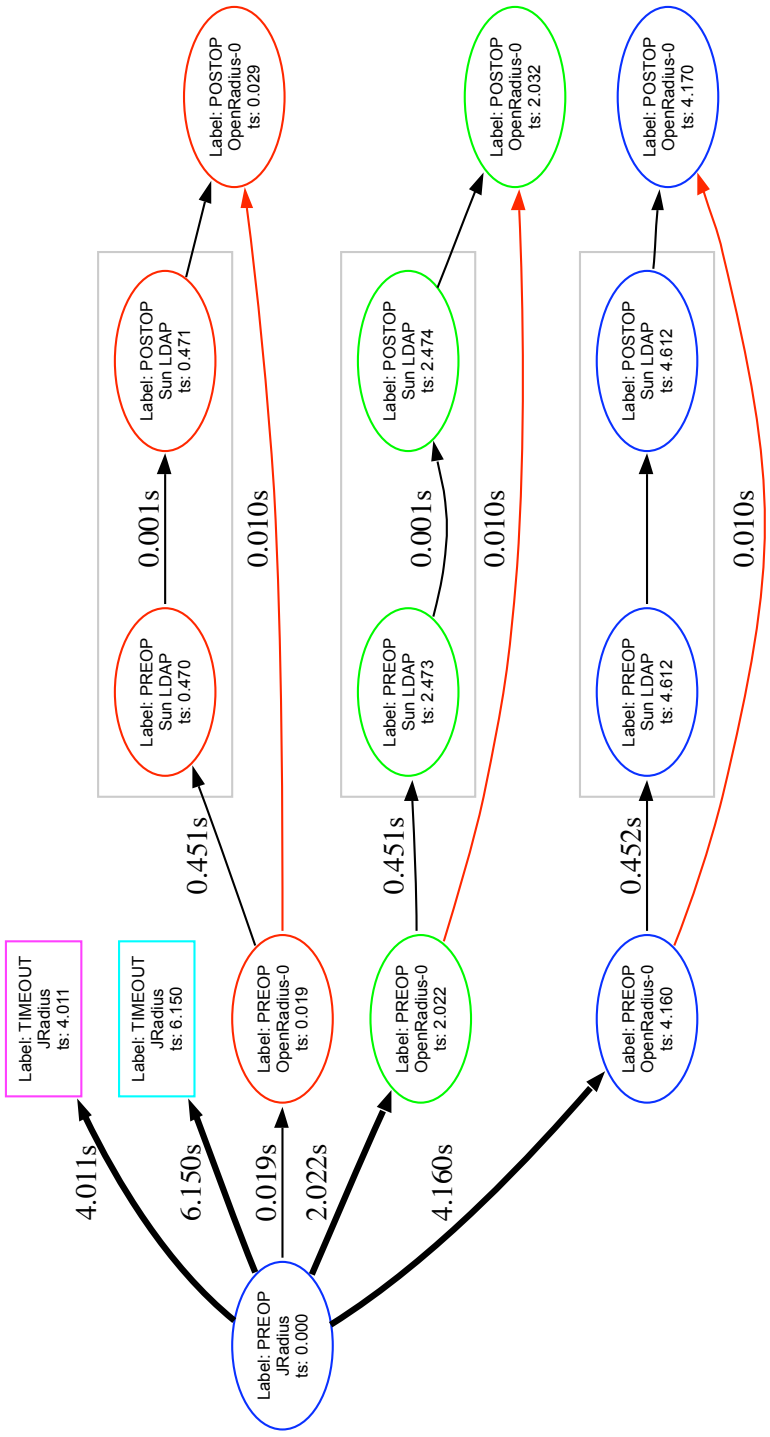


Figure 5.8: Root cause test 2: Authenticator-Side Packet Loss (Reverse path). Packets are lost on the reverse path, meaning that the entire datapath is transited, excepting for the final link between the RADIUS server and the authenticator. Since the authenticator does not receive this response, it retries, and eventually times out.

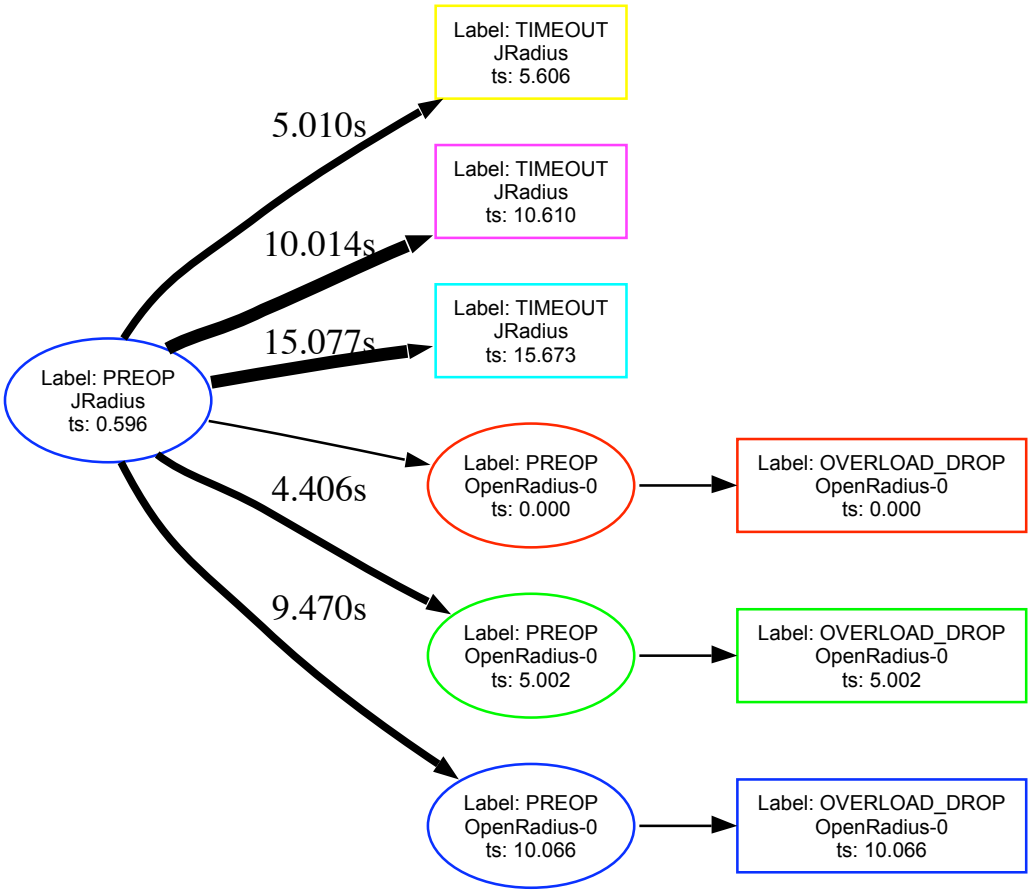


Figure 5.9: Root cause test 3: Detecting RADIUS server overload. Since the modified RADIUS server signals overload via X-Trace reports, we can deterministically detect overload by looking for "OVERLOAD_DROP" events.

which the client successfully invokes processing later in the path but is unable to receive the final response. Thus, it times out and reissues the request. This figure shows three complete protocol exchanges, but since the client did not receive the responses, it issued three timeouts.

Root Cause Test 3: RADIUS Overload We modified the OpenRadius server so that incoming requests are sent to one of several work queues. In the event that the server becomes overloaded, work from these queues is dropped. We modified this policy to examine the

request for X-Trace metadata VSAs, issuing reports if present. The result is presented in Figure 5.9.

Root Cause Test 4: Poor RADIUS to LDAP Connectivity Poor connectivity between the RADIUS server and LDAP server is difficult to measure directly since the underlying TCP channel masks packet drops and other failures from the application. In this work we did not add X-Trace support to the transport protocol, and so we can only measure transport performance indirectly through the authentication query behavior. In general, failures hinder TCP performance, and we can infer connectivity problems by examining the end-to-end query latency. Since datacenter networks should have very low network latency variability, we would expect sudden increases in query latency variance to indicate underlying failures. This inability to peer below the application into the transport is a limitation of the X-Trace system, and one that we discuss in the section on limitations at the end of this chapter.

Root Cause Test 5: LDAP Overload Lastly, we consider the effect of increased LDAP latency, which is detected by measuring the time between reports from the LDAP pre-operation and post-operation plugins. In this case, we applied a simple threshold of $t = 0.1s$ to detect excessively long LDAP latencies. The result is shown in Figure 5.10, in which the edge between the LDAP pre-op and post-op plugins indicates a latency greater than $0.1s$. When making these timing measurements, we only compare times within one server machine. Despite time synchronization infrastructure, machines within datacenters exhibit enough clock skew to prevent cross-machine time comparisons. In our work, we only calculate latencies by comparing timestamps from a single network location.

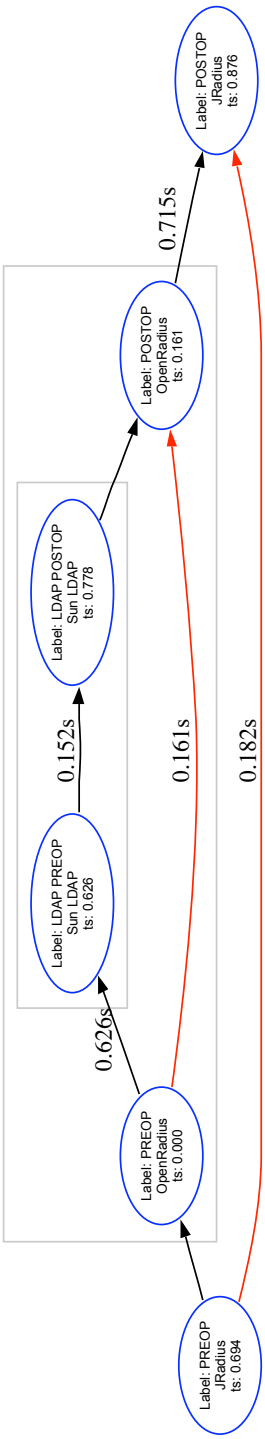


Figure 5.10: Root cause test 5: Slow LDAP performance. This can easily be detected by examining the edge labeled '0.152s'. Note that this latency measurement is accurate, since the LDAP PREOP and POSTOP events occurred on the same machine, and so clock drift is not a problem.

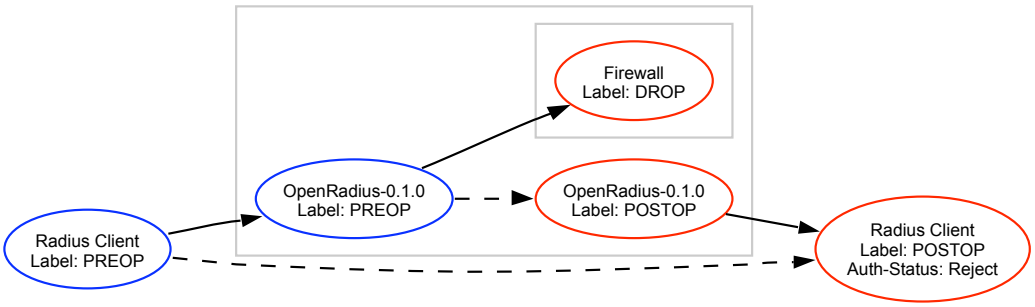


Figure 5.11: Root cause test 6: Firewall Interference. In this figure the firewall was *not* instrumented with X-Trace, and so the firewall report was emulated.

Root Cause Test 6: Firewall Interference Testing for the presence of interstitial firewalls between 802.1X components is similar to identifying poor RADIUS to LDAP connectivity, since in both cases we must either directly instrument the network layer to identify firewall interference, or infer that interference from the application layer. In the firewall case, inferring the firewall from the application layer is more reasonable, since we can extract logs from the firewall itself and correlate recorded packet drops with any partial X-Trace paths. In the poor connectivity case, the routers and switches along the path did not record packet drops, and the state of the TCP protocol was confined to the transport processes in both ends of the connection.

Figure 5.11 illustrates a trace that includes a report from the firewall that a packet drop was performed. In this experiment we were not able to add X-Trace support to the firewall, and so we emulated the report that the firewall would have produced. While direct instrumentation of middleboxes with X-Trace would produce a similar trace, this experiment demonstrates the utility of introducing trace elements extracted from other sources, in this case from the firewall logs. We hope that as an open standard, X-Trace instrumentation will

make its way into middleboxes and network devices, or even intrusion detection software such as the open-source BRO system[97]. Already we have seen X-Trace's adoption into three different Remote Procedure Call (RPC) middleware layers: Facebook's Thrift multi-protocol middleware, the lib-async library, and the Hadoop-Map/Reduce RPC system. The ease of introducing X-Trace into these packages gives us hope that some middleware appliances will include X-Trace as well.

5.6.4 Feasibility

Web 2.0 datacenter applications are in a state of constant flux, and new middleware platforms, languages (e.g., Ruby), and software components are being introduced to large-scale distributed systems. For this reason, we consider these rapidly changing environments beneficial for introducing end-to-end application trace generation and collection to commercial systems. In the case of 802.1X, there is an industry consortium, the OpenSEA Alliance [92], centered around standardizing cross-platform supplicants. Thus we would like to influence this effort to include datapath trace functionality into that standard.

