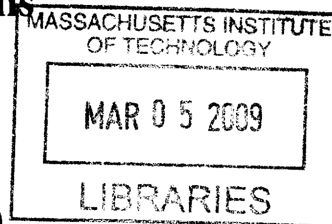
Increasing the Robustness of Networked Systems



by

Srikanth Kandula

M.S., University of Illinois at Urbana-Champaign (2003)
B.Tech., Indian Institute of Technology, Kanpur (2001)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2009

© Srikanth Kandula, MMIX. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and
electronic copies of this thesis document in whole or in part.

Author . . . .
Department of Electrical Engineering and Computer Science
September 24, 2008

Certified by . . . .
Dina Katabi
Associate Professor
Thesis Supervisor

Accepted by . . . .
Terry P. Orlando
Chairman, Department Committee on Graduate Students

ARCHIVES

Increasing the Robustness of Networked Systems
by
Srikanth Kandula

Submitted to the Department of Electrical Engineering and Computer Science
on February, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

What popular news do you recall about networked systems? You've probably heard about the several hour failure at Amazon's computing utility that knocked down many startups for several hours, or the attacks that forced the Estonian government web-sites to be inaccessible for several days, or you may have observed inexplicably slow responses or errors from your favorite web site. Needless to say, keeping networked systems robust to attacks and failures is an increasingly significant problem.

Why is it hard to keep networked systems robust? We believe that uncontrollable inputs and complex dependencies are the two main reasons. The owner of a web-site has little control on when users arrive; the operator of an ISP has little say in when a fiber gets cut; and the administrator of a campus network is unlikely to know exactly which switches or file-servers may be causing a user's sluggish performance. Despite unpredictable or malicious inputs and complex dependencies we would like a network to self-manage itself, i.e., diagnose its own faults and continue to maintain good performance.

This dissertation presents a generic approach to harden networked systems by distinguishing between two scenarios. For systems that need to respond rapidly to unpredictable inputs, we design online solutions that re-optimize resource allocation as inputs change. For systems that need to diagnose the root cause of a problem in the presence of complex subsystem dependencies, we devise techniques to infer these dependencies from packet traces and build functional representations that facilitate reasoning about the most likely causes for faults. We present a few solutions, as examples of this approach, that tackle an important class of network failures. Specifically, we address (1) re-routing traffic around congestion when traffic spikes or links fail in internet service provider networks, (2) protecting websites from denial of service attacks that mimic legitimate users and (3) diagnosing causes of performance problems in enterprises and campus-wide networks. Through a combination of implementations, simulations and deployments, we show that our solutions advance the state-of-the-art.

Thesis Supervisor: Dina Katabi
Title: Associate Professor

Contents

Prior Published Work 6

Acknowledgments 7

1 Introduction 8
- 1.1 Robustness as a Resource Management Problem 9
  - 1.1.1 Online Traffic Engineering 10
  - 1.1.2 Protecting Against Denial-of-Service Attacks that Mimic Legitimate Requests 11
- 1.2 Robustness as a Learning Problem 12
  - 1.2.1 Learning Communication Rules 14
  - 1.2.2 Localizing Faults and Performance Problems 15

2 Defending Against DDoS Attacks That Mimic Flash Crowds 16
- 2.1 Overview 16
- 2.2 Threat Model 19
- 2.3 Separating Legitimate Users from Attackers 20
  - 2.3.1 Activating the Authentication Mechanism 20
  - 2.3.2 CAPTCHA-Based Authentication 21
  - 2.3.3 Authenticating Users Who Do Not Answer CAPTCHAs 23
  - 2.3.4 Criterion to Distinguish Legitimate Users and Zombies 23
- 2.4 Managing Resources Efficiently 24
  - 2.4.1 Results of the Analysis 24
  - 2.4.2 Adaptive Admission Control 26
- 2.5 Security Analysis 27
- 2.6 Kill-Bots System Architecture 30
- 2.7 Evaluation 31
  - 2.7.1 Experimental Environment 31
  - 2.7.2 Microbenchmarks 33
  - 2.7.3 Kill-Bots under Cyberslam 33
  - 2.7.4 Kill-Bots under Flash Crowds 35
  - 2.7.5 Benefit from Admission Control 36
  - 2.7.6 User Willingness to Solve Puzzles 36
- 2.8 Related Work 37
- 2.9 Generalizing & Extending Kill-Bots 38