In general, adding the X-Trace framework to the software running in the network and annotating protocol exchanges with X-Trace metadata incurs overhead in several ways. First, the process of changing the software takes some time. To modify the open-source RADIUS client, server, and LDAP servers took approximately two student-days in total. Since then, we have improved the interface to X-Trace to make this process easier. Second, imposing trace operations on the datapath increases the end-to-end latency, which decreases total throughput. The additional processing required to propagate and log the trace data

from the X-Trace API was minimal, amounting to about 30 microseconds per operation (meaning each component could theoretically log about 30,000 events per thread per second on a dual-core 3 GHz server. In terms of network overhead, each network message includes between 9 and 30 bytes of overhead. However, given that 802.1X is a control-plane protocol that is infrequently invoked (depending on the configuration, once every 10 to 60 minutes per client), this additional overhead is minimal.

The most significant source of overhead was collecting the trace records from each of the nodes. Collecting these records is an out-of-band operation, so we batched the reports and applied compression to reduce the load placed on the network. Using the code that we instrumented, we experienced a compression factor of approximately 19:1, meaning that a typical 802.1X exchange involving a client, RADIUS server, and one LDAP server would generate 30 bytes of trace data per node for each 802.1X authentication operation. We can predict the amount of trace data that would be generated in a large-scale deployment of X-Trace over 802.1X by utilizing the CRAWDAD dataset collected from Dartmouth [77]. Based on the period of highest demand during that publically available trace (350 clients per access point per hour) [59], this would result in 10.5 KBytes of trace data per hour per access point.

5.6.5 Limitations

The ability for X-Trace to indicate underlying network failures is only possible if those failures manifest themselves in the traces. In this chapter, we have focused on inferring underlying failures from application traces. As such, there are network phenomenon that are difficult or impossible to detect (such as the effect of packet loss on TCP flows, or

the presence of unmodified middleboxes). In the next chapter, we directly instrument the network itself, and present a methodology for leveraging that data to identify performance bottlenecks.

5.7 Conclusions

We have found that by examining end-to-end application traces from the 802.1X protocol, we were able to detect and diagnose both correctness and performance faults in underlying network components. This is a good alternative to more traditional network monitoring approaches for environments where access to the network is limited, including virtualized datacenters, as well as differing administrative boundaries between network operators and application developers. We show that our approach is feasible, and are actively working to extend the set of failures and their root causes that we can detect. In the next chapter, we extend our analysis technique to identify performance failures in distributed systems. We carry out this analysis both in dedicated server platforms, as well as shared "virtual" datacenters which potentially exhibit higher performance variance than dedicated environments. Reasoning about the expected performance of applications in these environments is key to making effective use of them as an on-demand application deployment resource.

Chapter 6

TraceOn: Scalable trace graph traversal

So far in this dissertation we have presented X-Trace, a cross-layer, cross application trace tool. We have presented its design and implementation, and evaluated that design in a variety of real-world scenarios. In this chapter, we present scalability improvements to X-Trace to enable users to process large-scale traces with millions of elements in them. We begin by motivating the need for such scalability, and then we describe the TraceOn design and implementation. Lastly, we evaluate our design against a deployed application in a virtual datacenter.

6.1 Motivation

Our experience has shown that tracing is a powerful way to gain visibility into the actions and execution of distributed system. As systems grow increasingly large, this

visibility is critical to building reliable and dependable software. By utilizing the output of X-Trace, network operators and software developers can gain insight into the presence of, and hopefully the cause of, unexpected behavior and erroneous conditions. As we apply tracing to larger and larger problems, the size of the trace graph increases, leading to longer processing time and larger resource requirements for processing those graphs. For smaller traces, building the tree in memory for each ad-hoc query has sufficed. The current approach of in-core trace reconstruction, based on the algorithm developed by Rodrigo Fonseca, has been successful for these cases, but as our ability to peer into larger systems becomes more common, we will need to scale to large traces. To analyze long-term data, or data from complex operations, we need a more scalable solution.

The issue of scalability arises from two main sources. First, large traces can result from big tasks, such as bulk database loads, web service initialization, or Hadoop Map/Reduce jobs. Second, traces that are the result of many smaller tasks that are conceptually part of the same operation (i.e., measuring S3 performance on EC2 over a two week period) can grow very large as well. While partitioning graph reconstruction across many nodes (e.g., with Map/Reduce) might be a viable alternative, in this work, we describe an approach for single-system graph reconstruction. This single-system will likely benefit multi-system approaches as well.

In addition to scalability limitations, the current approach to trace analysis is carried out at the improper level of abstraction. Users wishing to write ad-hoc queries against an X-Trace dataset must understand the format of reports, must extract components of those reports to build graph vertices, and connect those vertices with edges. These graph

reconstruction procedures must be written by each X-Trace user, which leads to bugs and wasted effort. Graph reconstruction must be performed for each ad-hoc query, and so the cost of that construction is not amortized across multiple queries.

To address these challenges, and to provide the X-Trace developer with a simpler, easier to use, and more efficient interface to applying ad-hoc queries directly to the trace graph, we present *TraceOn*. TraceOn is an API, runtime software system, and set of algorithms that lets users access and analyze X-Trace graphs without having to refer to low level Report constructs. The TraceOn API takes the form of an iterator into the graph that adheres to two conditions. First, the iterator will visit either the next, or previous, event that is causally related to the starting point. In the case of concurrent executions, the user will visit one entire causal path before visiting the other branches of that concurrent region. Second, the iterator will only traverse "real edges", which represent direct causality. Indirect causal edges, or "virtual" edges, may still be visited by the user, but the API and TraceOn runtime are not optimized for those traversals. We will show that the real-edge iteration property is appropriate for a wide variety of queries. TraceOn pre-processes traces so that iteration is made as efficient as possible. As we will show in this chapter, appropriate preprocessing is essential for supporting high-performance and scalable queries on the underlying data.

The main obstacle to accomplishing scalable trace processing on a single node is the need to externalize, or move "out of the core", the analysis algorithms. This process involves making use of disk storage, in addition to physical memory, such that the intermediate storage needed to carry out the algorithm does not exceed the amount of memory allocated to the process, while at the same time minimizing I/O accesses to the disk, which

is significantly slower than main memory. Balancing these tradeoffs requires careful attention to balancing the tradeoffs between the CPU, memory, the disk, and the amount of time the algorithm takes.

The need to analyze and process very large graphs is not novel. Large web search engines must process what are in effect graphs with billions of vertices. Algorithms that process these graphs, such as PageRank[95], must be externalized due to their enormous size. PageRank was designed for graphs much larger than those considered in this chapter. The approach for PageRank-sized graphs have relied on very large clusters of computers coupled with special programming paradigms such as *Map/Reduce* to handle the scalability. These approaches, some of which are public, and some are industrial trade secrets, do not satisfy our needs, since we require a system that a single user of X-Trace could use without requiring a large computational cluster. In fact, since we only process graphs produced by X-Trace, we can take advantage of their structure and our knowledge of the semantics of the traces to customize and tailor our analysis algorithms to be more efficient and scalable for the data they will process.

We evaluate the efficacy of our approach three ways. First, we show that the API exported by TraceOn is expressive enough to capture a common use case of X-Trace, namely identifying changes in the performance characteristics of a subset of the distributed system over time. Second, we evaluate each phase of TraceOn using micro benchmarks that determine their contribution to the entire end-to-end processing time. Third, we compare our approach to in-memory approaches that have been used with success for smaller datasets.

Our evaluation is carried out in the context of a new distributed application that we wrote and deployed in the Amazon Elastic Compute Cloud (EC2) virtual datacenter. EC2 is a so-called "hardware as a service" system in which users can rent servers by the hour, with pricing based on the capabilities of those servers. Using EC2 as a base, we built an indexing and searching site for over 120,000 National Science Foundation (NSF) grants awarded between 1990 and 2003. The NSF award database ran live on EC2 for seventeen days, between April 6, 2008 and April 22, 2008, resulting in 58,750 X-Trace graphs totaling 12,053,884 unique events.

In this chapter, we first present the design of TraceOn, the algorithms used to process trace graphs, and the external implementations of those algorithms. We also present a paging and buffering memory manager we designed to efficiently externalize the stack structure used to process the topological sorts needed by TraceOn. Then, in Section 6.5, we present the design of the NSF award site, and then in Section 6.6, we evaluate the TraceOn data flow along the three axes of expressibility, microbenchmarks, and comparison to previous approaches. Lastly, we conclude by highlighting our experiences with TraceOn, and present extensions to TraceOn that would improve its usefulness and ease of use.

To begin, in the next section we describe the high-level design of TraceOn.

6.2 Design

In the previous section, we outlined the need for the TraceOn system. TraceOn must provide an easy-to-use API that users can use to traverse trace graphs without needing to manipulate the low-level structure of individual reports. We begin this section by

presenting the in-memory graph reconstruction algorithm. We then discuss the programming model as experienced by the user. We then describe the operational model, which puts TraceOn in perspective with the rest of the X-Trace infrastructure. This discussion highlights how data is introduced to TraceOn, how that data is pre-processed to improve efficiency, persisted to disk in optimized form, and finally how queries are executed against the data. Lastly, the in-memory version of the X-Trace graph reconstruction procedure is presented. In the next section, we describe how we externalize this procedure.

6.2.1 Graph reconstruction procedure

In this section, we present the algorithm used to reconstruct X-Trace task graphs from individual reports. The foundation of TraceOn is this procedure, which recovers the execution graph from the set of individual X-Trace Reports. Instances of this algorithm, developed by Fonseca[49], have been implemented in Perl, Java, and Ruby. These implementations are in-core, relying exclusively on main memory for their storage requirements. This algorithm takes as input the set of reports making up a unique task. The output is the real-edge ordering of those reports It works as follows:

1. Load the set of reports making up task T into memory
2. Store these reports in a Dictionary data structure indexed on the report's operation id as the primary key
3. For each element e of the dictionary, identify the set of e's parents by locating "Edge" keys in e's report.
4. For each parent of e, store the operation id of e in a child field.

5. Identify the root of the tree, which is either identified by operation id 0 or is the only report without any incoming "Edge" keys.

6. From the root, perform a depth first search, considering each node's child edges in arbitrary order. Keep track of the finishing time of each node in a dictionary data structure, which is the time that each node is visited according to the depth first search order.

7. Starting again at the root, perform a second depth first search. This time, consider each node's child edges in decreasing order of their finish times, according to the first DFS.

8. Edges in the second DFS traversal are ordered according to the real-edge ordering, whereas unvisited edges are virtual.

6.2.2 Programming model

The programming model as experienced by a TraceOn user has two main facets. First, the user must initialize TraceOn by defining the scope of data to be processed. We call this scope a dataset, and its definition an "input filter." TraceOn operates in units of datasets, and the input filter defines what data goes into those datasets. The most basic input filter is simply a list of Task Ids. Another input filter is based on time, looking for any tasks active during a provided time range. An interesting use case of the time-based input filter would be analyzing all traces active during the past 10 minutes, or all traces active shortly before a system crash or network event.

The second facet of the TraceOn programming model is the iterator it exports to

each of its datasets. The user submits queries to TraceOn using this iterator. Query processing is decoupled in time from initialization, and once the dataset is initialized, multiple queries can be processed against that dataset. TraceOn's iterator adheres to two properties: the iterator property, and the real-edge property. The iterator property means that when it is first created, it will point to the very first event in the system, that is, the event that is not caused by any other events. Each subsequent access of the iterator causes it to refer to the causally next event in the trace. The real-edge property means that edges in the trace are visited in the direct causal ordering. Figure 6.1 shows the HTTP example trace from Chapter 3. The highlighted edges are part of the real-edge path.

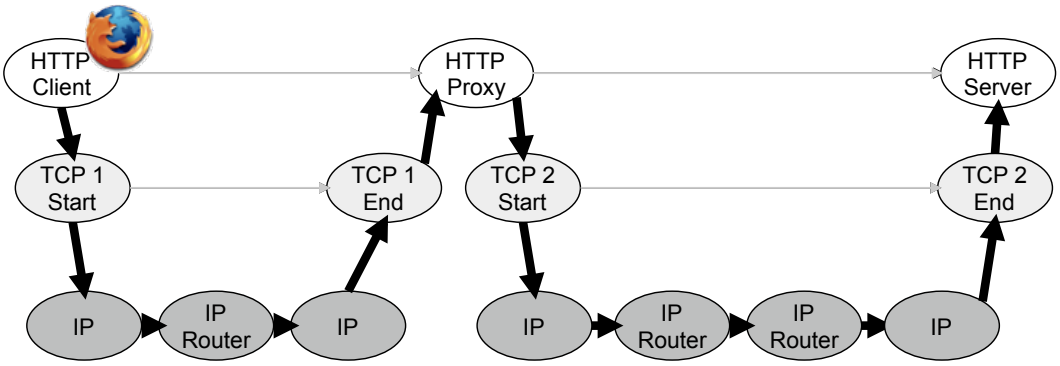


Figure 6.1: The "real edge" iterator visits edges along the direct causal route from source to destination. In this HTTP example, the emphasized edges are part of the real edge path.

6.2.3 Operational model

During its operation, TraceOn must support the introduction and definition of new datasets from input filters provided by the users. Additionally, it must export real-edge iterators when queries are submitted against those datasets. When defining new datasets, the reports needed for that definition are obtained directly from the X-Trace backend using

its web service interface. Once the reports are downloaded into TraceOn, it pre-processes them using what we call the TETRA algorithm, presented in the next section, resulting in an optimized on-disk database. When a client connects to TraceOn to submit a query, a handle to that on-disk database is accessed, and the iterator can return reports in the appropriate order. TraceOn is implemented as a multi-threaded server, which allows it to process more than one dataset at a time.

6.2.4 Graph reconstruction procedure

The foundation of TraceOn is the X-Trace graph reconstruction procedure, which recovers the execution graph from the set of individual X-Trace Reports. Instances of this algorithm, developed by Fonseca[49], have been implemented in Perl, Java, and Ruby. These implementations are in-core, relying exclusively on main memory for their storage requirements. This algorithm takes as input the set of reports making up a unique task. The output is the real-edge ordering of those reports It works as follows:

1. Load the set of reports making up task T into memory
2. Store these reports in a Dictionary data structure indexed on the report's operation id as the primary key
3. For each element e of the dictionary, identify the set of e's parents by locating "Edge" keys in e's report.
4. For each parent of e, store the operation id of e in a child field.
5. Identify the root of the tree, which is either identified by operation id 0 or is the only

report without any incoming "Edge" keys.

6. From the root, perform a depth first search, considering each node's child edges in arbitrary order. Keep track of the finishing time of each node in a dictionary data structure, which is the time that each node is visited according to the depth first search order.

7. Starting again at the root, perform a second depth first search. This time, consider each node's child edges in decreasing order of their finish times, according to the first DFS.

8. Edges in the second DFS traversal are ordered according to the real-edge ordering, whereas unvisited edges are virtual.

6.3 TETRA: TraceOn External Trace Reconstruction Algorithm

We have introduced several motivations for abstracting queries from traces from their underlying manifestation as individual reports. The API presented in the last section represents a basic primitive that is useful in supporting sophisticated query processing clients of TraceOn. In this section, we present the design of TETRA, or the *TraceOn External Trace Reconstruction Algorithm*. The input to TETRA is the standard trace output file generated by the X-Trace backend. TETRA processes this input file and generates a pre-processed version of the trace that is persisted on disk in a layout optimized for our exported API.

We will first present an overview of the flow of data through TETRA, and then

present each stage of its operation in detail. In the next section, we discuss TETRA's software implementation, and after that, we evaluate it against several common query workloads.

6.3.1 Dataflow overview

Figure 6.2 outlines the flow of data through TETRA. Data enters the system through one or more input trace files, which are generated by the X-Trace backend. The input processing stage (Section 6.3.2) linearly scans the input file, extracting individual trace reports. During this scan, three operations are performed for each report. First, an "offset index" is built (Section 6.3.3) that keeps track of the location of each report within the input file, indexed by its operation id. We do this to efficiently find the original reports and store them in the database at the end of TETRA processing. Second, each report is examined to determine if it is a "root" (Section 6.3.4). Roots are graph vertices that have no incoming edges. Third, each incoming edge to each report is stored in an unsorted "child map" (Section 6.3.5).

The child map, which is an adjacency list representation of the original graph, is used to determine the set of causal children for each event in the trace. X-Trace reports encode the set of causal parents of a given event, but not its causal children. To reconstruct these children, we record the set of all parent edges in the trace, which are edges connecting a report to one of its parents in the child map table. From this table, we build the `listChildren()` function (Section 6.3.5). When provided with an event's operation id, `listChildren()` will use the child map to return a list of all causal children of that event. This function serves as the primary input to the two topological sort operations, which we

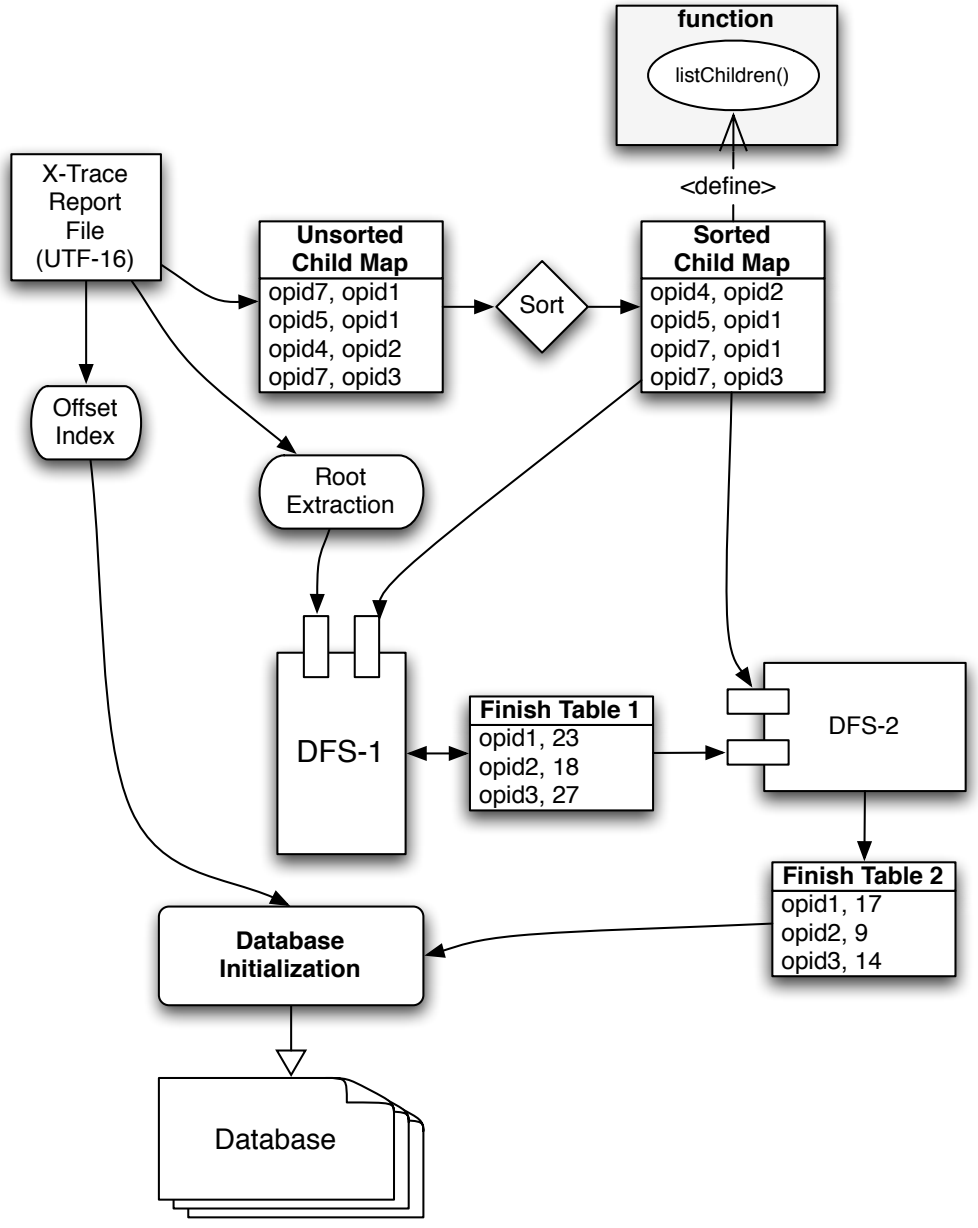


Figure 6.2: Dataflow description of the TETRA algorithm.

now describe.

Two topological sort operations are performed on the graph by implementing two depth first search, or DFS, operations (Sections 6.3.6 and 6.3.7). When the first depth first search visits a node in the graph, it first visits all of its children in no particular order, before finally visiting the node itself. The time that each node is visited is stored in *Finish Table 1*. The second DFS operation likewise visits each node's children before the node itself. However, during the second DFS the children are visited in the reverse order of their finishing times from the first DFS, as captured in *Finish Table 1*. This is the only difference between DFS-1 and DFS-2. The order that nodes are visited during DFS-2 is stored in *Finish Table 2*.

A database is created by laying out X-Trace reports on disk according to their topological sort order, which is simply the reverse order of their finishing times, as captured by *Finish Table 2*. To aid in finding the original reports, the offset index is used to avoid repeated linear scans of the input file. This database can also encode a special header record that includes metadata about the dataset. The end result is a disk-based, persisted database in which graph nodes are laid out according to their eventual access order when evaluating X-Trace queries. We now explore each phase of TETRA in detail. Then, in the next section, we will describe its implementation in software.

6.3.2 Input processing

The first phase of TETRA is extracting X-Trace reports from the input files provided by the backend server. These files are standard text files, in which reports are stored sequentially, separated by one or more empty lines. Each report is a variable length record

of key-value pairs. To process the input file, the input processor must scan the file, extracting each report. For each extracted report, its X-Trace metadata and the list of parent pointers must be recovered. Because the reports are not aligned within the file, we use a regular expression, R_{report} to find the start and end of each report. Within that region, we use a regular expression $R_{metadata}$ to extract the metadata, and another R_{parent} to find each parent pointer. Thus, the number of regular expression evaluations for a trace representing the graph $G = (E,V)$ is:

Expression	Num. Evaluations
R_{report}	$O(\|V\|)$
$R_{metadata}$	$O(\|V\|)$
R_{parent}	$O(\|E\|)$

For the regular expression classes to work, the input file must be in the UTF-16[60] character set, which is a precondition for TraceOn. Common system utilities can easily convert text files into UTF-16.

6.3.3 Offset index construction

For each report in the input processing phase, the R_{report} regular expression identifies its starting and ending offset within the input file. We cache these offsets into an offset index, so that we can easily extract specific reports from the input file during later phases of TETRA. The index is simply a binary file of records of the form shown in Figure 6.3. Before using this index, we first sort it, which is an $O(N \log N)$ running time operation. After sorting, we can search for records in it in $O(\log N)$ time.

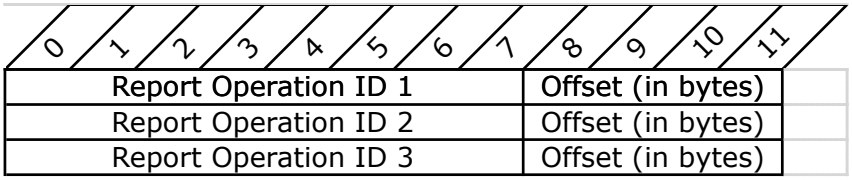


Figure 6.3: Field format for the offset index. Each record consists of an 8-byte operation id followed by a 4-byte length representing the offset into the original input file.

6.3.4 Root extraction

TraceOn operates at the granularity of a *Dataset*, which might consist of multiple individual traces, each with its own unique TaskID. Thus, a single dataset might consist of a forest of individual trace graphs. To perform the depth first searches later in the TETRA algorithm, we must start the roots of each of these graphs. During input trace processing, we look for nodes that do not have any causal parent edges (i.e., those reports that fail to match the R_{parent} regular expression). The roots are stored in a simple in-memory array.

6.3.5 Child map creation and `listChildren()`

Each X-Trace report, and thus each vertex in our trace graph, stores a list of its causal parents. However, to perform the topological sort operation, we need each node's list of children. Inverting the parent pointers into child pointers requires building an inverse index, which we call the child map. To build the child map, we store each parent pointer sequentially in a file in the same order they appear in the input file. For a node N with M parents, we construct M tuples of the form $< OPID_P, OPID_N >$, for each parent P of N. For a trace graph $G = (E,V)$, the final child map will consist of $|E|$ tuples, one for each edge in the graph.

Once the graph is formed, we then sort it on the parent operation id column. For

typical input an efficient comparison sort takes $O(N \log N)$ time. With the sorted child map, we define a function `listChildren()` that, given a node's operation id, returns the list of operation ids corresponding to that node's causal children. To find the children of a node with operation id x, the function works as follows. First, we perform a binary search on the sorted child map, looking for the first x in the first column, the parent's operation id. If such an entry is found, we then scan forward through the child map, looking for records with the parent operation id of x. For all such records, we accumulate entries in the second column that correspond to the children's operation ids. When we have exhausted all entries with parent ids of x, we return the accumulated list of children.

The `listChildren()` function is critical to both DFS searches, and its performance must be high, since for a trace graph $G = (E,V)$, it is called $O(2 * |E| * |V|)$ times. Its memory requirements are rather modest ($O(|E|)$), meaning that for a dataset with 2 million events, approximately 2.4 million entries would be stored in the table. Since each entry is 16 bytes, that would require 36.6 MB of memory.

6.3.6 First topological sort and Finish Table 1

Recall that the trace graph consists of both *real* and *virtual* edges. Real edges represent direct causal effects, while *virtual* edges represent indirect causal effects. The in-memory graph reconstruction algorithm, presented earlier in this chapter, is a twist on the standard topological sort procedure, which is extended to discriminate between these two edge types. It does this by choosing a particular topological sort order of the vertices in the graph (every directed, acyclic graph (DAG) has at least one topological sort, but may have more).

The first topological sort is shown in Figure 1.

This procedure begins by initializing the `visitTime` variable [line 2], which holds a running tally of the virtual time used during the depth first search. This virtual time will be used to keep track of when each node in the graph has been visited. Next, the Finish Time data structure and stack is initialized [lines 3-8] with each of the roots in the dataset. We assume that the roots are members of disjoint graphs. The rest of the procedure [lines 9-33] is the main body of the traversal, in which the node on the top of the stack is examined, and a subset of its children are pushed back onto the stack. When a node is examined that has no unvisited children, then we "close" that node and record its finish time in the `finishTime` table. The body of the procedure [lines 10-31] contain two main cases unique to traversing X-Trace graphs.

The first case [line 11] handles nodes that have not yet been visited along a given depth first search path. The node's children are each examined only if they have not been "closed" yet. This determination is made by consulting the Finish Table [line 14]. Those nodes that are still open (i.e., not in the Finish Table) are pushed onto the stack for later processing. If any open nodes are found [lines 19-21], then we set the node on the top of the stack's `visitTime` to the current `visitTime`. If not [lines 23-24], then we close the node on the top of the stack, which inserts the node and the current `visitTime` into the Finish Table.