2.10 Limitations & Open Issues 39
2.11 Conclusion 40

3 **Responsive Yet Stable Traffic Engineering** **41**
3.1 Overview 41
3.2 Problem Formalization 43
3.3 TeXCP Design 44
3.3.1 Path Selection 45
3.3.2 Probing Network State 45
3.3.3 The Load Balancer 46
3.3.4 Preventing Oscillations and Managing Congestion 48
3.3.5 Shorter Paths First 50
3.4 Analysis 51
3.5 Performance 52
3.5.1 Topologies & Traffic Demands 53
3.5.2 Metric 53
3.5.3 Simulated TE Techniques 53
3.5.4 Comparison With the OSPF Optimizer 54
3.5.5 Comparison With MATE 57
3.5.6 TeXCP Convergence Time 58
3.5.7 Number of Active Paths 60
3.5.8 Automatic Selection of Shorter Paths 60
3.6 Implementation & Deployment 60
3.7 Related Work 61
3.8 Further Discussion 62
3.9 Concluding Remarks 63

4 **Learning Communication Rules** **65**
4.1 Overview 65
4.2 Learning Communication Rules 68
4.2.1 From Dependency to Association 68
4.2.2 Significance of a Communication Rule 69
4.2.3 Generic Communication Rules 71
4.3 Algorithms to Mine for Rules 73
4.3.1 Composing Communication Rules 74
4.3.2 Towards a Streaming Solution 75
4.4 Evaluation 76
4.4.1 Dataset 76
4.4.2 Metrics 76
4.4.3 Nature of the Rule-Mining Problem 77
4.4.4 Evaluation in Controlled Settings 78
4.4.5 Micro-Evaluation 78
4.4.6 Case Study: Patterns in the Enterprise 80
4.4.7 Case: Patterns in the LabEnterprise 83
4.4.8 Case: Patterns on the Lab's Access Link 86

4.4.9 Case Study: Rules for HotSpot Traces . . . 88
4.5 Discussion . . . 90
4.6 Related Work . . . 90
4.7 Conclusion . . . 91

5 Localizing Faults and Performance Problems **92**
5.1 Overview . . . 92
5.2 Related Work . . . 95
5.3 The Inference Graph Model . . . 96
5.3.1 The Inference Graph . . . 96
5.3.2 Fault Localization on the Inference Graph . . . 100
5.4 The Sherlock System . . . 102
5.4.1 Constructing the Inference Graph . . . 103
5.4.2 Fault Localization Using Ferret . . . 104
5.5 Implementation . . . 105
5.6 Evaluation . . . 106
5.6.1 Micro-Evaluation . . . 106
5.6.2 Evaluation of Field Deployment . . . 108
5.6.3 Comparing Sherlock with Prior Approaches . . . 110
5.6.4 Time to Localize Faults . . . 110
5.6.5 Impact of Errors in Inference Graph . . . 111
5.6.6 Modeling Redundancy Techniques . . . 112
5.6.7 Summary of Results . . . 112
5.7 Discussion . . . 113
5.8 Conclusions . . . 116

6 Concluding Remarks **117**

Appendix **119**
A. Analysing Kill-Bots . . . 119
B. Interaction between Admission Control and Re-transmissions . . . 122
C. Stability of The Kill-Bots Controller . . . 123
D. Proof of TeXCP Stability due to Explicit Feedback . . . 125
E. Proof of TeXCP Load Balancer Convergence . . . 127
F. Value of Preferring Shorter Paths in TeXCP . . . 131
G. Propagating State for Meta-Nodes in Sherlock . . . 132

Bibliography **133**

Prior Published Work

This thesis builds on some prior published work.

Chapter 2 contains joint work with Dina Katabi, Matthias Jacob and Arthur Berger published at NSDI'05 [100]. Chapter 3 contains joint work with Dina Katabi, Bruce Davie and Anna Charny that was published at SIGCOMM'05 [99]. Chapter 4 contains joint work with Ranveer Chandra and Dina Katabi published at SIGCOMM'08 [102]. Chapter 5 builds on Sherlock [54], joint work with Victor Bahl, Ranveer Chandra, Albert Greenberg, David Maltz and Ming Zhang at SIGCOMM'07.