The second case [lines 26-28] handles nodes that have not yet been visited, but are not part of the current depth first search path (i.e., the algorithm is backtracking to an earlier part of the graph). By setting the top of the stack's `visitTime` to the current `visitTime`

Algorithm 1 DFS-1 procedure

```
1:  procedure DFS-1(RootList)
2:      visitTime ← 0
3:      S ← new ExternalStack()
4:      FinishTime ← new FinishTime()
5:      for all r ∈ RootList do                ▷ Initialize the stack with the graph roots
6:          S.push(<r, visitTime, null>)
7:          visitTime ← visitTime + 1
8:      end for
9:      while S.size() > 0 do                  ▷ Main body of the DFS
10:         <opid, start, finish>← S.top()
11:         if start ≠ null ∧ finish = null then          ▷ Unvisited node case
12:             hasOpenChildren ← False
13:             for all child ∈ finishTime.listChildren(opid) do
14:                 if finishTime.isOpen(child) then
15:                     S.push(<child, null, null>)
16:                     hasOpenChildren ← True
17:                 end if
18:             end for
```

Algorithm 2 DFS-1 procedure (continued)

```
19:            if hasOpenChildren = True then
20:                <topopid, start, finish>← S.pop()
21:                S.push(<topopid, visitTime, null>)
22:            else
23:                finishTime.closeNode(opid, visitTime)
24:                S.pop()
25:            end if
26:        else if start = null ∧ finish = null then          ▷ Begin back tracking
27:            S.pop()
28:            S.push(<opid, visitTime, null>)
29:        end if
30:        visitTime ← visitTime + 1
31:    end while
32:    S.destroy()
33: end procedure
```

value, that node becomes the root of the new depth first search path. Thus, subsequent iterations of the algorithm will fall into the first case.

Lastly, once every node in the graph has been assigned a finish time and places into the Finish Table, we destroy the stack [line 32], which reclaims its resources. The end result is a Finish Table whose finishing times represent a topological sort of the graph. However, each node's children were examined in arbitrary order. The key to appropriately assigning each edge with a "real" or "virtual" label is examining those children in *decreasing order of their finishing times from a previous topological sort*. To do this, we use the values in the Finish Table (which we now call "Finish Table 1") to perform a second topological sort.

6.3.7 Second topological sort and Finish Table 2

The the second topological sort is responsible for assigning "real" and "virtual" labels to each edge of the trace graph. Real edges are those that are visited in the second DFS traversal. This second traversal is identical to the first one, described in the previous section, with the following difference: the children of each node must be visited in the *reverse order of their finishing times as recorded in Finish Table 1*. For the second topological sort, we use the same algorithm as the first sort, but we replace lines 12 and 13 with the following code segment:

In this algorithm, the `sort()` function returns the list of children in *increasing* order of their DFS-1 finishing times, so that when those children are pushed onto the stack, they will be visited in *decreasing* order (since items on the stack are visited in a "Last-In, First-Out" ordering).

Algorithm 3 DFS-2. This snippet replaces lines 12-13 in DFS-1

```
hasOpenChildren ← False

children ← finishTime.listChildren(opid)

sortedChildren ← sort(children)

for all child ∈ sortedChildren do

    (rest of code from previous section)

end for
```

6.3.8 Database initialization and layout

The final stage of pre-processing is to persist the original X-Trace reports onto the disk in real-edge visit order. The reports are extracted from the source file, using the offset index to quickly identify where in the source file a report is located. The order they are committed to the disk is given by the *Finish Table 2*, computed in the second topological sort (DFS-2). For the current implementation of TraceOn, the reports are organized on disk in a binary file of report records. Each report record is simply a tuple of the form <opid,report>, where *opid* is a `long`-typed variable indicating the operation id of the given report, which is stored in UTF-8 format. Therefore, the real-edge report iterator is implemented by accessing reports in the order in which they appear in the database file. The binary data file format allows for future expansion, since metadata about the dataset is easily stored in header of the database file.

6.4 Implementation

Now that we have described the programming and operational models, as well as the design of the TETRA graph reconstruction algorithm, we now describe these components' implementations as a Java software artifact. In this section, we highlight those parts of the implementation that required special attention or care. The first component we consider is the input processing stage.

Because of its possibly very large size, ranging from hundreds of megabytes to several gigabytes, it is not always possible to load the input file into memory to perform the regular expression processing. Thus, we made use of a streaming interface to the file, making use of the operating system's virtual memory subsystem. The input file is first mapped into TraceOn's address space with `mmap()`. In the Java language, this is done by the `ByteBuffer` class. This byte buffer can then be passed to the set of `Pattern` and `Matcher` classes that perform the regular expression matching. As the regular expressions work through the file, the underlying operating system automatically buffers and pages segments of the file into memory as needed. The memory used for this operation is allocated outside of the Java Virtual Machine, and under the control of the host. As we discovered during our implementation of the external stack datastructure, even a modest amount of buffering drastically improves I/O performance in Java. Thus, the external memory requirements used by the `mmap()` feature are minimal, meaning that even if several concurrent TraceOn instances are running on the same machine, memory contention on the input processing phase should be minimal.

Like the input file processing stage, the external stack data structure must sup-

port on-demand paging and buffering to minimize the number of I/O operations required for its operation. We implemented this data structure using the `RandomAccessFile` Java construct. While this data structure allows for arbitrary access within a given file, those accesses are not buffered. This results in substandard performance.

We wrapped our external stack data structure within an interface that conceptually divides the file into multiple, fixed-length pages. Whenever a byte within a page is accessed, a check is made as to whether that page is in the memory cache, or whether it is on disk. Requests to in-memory pages are satisfied immediately. When a region of an on-disk page is accessed, the page currently in memory is written back to the disk, and the requested on-disk page is brought into memory. Unlike typical virtual memory systems that support caches with multiple in-memory pages, our stack only allows for one page to be in memory at a time. Since the stack is only ever accessed from the top, having more than one page in memory is unnecessary for our application.

Lastly, we implemented the on-disk database by using the *DataInput* and *DataOutput* interfaces to a *FileInputStream* and *FileOutputStream*. As a result, we can directly read and write both the `long` and `String` data types to the disk. This makes accessing adjacent reports very simple and efficient.

Before we present an evaluation of our software implementation, we first describe the application used to generate input data for TraceOn.

6.5 Demonstration Application: an NSF award search engine

To evaluate our design, we deployed a web-based application into the Amazon EC2 virtual data center environment. This application is a search engine for the National Science Foundation's grants awarded between 1990 and 2003, inclusive. During this time, over 120,000 awards were given. Figure 6.4 shows the results of searching for awards with the text "particle" in them. The search engine allows the user to search based on the principle investigator, the title, the award number, the award amount, and in the example shown in the figure, anywhere in the body of the award. This search screen shows the title of each award, a unique document identifier, and a score, which represents how well the document conforms to the search terms given. Once the user clicks on a particular award, they are brought to the award grant display screen, shown in Figure 6.5. This screen displays every field in the award abstract.

The design of this application is shown in Figure 6.6. It is based on the Lucene open-source document search engine, and the source data is kept in Amazon's Simple Storage Service (S3). Interfacing to S3 is a set of Java server processes that export a Facebook Thrift interface. Thrift is an RPC-based middleware system that lets different pieces of software, possibly written in different languages, to interface with each other. Interfacing to the Java version of Lucene is another set of servers, also outfitted with Thrift. The first tier of the system is a Java servlet server which hosts the set of servlets responsible for processing user queries.

We collect traces by instrumenting Facebook's Thrift with X-Trace. By doing this, we can determine the amount of time spent in each part of the indexing and search-

Figure 6.4: The NSF Award search application results screen. Users can search based on the P.I., the title, the award number, the award amount, and a free text search which matches any field in the award. The right column indicates the score, or how well the document matches the query.

Figure 6.5: The NSF Award grant display screen. This screen shows the entire NSF award.

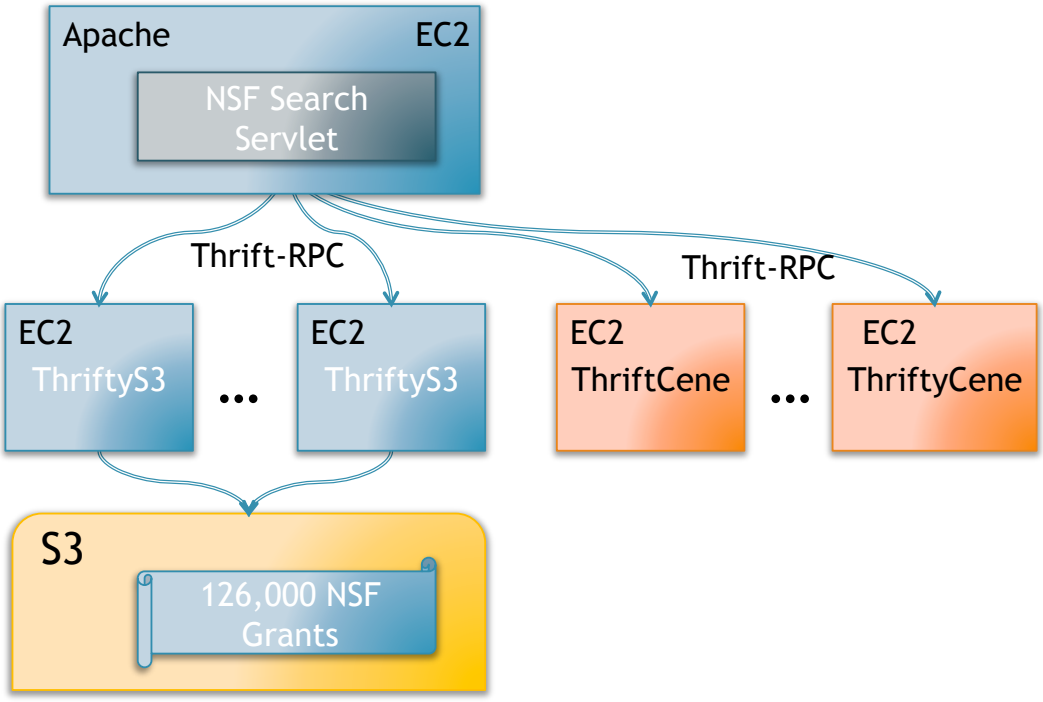


Figure 6.6: An overview of the NSF award search engine, hosted in the Amazon EC2 virtual datacenter environment. The grant source data is stored in S3, and is accessed by a horizontally scaled set of Thrift-enabled server processes, called ThriftS3. The index servers (ThriftyCene) store their indices on local disks. Trying ThriftyS3 and ThriftyCene together is the NSF Search Servlet, a Java-based servlet that powers the user interface.

ing system. We also added instrumentation points around the S3 storage calls, giving us insight into the performance of S3 over time. The NSF award database ran live on EC2 for seventeen days, between April 6, 2008 and April 22, 2008. During this time, two client load generators, also located in EC2, requested documents from the index on average of 2.38 times per minute, resulting in 3,455 requests per day, or 58,749 requests during the lifetime of the experiment. This demand resulted in 58,749 X-Trace task graphs. We also traced the initialization of the award database, which resulted in a single task graph containing 1,323,668 events. Combined, the initialization and experimental workload resulted in 12,053,884 unique X-Trace events.

6.6 Evaluation

The previous sections have introduced the need for TraceOn, have presented its programming and operational models, and its realization as a software artifact. We have presented the NSF Award Search Engine, a realistic use case of a software service deployed in the EC2 virtual datacenter environment. In this chapter, we evaluate TraceOn using data from the EC2 deployment along three axes: expressiveness, per-component microbenchmarks, and in comparison to previously implemented, in-core implementations of the graph reconstruction algorithm. In the following evaluations, we applied TraceOn to a 1.3+ million event dataset representing the initialization of the NSF award search engine. We start by analyzing the performance of the input file processor.

6.6.1 Input file processing

One of the key limiting factors to TraceOn is its ability to read data out of the input files provided to it. Since TraceOn acts like a pipeline of individual processing steps, the input file processing performance dictates the ultimate scalability of the system as a whole. Figure 6.7 shows the performance characteristics of the input processing.

Hex	Offset	ChildMap	Reports / sec (Mean)	Std. Dev.	% efficiency
✗	✗	✗	32,376	8.01	100%
✔	✗	✗	30,276	10.58	93.5%
✔	✗	✔	29,400	95.39	90.8%
✔	✔	✗	21,209	7.02	65.5%
✔	✔	✔	20,715	60.32	64.0%

Figure 6.7: The TETRA import file processing phase performance. Extracting the root nodes, building the unsorted child map, and building the offset map into the original report file results in a processing efficiency approximately 36% below optimal (at 20,715 reports processed per second). To put that performance in context, the current X-Trace backend server can handle approximately 30,000 reports per second.

Recall that processing the input file requires three separate operations: edge extraction and root identification, offset index construction, and child map construction. In Figure 6.7, we evaluate the overhead of each of these steps individually, which allows us to evaluate their contribution to the overall system's overhead. The first row of the figure shows the report processing speed when none of the three required steps are performed. This means that TraceOn is simply reading the file from start to finish. We see that in this case, TraceOn was able to handle 32,376 reports per second. This experiment was performed on a MacBook Pro laptop with a 5,400 RPM disk drive, 1.5 GB of memory, and a 1.83 GHz Intel Core Duo processor. This performance result indicates the inherent I/O

performance of the underlying system.

The second row in the figure shows the performance of the root identification and edge extraction step. The net effect of this step is an approximately 7% drop in efficiency, to 30,276 reports. Child map construction results in a further 3% reduction. Building the offset index was the most expensive operation in this processing step, resulting in an almost 30% drop in efficiency. Putting together all three operations results in a system that can handle 20,715 reports per second, at an efficiency of 64.0%.

6.6.2 TETRA external stack

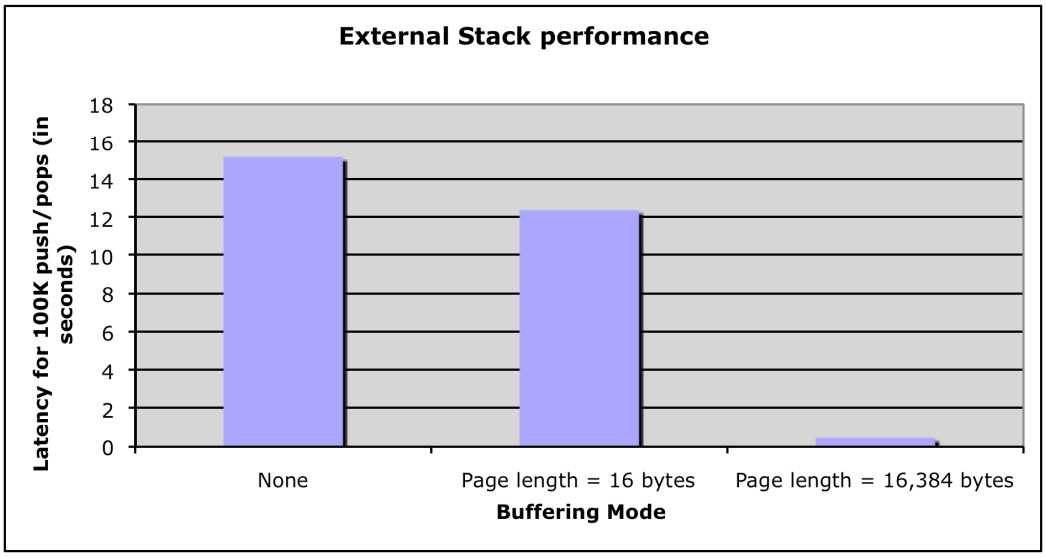


Figure 6.8: The TETRA external stack performance. Using Java's unmodified `RandomAccessFile` for the stack takes about 15 seconds to handle 100,000 `push()` operations, followed by 100,000 `pop()` operations. Adding a small amount of buffering reduces this time to about 12 seconds. Finally, adding a modest-sized, 16 KByte buffer reduces this considerably, to about 500 ms. Increasing the buffer size beyond 16 KBytes does not result in significant additional gain.

As we discussed previously, implementing the external stack as a paging, buffered data structure greatly improves performance. In Figure 6.8, we see that compared to the

baseline data structure, applying a 16 KByte buffer improves performance by approximately 30 times, from 15,000 milliseconds per 100,000 push/pop pairs to 500 ms. We take advantage of this increase in performance to improve the throughput of the two depth first search algorithms.

6.6.3 Depth first search graph iteration

	Mean latency	Std. Dev.	Memory requirement	Events/sec
DFS-1	36.57	0.08	92.86 MB	36,194
DFS-2	43.21	0.10		30,637

Figure 6.9: TETRA graph reconstruction relies on two depth first search evaluations of the graph. By making use of the paging and buffering external stack data structure, DFS-1 and DFS-2 complete in 36.57, and 43.21 seconds, respectively. Their total memory footprint is under 100 MB., meaning that multiple DFS operations can be executed concurrently if multi-core environments are available.

The two topological sorts are implemented using depth first search algorithms. These are in turn implemented directly using the external stack data structure. Figure 6.9 shows that the first DFS was completed in 36.57 seconds, or 36,194 reports per second. The second DFS, which is based on the output of the first DFS, completed in 43.21 seconds. The two DFS operations together used 92.86 MB of memory. This low memory requirement is ideal for multi-core environments, since each core would be able to operate on a separate dataset. On a system with 4 GB of memory, using datasets of the same size as our evaluation, over 32 cores could be kept busy without exceeding the memory, taking into account the operating system and JVM overheads.

6.6.4 Database construction

The last pre-processing step we consider is the construction of the database, whose performance characteristics are shown in Figure 6.10. Using the `DataOutput` based I/O stream, we were able to stream out the X-Trace reports in DFS-2 finish time order in 43.68 seconds, which is 30,304 events per second. The on-disk layout was 281 MB in size.

	Mean latency	Std. Dev.	On-disk size	Events/sec
BuildDatabase()	43.68	11.02	281 MB	30,304

Figure 6.10: Once the TETRA topological sorts are complete, the reports must be stored on-disk in real-edge visit order. This process took 43.68 seconds for the 1.3 million node dataset, resulting in an on-disk size of 281 MB.

When we put each of the steps together, we are able to pre-process the 1.3 million node dataset in 184.59 seconds, using less than 100 MB of memory at any one time.

6.6.5 Ad-hoc query evaluation

Once TraceOn has finished pre-processing a dataset, it is ready for query evaluation. Evaluating a query involves accessing the on-disk dataset, and visiting X-Trace reports in the real-edge ordering, which happens to be the same as the on-disk ordering. Figure 6.11 shows that for the large NSF award dataset, this takes about half a minute to complete (36.08 seconds, or 36,688 reports per second). The memory requirements for this iterator are minimal (less than 16 KBytes).

The result of the ad-hoc query is shown in Figure 6.12, which indicates the write latency to S3 during the initialization of the NSF award search engine. We ran a different

Experiment	Mean latency	Std. Dev.	Memory requirements	Events/sec
Plot s3 put latency	36.08	4.91	< 16 KB	36,688

Figure 6.11: The TETRA pre-processing step allows for rapid query evaluation against the optimized, on-disk trace layout. This query plots Amazon S3 `put` latency over time. Its source data, a 1.3 million event trace, is processed in just 36.08 seconds, using a small constant amount of memory. This enables the user to rapidly adjust and refine the offered queries based on observing the output of previous queries.

query, a CDF plot of that distribution, in the same amount of time, shown in Figure 6.13. This high level of responsiveness leads to a tool that is more user friendly, and allows for rapidly changing the analysis of the underlying data. For researchers and network operators, this rapid turnaround of queries opens up new possibilities for answering questions about the state of their networks, applications, and datacenters.

6.6.6 Comparison to in-core approaches

There are three in-core implementations of the graph reconstruction algorithm available to X-Trace developers. One is written in Perl, another in Java, and a third written in Ruby. We have experience applying these to X-Trace graphs of approximately 1.2 million events. These alternative implementations use a considerable amount of memory–approximately 9 GB–to capture the entire text of each report, as well as data structures representing the graph structure. When executed on a typical server with 3 GB of memory the algorithm had to rely on the operating system's virtual memory system to page memory to and from disk. This resulted in a condition called thrashing, and as a result the analysis took over 11 hours. Running the same tool against the same graph on a larger server with 15

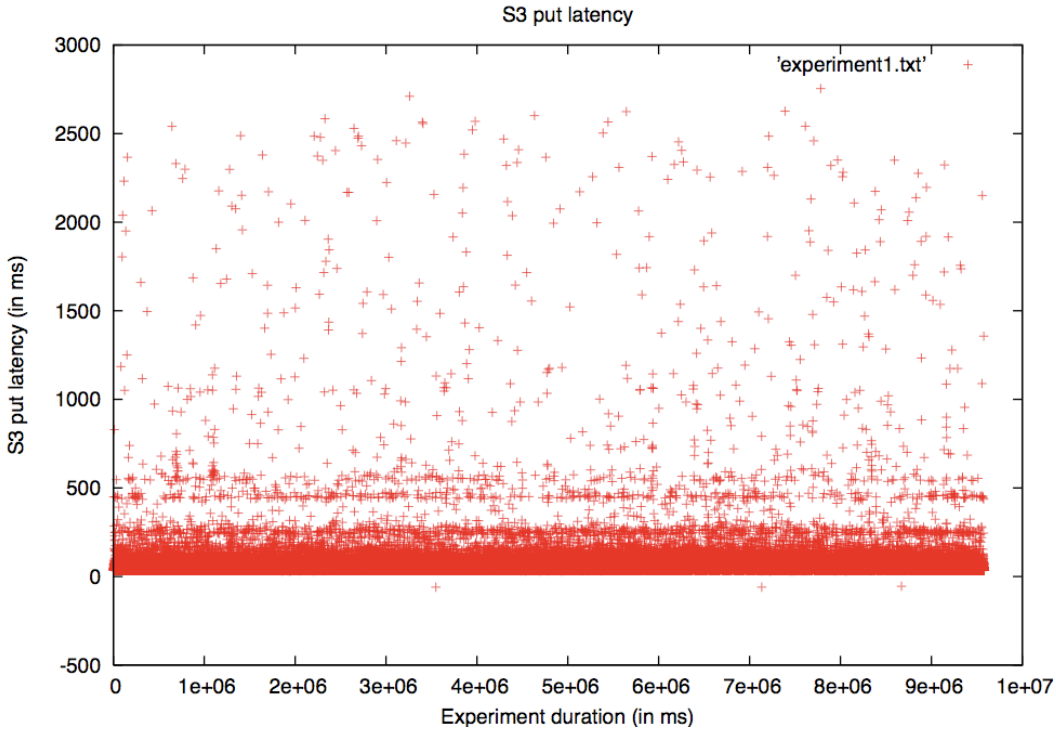


Figure 6.12: S3 put latency during the NSF Award database initialization. This figure was produced using the real-edge iterator programming interface exported by TraceOn.

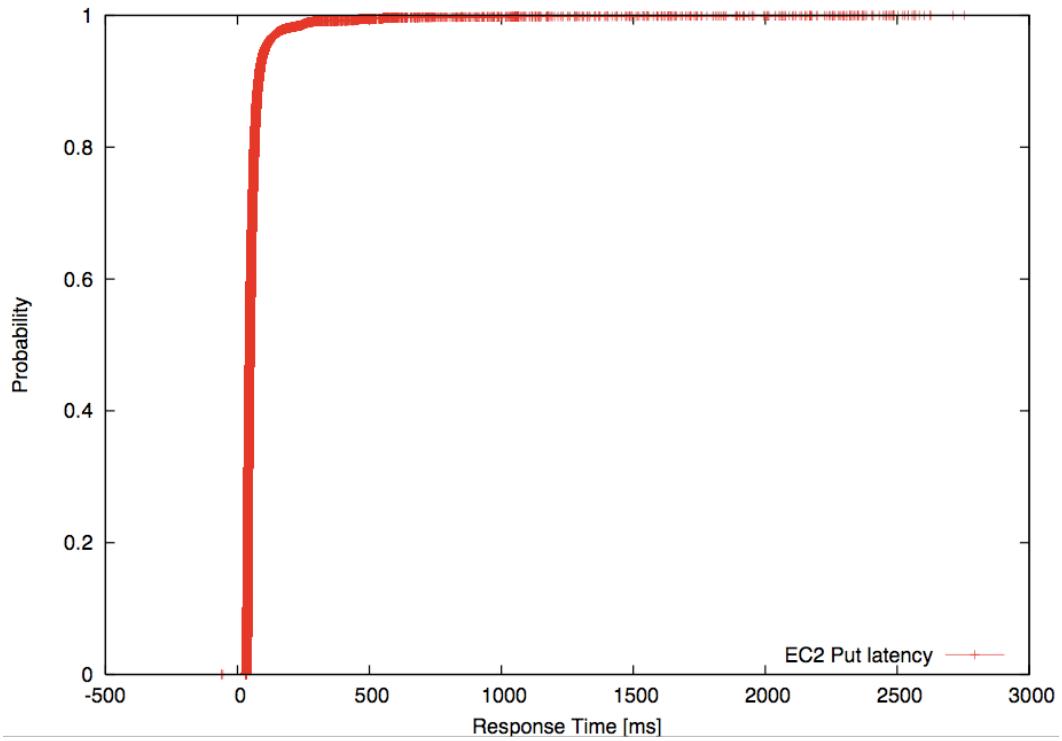


Figure 6.13: S3 put latency during the NSF Award database initialization, shown as a cumulative distribution.

GB of memory resulted in a significant drop in execution time, to approximately 5 minutes. While a considerable improvement, the approach presented in this chapter is still a valuable alternative, since machines with that much memory are not readily accessible to the average developer, and as graphs grow larger, even 15 GB of memory will not be enough, and thrashing will eventually hinder their performance.

6.7 Discussion and future work

This chapter presents TraceOn, which is a software system and analysis methodology that shows promise in handling large trace datasets at scale. Based on our experience using the tool, we can envision a variety of improvements that would benefit its users.

First, we would like to expand the set of input filter criterion used to define datasets. In addition to providing a list of Task Ids, or a time range, it would be beneficial to define a dataset based on keys and values in the reports themselves. For example, one could define a dataset as any traces that transit a particular web server. Second, TraceOn would benefit from the ability to fetch trace data from multiple backends. In this scenario, a datacenter could have a variety of X-Trace backends operating autonomously. A single instance of TraceOn could query those backends, fetching any data it needs to build the appropriate dataset. A key use case of this would be to have a single X-Trace backend per rack of computers, and to have a centrally located TraceOn server. This would minimize the amount of trace data transiting rack-to-rack network connections.