Acknowledgments

I am grateful for the support from Dad, Mom, Neelima and Sasi without which this thesis wouldn't exist. I admire my advisor Prof. Dina Katabi's technical acumen and would be thrilled if a part of that rubs off onto me. I would also like to thank my committee for patiently reading through and commenting on vast chunks of the thesis. Dr. Victor Bahl, Prof. Hari Balakrishnan, Prof. John Guttag, and Prof. Dina Katabi, it is a wonder how well you juggle the many things on your respective plates and manage to pull off each one efficiently. My office mates, Ali Shoeb, Zeeshan Syed, Jerry Chew-Ping and Abe Bachrach were a constant source of fun, frolic and distraction, which goes a long way to keeping sane. For long stretches, life outside of work was courtesy of Rohit and Sanskriti Singh, thanks! I also thank Nate Kushman and Asfandyar Qureshi for listening to many random ideas, a few of which even worked out, Rahul Hariharan for his adroit marketing pitches on how to present and Sachin Katti for generally being the man. Szymon Chachulski was a champ at figuring out logical bugs in drafts. Shyamnath Gollakota and Grace Woo were wonderful group-mates. Ranveer Chandra was a constant source of steady advice and is the man at doing a lot of work while also enjoying it. Folks at Microsoft Research, Dave Maltz, Ming Zhang, Ratul Mahajan, Albert Greenberg and Victor Bahl advised, guided and prodded me through much interesting research for which I can't be grateful enough. Many folks made the ninth floor in Stata (and the erstwhile fifth floor in NE-43) a vibrant place, Magda and Michel, Dave Anderson, Daniel Abadi, Vlad Bychkovsky, Nick Feamster, Ramki Gummadi, Bret Hull, Kyle Jamieson, Jaeyeon Jung, Allen Miu, Stan Rost, Mythili Vutukuru, Eugene Wu, and of course Michael Walfish; work and life was a lot richer because of you guys. Mary McDavitt baked the largest cake anyone has ever baked for me. Dorothy Curtis would always bail me out whenever things crashed. I wrap up by applauding my co-authors, it was a lot of fun working with you and I am glad that I had an opportunity to learn from you – Tural Badirkhanli, Victor Bahl, Arthur Berger, Ranveer Chandra, Anna Charny, Bruce Davie, Albert Greenberg, Matthias Jacob, Dina Katabi, Nate Kushman, Kate Lin, Bruce Maggs, Dave Maltz, Asfandyar Qureshi, Shan Sinha and Ming Zhang.

Chapter 1

Introduction

What popular news do you recall about networked systems?

- You probably heard about the several hour failure at Amazon's computing cluster that brought down many startups who were leasing infrastructure from Amazon [65, 115, 152, 153]. Several of these startups (e.g., Twitter, SmugMug, 37Signals etc.) no longer use Amazon's service.
- You have probably heard about the crippling denial of service attacks that brought down many government websites of Estonia, a neighbor of Russia, by simply fetching the web pages at these sites [14, 15]. In fact, the attack was so severe that the Estonian government resorted to denying access to all IP addresses outside the country believing that the attack was from Russia [16].
- You have probably experienced inexplicably slow responses, or even errors, from websites including services such as Google Mail [138, 141, 154]. Frustrated users do complain (e.g., *Gmail Slow and "Still Working": Enough Is Enough!* [105]) and eventually move on to alternate providers.
- You have probably heard about Wells Fargo, the fifth largest bank in the United States, suffering a fault in its internal network that brought down its entire Internet and telephone banking operations [40, 41]. Not only did this inconvenience users for several hours, costing significant revenue loss to Wells Fargo, but it also allowed attackers to set up phishing websites claiming that they were the real Wells Fargo [42].

Needless to say, keeping networked systems robust to attacks and failures is an increasingly critical problem. In fact, some would say that this is an aftereffect of progress. More than twenty years of effort has led to networks that are highly functional and play key roles in modern life. Consequently, networked systems have become complex, unwieldy and unpredictable. As the demands on networked systems have become more stringent, catastrophic failures and niggling performance problems happen often. Even worse, users and administrators, who are largely left to deal with these problems struggle to cope.

We state the *robustness problem* as follows: When unpredictable events happen, how do we enable a system to maintain good performance as long as possible and then degrade gracefully? The key metrics are responsiveness and stress, i.e., how quickly can the system react to the change, and how much stress can it tolerate before degradation in performance.

Our definition is intentionally broad. For example, the system could be a website, an enterprise network or an internet service provider's (ISP's) backbone network and the event can be a server overload, a fiber-cut or a denial of service attack.

Why is it hard to keep networked systems robust? Let us consider a common scenario. Suppose you are the owner of a website. At best the only thing you can control is your own network and web server. You have little control on the rest of the Internet, on when users arrive, or on their traffic. The fundamental value of your website comes from keeping it openly accessible, yet this means request overloads and attacks might happen at any time. So a challenge for networked systems is the requirement that they be *open yet robust to unpredictable inputs*. Even if you could control everything, suppose you are the administrator of an enterprise network with privileged access to all desktops and routers. Even then there are complex inter-dependencies. Web-servers depend on database backends, network services depend on mapping addresses to names (DNS), and traffic depends on the links, routers and switches along its network path. When faults happen, or when users see poor performance, any one of these components might be responsible. We need to learn these dependencies to figure out what went wrong. An administrator, however, can neither keep track of all the different protocols and server configurations nor can she instrument every component inside an enterprise. So, a second requirement for robustness is *to learn the functional inter-dependencies in operational networks in order to diagnose user's performance problems*.

We observe that the above challenges reflect two faces of the robustness problem. First, the environment is constantly in flux due to unpredictable inputs, attacks and failures. We need to harden systems against this external complexity. Second, networked systems have significant intrinsic complexity owing to the many components, distributed over the network, that interact in myriad ways. These dependencies complicate the job of keeping a network robust. We tackle these prongs of the robustness problem as follows:

- To manage the external complexity, we enable networked systems to adapt quickly and manage resources efficiently in the face of unforeseen events.
- To manage the intrinsic complexity of systems, we learn the implicit dependencies between their parts and build functional representations of the systems.