A third extension to TraceOn would be additional language bindings for the real-edge iterator exported by the system. Right now queries must be written in the Java lan-

guage. The rise of dynamic scripting languages such as Ruby and Python mean that many users might not be familiar with Java, but rather these newer languages. Providing an interface to TraceOn would open it up to use by this class of users.

Finally, TraceOn could benefit from the ability to automatically extract semantic information from reports, building various visualizations automatically. A specific example of this would be to extract RPC calls from the trace data during pre-processing. Statistics about those RPC calls could be collected and stored in the on-disk database. This would mean that a variety of useful statistics about the dataset would be visible without the user having to write any queries at all. These statistics would take the form of histograms, distributions of the time spent in various parts of the RPC system, and a list of those tasks that ran exceptionally slow, or exceptionally fast. For this extension to be feasible, a more structured schema would be necessary for the reports. Formalizing and standardizing these schemas is also ongoing work within the X-Trace project.

6.8 Conclusion

A key challenge to tracing large distributed systems is handling the processing of traces at scale. To tackle this problem, we applied the insight gained from a large amount of user experience among X-Trace collaborators at the University of California, Berkeley. We distilled a common set of operations needed to support the types of ad-hoc queries that we have used so far in our projects. First among these is the need to recover a particular topological sort of each X-Trace graph, which discriminates between “real” and “virtual” edges. This function is provided by TraceOn, a software artifact that encapsulates a set of

classical algorithms, modified to be external, as to avoid excessive memory usage.

It pre-processes X-Trace graphs, storing them in a augmented Java language data structures that make use of X-Trace graph locality to more efficiently tradeoff the needs of physical memory and Disk I/O. Once the graph pre-processing is complete, we persist the graphs in on-disk data structures that have been laid out according to their eventual query access patterns. This layout is specific to the particular query interface that we export, meaning that we restrict the way users can interact with the traces. In return, the performance of those restricted queries is significantly improved over naive storage schemes. This improved query time facilitates rapid evolution of those queries, allowing network operators and software developers to address performance and correctness problems within their systems. By giving these groups the tools they need to better understand their systems, they will be able to proactively and quickly improve the reliability and dependability of those systems.

Chapter 7

Future Work and Conclusions

The past sixty years have seen dramatic and fundamental changes in the field of computing. Within a single lifetime, computer programs have grown from single-page calculation routines to globe-spanning distributed systems supporting millions of concurrent users. In terms of our creativity, and vision for what these systems can become, the field of computing is still in its infancy. Yet our ability to engineer the next generation of computing systems is in jeopardy. Unless we can gain better understanding of our creations, we will be unable to guarantee that they faithfully carry out their specifications. The work presented in this dissertation provides for a foundation into providing software developers, network operators, and even end users with better visibility into the systems that they depend on. In the next section, we present a set of future work that would benefit X-Trace. We then conclude by summarizing the main contributions of this dissertation.

7.1 Future directions

X-Trace is the first realization of a tracing methodology that provides for cross-layer, end-to-end tracing through the use of instrumented applications. Its realization in software has allowed us to demonstrate its usefulness in a variety of real-world, deployed applications. However, there are a future directions that would make it more powerful, would provide for better visibility, and would make it easier to use. These future directions are:

1. **Additional language bindings:** X-Trace can currently instrument Java, C, and C++ programs. The rise of dynamic scripting languages like Python and Ruby provide web developers with easy-to-use platforms for developing new and powerful applications, and X-Trace should be ported to include these languages. Furthermore, in the case of Ruby, the Ruby integrated development environment (IDE) is responsible for generating much of the final code, based on various conventions in the source language. We believe that the IDE could also generate X-Trace instrumentation during this code generation process. Furthermore, since the IDE is generating code based on various assumptions (e.g., where the database is located), these assumptions could be recorded as assertions. After the fact, these assertions could be checked against the generated X-Trace graphs, signaling the network operator in cases of mismatch.

2. **Automated instrumentation:** The Java language provides for load-time modification of the underlying bytecodes. By writing an X-Trace module that uses these interfaces, we believe that a set of basic X-Trace instrumentations could be applied to code without the developer having to write any additional code. While not applica-

ble for all software packages, there are a variety of well-known and commonly used software packages which could be automatically instrumented this way.

3. **Application-aware trace analysis:** For common and popular applications such as Hadoop Map/Reduce, the X-Trace backend could automatically generate a variety of visualizations that would provide the software developer and network operator with insight into the inner workings of that software. The backend would be able to extract information about semantically meaningful software events, flagging those that sufficiently deviate from expected behavior.

Now that we have discussed some initial directions for future work, we conclude the dissertation by reviewing the primary contributions of this work.

7.2 Contributions of the dissertation

In this work, we have made three primary contributions, which are:

1. **X-Trace system and methodology:** X-Trace is a set of application programming interfaces, libraries, and software infrastructure for instrumenting distributed systems, collecting trace data from those systems, visualizing, and analyzing that data. Software developers modify their software to include instrumentation points. During program execution, the X-Trace library uses these instrumentation points to generate trace data that is stored locally on each node. The X-Trace backend is responsible for collecting the trace data together into a logically centralized place.

2. **Use cases demonstrating the usefulness of X-Trace.** We have integrated X-Trace

into a variety of real-world applications, and deployed those applications into local testbeds, wide-area distributed testbeds, and virtual datacenters. The experience gained in these deployments not only validated our design decisions, but shaped those decisions, providing for an incremental refinement of the X-Trace interfaces and infrastructure.

3. **Support for analyzing traces at scale.** Because X-Trace has been useful in diagnosing failures in deployed systems, it has been integrated into larger and larger applications. The result has been traces that are too large to analyze with the original X-Trace tools. To address this deficiency, we have designed new algorithms and software encapsulating those algorithms to process traces with millions of events in them. By utilizing external data structures and by careful management of buffers, we are able to process X-Trace graphs from large systems on a single server, or even a laptop. The result is a more usable tool for researchers, software developers, and other X-Trace users.

Bibliography

[1] Ieee 802.1x port-based network access control. http://www.ieee802.org/1/pages-/802.1x.html.

[2] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 13(1):80–96, 2002.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003. ACM Press.

[4] Nortel alteon webswitch. http://www.nortelnetworks.com/products/01/alteon/-webswitch/.

[5] Amazon.com. http://www.amazon.com/.

[6] Amazon elastic compute cloud (ec2). http://aws.amazon.com/ec2.

[7] C. Amza, E. Cecchet, A. Chanda, A. Cox, R. Gil S. Elnikety, and J. Marguerite.

Specification and implementation of dynamic web site benchmarks. In *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, Nov 2002.

[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] The apache webserver. http://www.apache.org/.

[10] Application Response Measurement, http://www.opengroup.org/tech/management/-arm/.

[11] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 256–269, New York, NY, USA, 1996. ACM Press.

[12] Luciano Baresi, Carlo Ghezzi, and Sam Guinea. Smart monitors for composed services. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 193–202, New York, NY, USA, 2004. ACM Press.

[13] Paul T. Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Usenix OSDI*, pages 259–272, 2004.

[14] Bea weblogic platform. http://www.bea.com/.

[15] Nina Bhatti, Ann Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. In *9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000.

[16] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated services, December 1998. Status: PROPOSED STANDARD.

[17] L. Blunk and J. Vollbrecht. PPP Extensible Authentication Protocol (EAP). RFC 2284 (Proposed Standard), March 1998. Obsoleted by RFC 3748, updated by RFC 2484.

[18] Peter Bodik, Greg Friedman, Lukas Biewald, Helen Levine, George Candea, Kayur Patel, Gilman Tolle, Jon Hui, Armando Fox, Michael Jordan, and David Patterson. Combining visualization and statistical analysis to improve operator confidence and efficiency for failure detection and localization. In *2nd IEEE International Conference on Autonomic Computing (ICAC '05)*, Seattle, June 2005.

[19] J. A. Boyan and M. L. Littman. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in Neural Information Processing Systems*, 6, 1994.

[20] R. Braden, D. Clark, and S. Shenker. RFC 1633: Integrated services in the Internet architecture: an overview, June 1994. Status: INFORMATIONAL.

[21] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Re-

source ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997. Status: PROPOSED STANDARD.

[22] Jose Brustoloni. Protecting electronic commerce from distributed denial-of-service attacks. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 553–561, New York, NY, USA, 2002. ACM Press.

[23] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middlware architectures for generating dynamic web content. In *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

[24] E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *17th ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOpsla 2002)*, Seattle, WA, Nov 2002.

[25] Anupam Chanda, Khaled Elmeleegy, Alan L. Cox, and Willy Zwaenepoel. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proceedings of the 6th Intl. Middleware Conference – Middleware 2005*, pages 42–59, November 2005.

[26] Mohit Chawla, Teemu Koponen, Karthik Lakshminarayanan, Anirudh Ramachandran, Arsalan Tavakoli, Atul Vasu, Scott Shenker, and Ion Stoica. Data-Oriented Network Architecture (DONA), in submission.

[27] Checkpoint InterSpect firewall. http://checkpoint.com/products/interspect-/index.html.

[28] J. Chen and P. Druschel. Ants and reinforcement learning: a case study in routing in dynamic networks. Technical Report TR96-259, Computer Science Dept, Rice University, 1998.

[29] 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.

[30] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Dave Patterson, Armando Fox, and Eric A. Brewer. Path-based failure and evolution management. In *Proc. USENIX NSDI*, March 2004.

[31] Samuel P.M. Choi and Dit-Yan Yeung. Predictive q-routing: A memory-based reinforcement learning approach to adaptive traffic control. *Advances in Neural Information Processing Systems*, 8, 1996.

[32] Cisco MDS 9000 Series Multilayer Switches. http://www.cisco.com/en/US-/products/hw/ps4159/ps4358/.

[33] Cisco PIX 500 Security Appliance. http://www.cisco.com/en/US/products/hw-/vpndevc/ps2030/.

[34] David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. A knowledge plane for the internet. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–10, New York, NY, USA, 2003. ACM Press.

[35] Cohen, Chase, Goldszmidt, Kelly, and Symons. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, CA, Dec 2004.

[36] Ira Cohen, Moises Goldszmidt, Terence Kelly, Symons Julie, and Chase Jeff. Correlating instrumentation data to system states: A building block for automated diagnosis and control. In *Operating System Design and Implementation (OSDI)*, December 2004.

[37] R. L. Cottrell, C. Logg, M. Chhaparia, M. Grigoriev, F. Haro, F. Nazir, and M. Sandford. Evaluation of techniques to detect significant network performance problems using end-to-end active network measurements. In *NOMS'06: Network Operations and Management Symposium*, pages 85–94. IEEE/IFIP, 2006.

[38] D. Crocker. RFC 822: Standard for the format of ARPA Internet text messages, August 1982.

[39] Martin Davis. *The Universal Computer: The Road from Leibniz to Turing*. W. W. Norton & Company, May 2000.

[40] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, December 1998.

[41] Yixin Diao, Xue Lui, Steve Froehlich, Joseph L Hellerstein, Sujay Parekh, , and Lui Sha. On-line response time optimization of an apache web server. In *Eleventh International Workshop on Quality of Service (IWQoS '03)*, Monterey, CA, Jun 2003.

[42] Ebay.com. http://www.ebay.com/.

[43] Douglas Ennis, Divyangi Anchan, and Mahmoud Pegah. The front line battle against p2p. In *SIGUCCS '04: Proceedings of the 32nd annual ACM SIGUCCS conference on User services*, pages 101–106, New York, NY, USA, 2004. ACM Press.

[44] F5 networks big-ip blade server. http://www.f5.com/f5products/products/bigip/ltm-/BladeController/.

[45] F5 networks icontrol api. http://f5.com/f5products/products/iControl/.

[46] 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.

[47] Mozilla firefox browser. http://www.mozilla.com/firefox.

[48] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *NSDI'07: Networked Systems Design and Implementation*. USENIX, Cambridge, MA, 2007.

[49] 2-dfs graph reconstruction algorithm. Personal communication, Rodrigo Fonseca.

[50] Ganglia. http://ganglia.sourceforge.net/.

[51] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proc. USENIX Annual Technical Conference*, 2006.

[52] A. Goel, D. Steere, C. Pu, and J. Walpole. Swift: A feedback control and dynamic

reconfiguration toolkit. Technical Report CSE-98-009, Oregon Graduate Institute, Portland, OR, June 1998.

[53] A. Goel, D. Steere, C. Pu, and J. Walpole. Adaptive resource management via modular feedback control, 1999.

[54] GraphViz http://www.graphviz.org/.

[55] Anders Gunnar, Mikael Johansson, and Thomas Telkamp. Traffic matrix estimation on a large ip backbone: a comparison on real data. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 149–160, New York, NY, USA, 2004. ACM Press.

[56] Pankaj Gupta and Nick McKeown. Packet classification on multiple fields. In *SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[57] Hadoop map/reduce system. http://hadoop.apache.org/core.

[58] S. Hanks, T. Li, D. Farinacci, and P. Traina. RFC 1702: Generic routing encapsulation over IPv4 networks, October 1994. Status: INFORMATIONAL.

[59] Tristan Henderson, David Kotz, and Ilya Abyzov. The changing usage of a mature campus-wide wireless network. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 187–201, New York, NY, USA, 2004. ACM Press.

[60] P. Hoffman and F. Yergeau. UTF-16, an encoding of ISO 10646. RFC 2781 (Informational), February 2000.

[61] 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.

[62] Alefiya Hussain, Genevieve Bartlett, Yuri Pryadkin, John Heidemann, Christos Papadopoulos, and Joseph Bannister. Experiences with a continuous network tracing infrastructure. In *MineNet '05: Proceeding of the 2005 ACM SIGCOMM workshop on Mining network data*, New York, NY, USA, 2005. ACM Press.

[63] Ipmonitor. http://www.ipmonitor.com/.

[64] Intel IXP 1200 network processor. http://www.intel.com/design/network/products-/npfamily/ixp1200.htm.

[65] Intel IXP 2000-Series network processor. http://www.intel.com/design/network-/products/npfamily/ixp2xxx.htm.

[66] Sun java enterprise edition 2. http://java.sun.com/j2ee/index.jsp.

[67] A Jain, S Floyd, M Allman, and P Sarolahti. Quick-Start for TCP and IP. IETF draft-tsvwg-quickstart-00.txt, May 2005.

[68] Jboss application server. http://www.jboss.org/products/jbossas/.

[69] Jonas: Java open application server. http://jonas.objectweb.org/.

[70] JRadius http://www.jradius.org/.

[71] Thomas Karagiannis, Andre Broido, Michalis Faloutsos, and Kc Claffy. Transport layer identification of p2p traffic. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 121–134, New York, NY, USA, 2004. ACM Press.

[72] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 229–240, New York, NY, USA, 2005. ACM Press.

[73] Frank Kargl, Joern Maier, and Michael Weber. Protecting web servers from distributed denial of service attacks. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 514–524, New York, NY, USA, 2001. ACM Press.

[74] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 89–102, New York, NY, USA, 2002. ACM Press.

[75] Personal communication. Randy H. Katz.

[76] Ramana Rao Kompella, Albert Greenberg, Jennifer Rexford, Alex C. Snoeren, and Jennifer Yates. Cross-layer visibility as a service. In *Proc. ACM SIGCOMM HotNets Workshop*, November 2005.

[77] David Kotz, Tristan Henderson, and Ilya Abyzov. CRAWDAD

data set dartmouth/campus (v. 2007-02-08). Downloaded from http://crawdad.cs.dartmouth.edu/dartmouth/campus, February 2007.

[78] Anurag Kumar. Comparative performance analysis of versions of tcp in a local network with a lossy link. *IEEE/ACM Trans. Netw.*, 6(4):485–498, 1998.

[79] S. Kumar and R. Miikkulainen. Confidence based dual reinforcement q-routing: An adaptive online network routing algorithm. In *Sixteenth international Joint Conference on Artificial intelligence*, pages 758–763, San Francisco, CA, August 1999.

[80] Aleksandar Kuzmanovic and Edward W. Knightly. Low-rate tcp-targeted denial of service attacks: the shrew vs. the mice and elephants. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 75–86, New York, NY, USA, 2003. ACM Press.

[81] T. V. Lakshman, Upamanyu Madhow, and Bernhard Suter. TCP/IP performance with random loss and bidirectional congestion. *IEEE/ACM Trans. Netw.*, 8(5):541–555, 2000.

[82] E. Lassettre, D. W. Coleman, Y. Diao, S. Froehlich, J. L. Hellerstein, L. Hsiung, T. Mummert, M. Raghavachari, G. Parker, L. Russell, M. Surendra, V. Tseng, N. Wadia, and P. Ye. Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges with Resource Actions that Have Lead Times. In *Lecture Notes in Computer Science*, volume 2867, pages 82–92. Springer-Verlag, Jan 2004.

[83] Libasync library. http://pdos.csail.mit.edu/6.824-2004/async/index.html.

[84] C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001.

[85] K. McCloghrie and M. T. Rose. RFC 1213: Management information base for network management of TCP/IP-based internets:MIB-II, March 1991.

[86] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: existing techniques and new directions. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–174, New York, NY, USA, 2002. ACM Press.

[87] Gokhan Memik and William H. Mangione-Smith. A flexible accelerator for layer 7 networking applications. In *DAC '02: Proceedings of the 39th conference on Design automation*, pages 646–651, New York, NY, USA, 2002. ACM Press.

[88] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–60, New York, NY, USA, 2005. ACM Press.

[89] Netfilter and IP-Tables. http://www.netfilter.org/.

[90] Cisco NetFlow Services and Applications White Paper, http://www.cisco.com/go-/netflow.

[91] OpenRadius http://www.xs4all.nl/evbergen/openradius/.

[92] Opensea alliance. http://www.openseaalliance.org/.

[93] HP OpenView. http://www.managementsoftware.hp.com/.

[94] Packeteer packetseeker. http://www.packeteer.com/prod-sol/products/packetshaper-.cfm.

[95] Page and Lawrence. Method for node ranking in a linked database. United States Patent 6,285,999, January 1998.

[96] Konstantina Papagiannaki, Nina Taft, and Anukool Lakhina. A distributed approach to measure ip traffic matrices. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 161–174, New York, NY, USA, 2004. ACM Press.

[97] Vern Paxson. An integrated experimental environment for distributed systems and networks. In *Computer Networks*, volume 31, pages 2435–2464, Boston, MA, December 1999.

[98] J. Postel. RFC 791: Internet Protocol, September 1981.

[99] Patrick Reynolds, Charles Killian, Janet Wiener, Jeffrey Mogul, Mehul Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *Proc. USENIX NSDI*, May 2006.

[100] C. Rigney, S. Willens, A. Rubens, and W. Simpson. Remote Authentication Dial In User Service (RADIUS). RFC 2865 (Draft Standard), June 2000. Updated by RFCs 2868, 3575.

[101] E. Rosen, A. Viswanathan, and R. Callon. RFC 3031: Multiprotocol Label Switching Architecture, January 2001.

[102] 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.

[103] The rubis workload. http://rubis.objectweb.org/.

[104] Stefan Savage, David Wetherall, Anna R. Karlin, and Tom Anderson. Practical network support for IP traceback. In *SIGCOMM*, pages 295–306, 2000.

[105] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[106] Subhabrata Sen, Oliver Spatscheck, and Dongmei Wang. Accurate, scalable in-network identification of p2p traffic using application signatures. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 512–521, New York, NY, USA, 2004. ACM Press.

[107] SNMPv2 Working Group, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. RFC 1902: Structure of management information for version 2 of the Simple Network Management Protocol (SNMPv2), January 1996.

[108] SNMPv2 Working Group, J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. RFC 1905: Protocol operations for version 2 of the Simple Network Management Protocol (SNMPv2), January 1996.

[109] Snort. http://www.snort.org.

[110] Augustin Soule, Antonio Nucci, Rene Cruz, Emilio Leonardi, and Nina Taft. How to identify and estimate the largest traffic matrix elements in a dynamic environment. In *SIGMETRICS 2004/PERFORMANCE 2004: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 73–84, New York, NY, USA, 2004. ACM Press.

[111] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 216–229, New York, NY, USA, 2001. ACM Press.

[112] Splunk http://www.splunk.com.

[113] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. *SIGCOMM Comput. Commun. Rev.*, 29(4):135–146, 1999.

[114] I. Stoica and H. Zhang. Lira: An approach for service differentiation in the internet. In *Proceedings of Nossdav*, Jun 1998.

[115] Ion Stoica. *Stateless Core: A Scalable Approach for Quality of Service in the Internet*. PhD thesis, Carnegie Mellon University, December 2000.

[116] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 73–86, New York, NY, USA, 2002. ACM Press.

[117] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. *IEEE/ACM Trans. Netw.*, 12(2):205–218, 2004.

[118] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[119] Ion Stoica, Scott Shenker, and Hui Zhang. Core -stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *SIGCOMM*, pages 118–130, 1998.

[120] Sun Directory Server 5.2 http://sun.com/software/products/directory_srvr-/home_directory.xml.

[121] Sun Secure Application Switch. http://www.sun.com/products/networking/switches-/n2000/.

[122] TCPDUMP. http://www.tcpdump.org/.

[123] Facebook Thrift. http://developers.facebook.com/thrift/.

[124] Tom Verdickt, Wim Van de Meerssche, and Koert Vlaeminck. Modeling the performance of a nat/firewall network service for the ixp2400. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 137–144, New York, NY, USA, 2005. ACM Press.

[125] Paul Vixie. Extension Mechanisms for DSN (EDNS0). RFC 2671, August 1999.

[126] Vmware. http://www.wmware.com/.

[127] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251 (Proposed Standard), December 1997. Obsoleted by RFCs 4510, 4511, 4513, 4512, updated by RFCs 3377, 3771.

[128] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable high-speed prefix matching. *ACM Trans. Comput. Syst.*, 19(4):440–482, 2001.

[129] Xin Wang, Henning Schulzrinne, Dilip Kandlur, and Dinesh Verma. Measurement and analysis of ldap performance. In *SIGMETRICS '00: Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 156–165, New York, NY, USA, 2000. ACM Press.

[130] D. Watson, M. Smart, G. R. Malan, and F. Jahanian. Protocol scrubbing: network security through transparent flow modification. In *IEEE/ACM Transactions on Networking*, volume 12, pages 261–273, April 2004.

[131] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI02*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[132] Wikipedia Infrastructure, http://meta.wikimedia.org/wiki/wikimedia_servers.

[133] F. C. Williams Remarks. http://www.computer50.org/mark1/new.baby.html.

[134] X-Trace API http://www.x-trace.net/.

[135] Ying Xu and Roch Guerin. On the robustness of router-based denial-of-service (dos) defense systems. *SIGCOMM Comput. Commun. Rev.*, 35(3):47–60, 2005.

[136] Avi Yaar, Adrian Perrig, and Dawn Song. An endhost capability mechanism to mitigate DDoS flooding attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2004.

[137] David K. Y. Yau, John C. S. Lui, Feng Liang, and Yeung Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Netw.*, 13(1):29–42, 2005.

[138] F. Yergeau. UTF-8, a transformation format of ISO 10646. RFC 3629 (Standard), November 2003.

[139] L. Yu and H. Liu. Feature selection for high-dimensional data: A fast correlation-based filter solution. In *Twentieth International Conference on Machine Leaning (ICML-03)*, pages 856–863, Washington, D.C., USA, August 2003.

[140] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *International Conference on Distributed Computing Systems*, Vienna, Austria, July 2002.

[141] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensembles of models for automated diagnosis of system performance problems. In *International conference on Dependable Systems and Networks (DSN)*, pages 644–653, June 2005.

[142] Yin Zhang, Matthew Roughan, Nick Duffield, and Albert Greenberg. Fast accurate computation of large-scale ip traffic matrices from link loads. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 206–217, New York, NY, USA, 2003. ACM Press.

Appendix A

X-Trace Enterprise Edition user manual

X-Trace

Installation, Administration, and Usage Guide

UC Berkeley • Version 1.0 • April 10, 2007 • http://www.x-trace.net

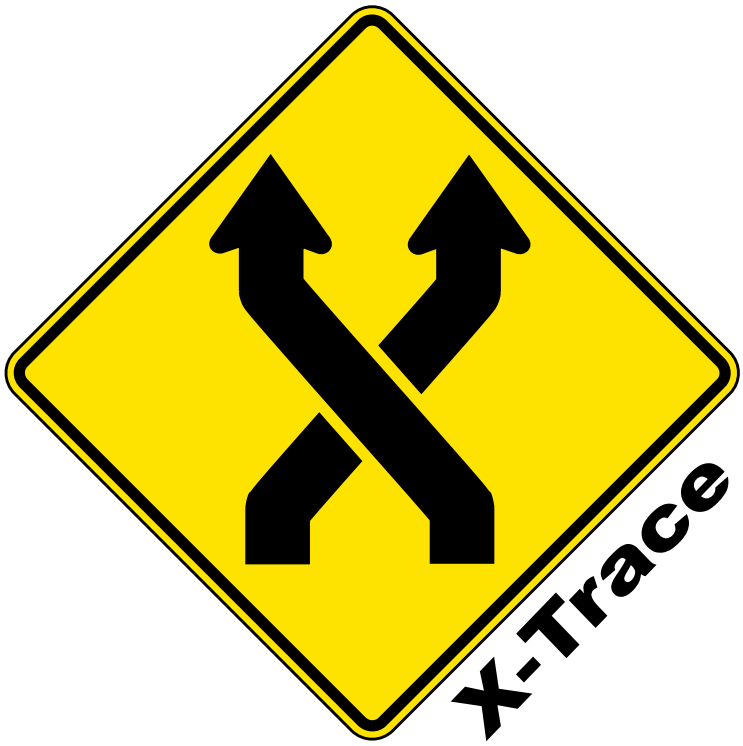


Table of Contents

Introduction 3

X-Trace Architecture 4

- X-Trace front-end 5
- X-Trace Back-end 7

Installation prerequisites 9

Installation 11

- Installing service agents 11
- Installing the per-host reporting daemon 11
- Installing the back-end (Overview) 11

Post-installation configuration 12

Starting and stopping X-Trace 13

Accessing trace data 14

- Via the HTTP protocol 14
- Via the ReportRetriever command-line tool 14

License 15

Introduction

Internet services are built out of distributed components (e.g., load balancer, web server, back-end database), make use of sophisticated network mechanisms (e.g., VPNs, NATs, overlays, tunnels), and can span multiple administrative domains (e.g., the client's web proxy and the server's load balancer). When these complex systems misbehave, it is often quite difficult to diagnose the source of the problem.

To this end, we have developed an integrated tracing framework called X-Trace. A user or operator invokes X-Trace when initiating an application task (e.g., a web request), by inserting X-Trace metadata with a task identifier in the resulting request. This metadata is then propagated down to lower layers through protocol interfaces, that may need to be modified to carry X-Trace metadata, and also along all recursive requests that result from the original task. This is what makes X-Trace comprehensive; it tags all network operations resulting from a particular task with the same task identifier. We call the set of network operations connected with an initial task the resulting *task tree*.

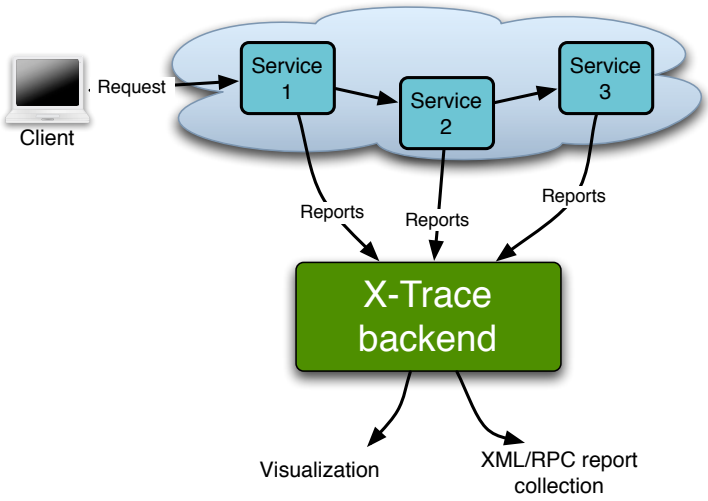
X-Trace-enabled devices log the relevant information connected with each tagged network operation, which can then be reported back. The trace information associated with a task tree gives the user or operator a comprehensive view of what network operations were executed as part of a task. X-Trace task trees are runtime traces of a task execution, and so long as individual components are integrated into the framework, there is no need for prior configuration of their dependencies.

X-Trace includes tools to insert and examine the metadata stored in requests, as well as infrastructure necessary to instrument services along the path, collect reports from those services, and store them in persistent storage. From there, interested parties can collect the reports or make use of some basic visualization tools provided with the distribution.

X-Trace Architecture

The components of the X-Trace system are distributed throughout the network, yet can be grouped roughly into two categories:

1. Front-end infrastructure, which runs on hosts in your datacenter.
2. Back-end infrastructure. This infrastructure is hosted at UC Berkeley, however you can also run it in your datacenter.



A client's request might transit a number of composed services (in this case services 1, 2, and 3). The client includes X-Trace metadata in their request, which is processed by each of the X-Trace enabled services on the path. These services use that metadata to generate reports, which are send to the back-end infrastructure. The client, or another interested party, can query the back-end to collect the reports or to view visualizations of the datapath.

In the figure above, the client's request (which is annotated with X-Trace metadata) is handled by a series of services. For example, a client might contact an HTTP load balancer, which then forwards the request to an Apache process on a webserver. That webserver might have to issue SQL requests to a Postgres database. To trace the request as it transits through each of these services, each service must *propagate* X-Trace metadata to the next service, as well as *issue reports* which can be used later to reconstruct the user's task.

> A *task* is a set of related requests, typically initiated by the user, necessary to carry out an operation. The requests making up a task might span layers (i.e., IP, TCP, HTTP, SQL) and might span multiple services.

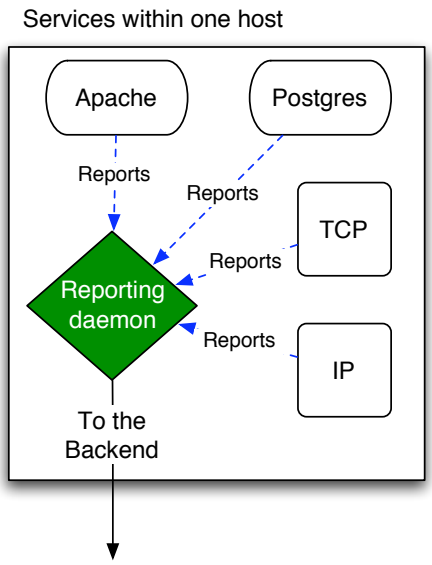
X-Trace *service agents* are linked into each of the services to carry out the propagation and reporting duties. A *reporting daemon* collects the reports from the agents and sends them to the back-end. Front-end agents communicate with the reporting daemon over a local UDP socket.

> *Service agents* are linked into application code, and are responsible for propagating X-Trace metadata from one request to another, as well as generating reports that are sent to the reporting daemon. The *reporting daemon* is a user-level program that runs in each host. It collects reports from X-Trace service agents, and forwards them to the back-end infrastructure.

It is the responsibility of the back-end infrastructure to collect reports from the reporting daemons that exist on hosts that are on the datapath. These reports are used to generate visualizations of the task, or they can be collected directly by interested parties for use in other types of analysis. The reporting daemons communicate with the back-end via the Java message service (JMS).

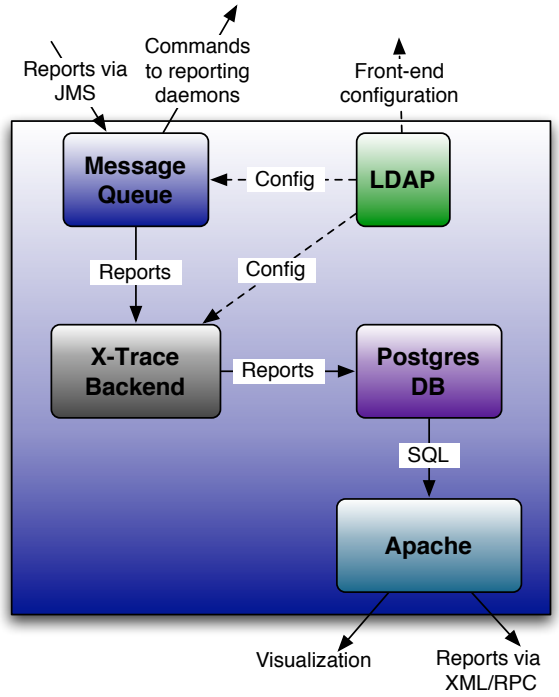
X-Trace front-end

The front end consists of X-Trace service agents that run in each service, as well as a single reporting daemon per host:



Reports are generated by X-Trace service agents, which are located in the Apache, Postgres, TCP, and IP modules shown above. These reports are sent in a UDP message to the reporting daemon, which forwards them to the back-end.

X-Trace Back-end



The X-Trace back-end, which runs as a hosted service at UC Berkeley, but can also be installed in your network, consists of several components. These components include:

1. LDAP (Lightweight Directory Access Protocol). This components serves as the store of configuration information for the entire X-Trace system. By localizing configuration information in this centralized service, the administrator only has to update a single place to adjust the service. Also, the front-end clients to X-Trace require substantially less configuration. In fact, they only need a *jndi.properties* file, explained later in this document. When clients want to connect to the message queue, or when the X-Trace back-end code wants to connect to the database, they get connections directory form the LDAP database.

2. Message Queue. The message queue is a JMS (Java Messaging Service) component that provides two-way communication channels between the front-end clients and the back-end. This JMS channel supports configurable reliability semantics, security (TLS), and other delivery features. It is used to provide a two-way channel from the back-end to the front-ends for the X-Trace control plane, which is used to locate, control, and manage the front-end clients. It is also used as a data-plane for the delivery of X-Trace reports from the front-ends to the back-end.

3. Postgresql Database. X-Trace reports are stored in a this database. Additionally, information about each of the connected clients is stored as well, meaning that an administrator can query that database for information on clients that are connected or have recently been connected to this back-end infrastructure.

4. Apache webserver. The Apache webserver provides a bridge between the back-end infrastructure and users who need to access to the trace data. Currently two interfaces to this data are available: an XML/RPC interface for programatically accessing the trace data, and a cgi-bin program that provides graphic visualizations of X-Trace tasks.

5. X-Trace Java code. Custom-written Java code glues together these components and provides a unified trace collection service.

Installation prerequisites

The X-Trace front-end infrastructure must be installed on each host that is going to issue reports (typically each host on a datapath in your datacenter). An X-Trace installation consists of three different components:

1. Each network service (Apache, Postgres, IP, Chord, etc) must be linked with a service agent. There can be multiple of these service agents per host.
2. Exactly one reporting daemon must be installed on each host.
3. The X-Trace back-end software must be installed logically once for each datacenter or autonomous domain. This back-end consists of multiple components that do not necessarily have to reside on the same server, however the distribution makes them available as a single virtual appliance image.

To install the X-Trace front-end reporting daemon, you need:

- The Java 1.5 runtime
- The ability to open and listen to a UDP port on the *localhost* interface. This port is configurable.

To generate reports to send to the reporting daemon, X-Trace service agents must be installed into each of your applications. The prerequisites for installing these agents depend on the specific application, and are outlined in the X-Trace Agent Reference Guide.

The X-Trace back-end is available in two forms: a VMware appliance, and as individual components.

Installing the VMware appliance requires:

- The free VMware player, available from http://www.vmware.com/products/player/
- An available IP address for the appliance, obtainable via DHCP

Installing the back-end from individual components is an advanced procedure that is not described in this document. E-mail the authors (info@x-trace.net) for more information on how to do this.

Installation

Installing service agents

Installing service agents is described in the X-Trace Agent Reference Guide.

Installing the per-host reporting daemon

On each host that will generate reports, you need to install and run the front-end reporting daemon. To install the daemon:

1. Download the X-Trace front-end distribution in either **.tar.gz** or **.zip** format.
2. Unzip (or un-tar) these files into a directory. This directory will serve as the root of the front-end daemon:
    1. (for **.tar.gz** distributions):

        ```
        cd /usr/local

        gzip -d xtrace_fe.tar.gz

        tar -xvf xtrace_fe.tar
        ```

    2. (for **.zip** distributions):

        ```
        cd /usr/local

        unzip xtrace_fe.zip
        ```

Installing the back-end (Overview)

Installation instructions for the back-end will be available in the next version, which will include the back-end code and infrastructure components.

Post-installation configuration

The front-end reporting daemon configuration is located in the `jndi.properties` file, located in `XTRACE_HOME/conf`.

To make use of the UC Berkeley-hosted X-Trace backend, this file should have the following entries:

```
java.naming.factory.initial=com.sun.jndi.ldap.LdapCtxFactory

java.naming.provider.url=ldap://jndi.x-trace.net:389/ou=Public,dc=x-trace,dc=net
```

Since the service agents forward all reporting traffic to a local, well known port, they do not require post-installation configuration.

Starting and stopping X-Trace

Assuming that you installed the daemon in the `/usr/local/xtrace_fe` directory, execute the following commands. If you installed X-Trace elsewhere, then execute the scripts in the appropriate directory.

To start the X-Trace front-end reporting daemon:

```
$ /usr/local/xtrace_fe/start_fe.sh
```

To stop the daemon:

```
$ /usr/local/xtrace_fe/stop_fe.sh
```

Accessing trace data

X-Trace trace data can be collected from the back-end via the HTTP protocol. Other methods (i.e., XML/RPC) are planned, though currently unimplemented.

Via the HTTP protocol

If you are not using the UC Berkeley-hosted back-end, then you will need to substitute the host-name of the node storing X-Trace reports in the example code below.

All X-Trace reports belonging to a given Task ID can be retrieved by requesting the following URL:

```
http://reports.x-trace.net:8080/RetrieveReports/RetrieveReports?taskid=XX
```

where XX is the task ID you want. The format of the results are:

1. report 1
2. newline
3. report 2
4. newline
5. etc

Via the ReportRetriever command-line tool

There is a small command-line tool that will read reports from the back-end (assuming they are hosted at UC Berkeley. If the back-end infrastructure is running in your network, this tool will need to be modified to reflect the hostname where your reports are stored.

The tool is called with a single argument which is the task ID, in string format:

```
$ java edu.berkeley.xtrace.util.ReportRetriever F3C60172
```

License

Copyright (c) 2006, 2007, 2008 Regents of the University of California

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Appendix B

X-Trace metadata specification version 2.0

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, to tasks. These 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**		
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
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

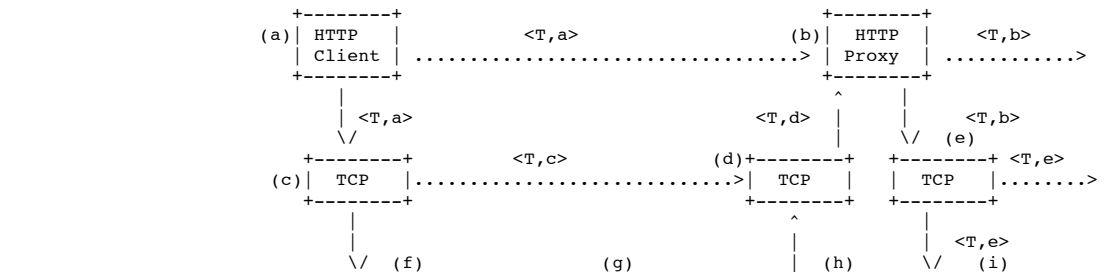
For text-based protocols (ASCII, Unicode), like HTTP, SIP, or email, for example, X-Trace Metadata is represented as a the hexadecimal string of the successive bytes in the binary representation. The characters A to F SHOULD be represented in CAPITAL LETTERS, but implementations SHOULD be tolerant to 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.



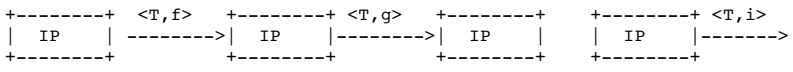


Figure 1.

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