1.1 Robustness as a Resource Management Problem

Our approach to deal with unpredictable external events is to cast them as the inputs to a resource management problem. The objective of such a resource management problem is some desired combination of performance and fairness to users. The constraints to the problem are the incoming traffic load, the underlying topology of the networked system and inherent capacity limits. Note that any of these constraints might change unexpectedly. For example, legitimate traffic might arrive in massive unpredictable volumes, the so-called slashdot effect [37, 98]; denial of service attacks may create overloads large enough that legitimate requests are starved [13]; optical cables may get cut or routers may fail within an Internet Service Provider's network [170]. To deal with unpredictable inputs, we devise online solutions to the resource management problem so that systems can quickly react to changes and redistribute resources to continue to maintain good performance.

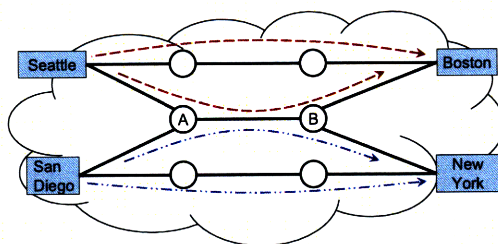


Figure 1-1 – For each Ingress-Egress (IE) pair, there is a TeXCP agent at the ingress router, which balances the IE traffic across available paths in an online, distributed fashion.

We believe that online resource management is necessary to maintain good performance in the face of unpredictable events. Often, the space of possible inputs (e.g., overloads, link failures) is so large that the resource allocation that is appropriate for the common case becomes highly inefficient when the inputs change. Further, it might be impossible to pre-compute the best allocation of resources for all possible inputs. Hence, a reactive solution becomes necessary as we will show in the following examples.

Below we summarize two problems where we cast robustness to external events as an online resource management problem.

1.1.1 Online Traffic Engineering

Internet Service Providers (ISPs) are in the business of carrying user traffic on their underlying topology. When fibers get cut or traffic spikes due to flash crowds, sending packets along their old routes results in dropped packets and unacceptably poor performance for users. Current deployments handle link failures by pre-computing backup paths to route traffic around the failed link(s). To handle traffic spikes, ISPs significantly over-provision their links and distribute load on their underlying topology based on weekly averages. Such schemes cannot deal with unforeseen events for which they did not pre-compute solutions, and as a result, end up over-provisioning links excessively as insurance.

We advocate an alternative approach. If the network were able to react to such unexpected events, it could move traffic away from the faulty links or the highly congested links onto other lightly loaded links. Such an adaptive scheme could support the same amounts of traffic demands, using the same network topology but with fewer spare capacity at the links, thereby reducing cost for ISPs.

We cast dealing with link failures and traffic spikes as the following resource management problem:

$$\min_{\text{Load Distribution}} \quad \text{Maximum Link Utilization} \tag{1.1}$$

subject to the constraints:

$$\text{Meet User Demands} \tag{1.2}$$
$$\text{Existing Network Topology and Link Capacities} \tag{1.3}$$

In Chapter. 3, we present TeXCP, an online solution to the above traffic engineering

problem. TeXCP reacts quickly to link failures and traffic spikes and re-balances the load across the ISP network. Minimizing the max-utilization is a typical formalization of the traffic engineering problem [51, 59, 85, 124]— it removes hot spots and balances the load in the network. Our ideas extend to other metrics such as minimize sum of delays, maximize the minimum spare capacity or maximize the sum of TCPs throughput.

The challenge in re-balancing load lies in responding quickly to changes without destabilizing the network. TeXCP pins multiple paths between each ingress-egress (IE) pair of routers in the ISP core. An IE controller at the ingress node, measures the utilization level along each available path and adapts the share of this IE pair's traffic on each path, moving traffic away from congested or faulty paths and onto the lightly loaded paths. Together, the IE controllers minimize the maximum utilization in the network. To move traffic quickly (much faster than TCP does, for example) and yet not cause synchronized oscillations, TeXCP uses a much-simplified version of XCP [107] feedback from routers in the middle of the network. While an IE controller determines how much traffic should be moved from one path to the other, the traffic is moved only after the routers in the middle of the network allow the change in traffic. We show via simulations on actual ISP topologies, that TeXCP reacts faster and in a more stable fashion than pre-existing online techniques [78], and provides the same utilization and failure resilience as the offline load balancers while requiring only a half or a third of the capacity [85].

1.1.2 Protecting Against Denial-of-Service Attacks that Mimic Legitimate Requests

A denial of service (DoS) attack is a flood of requests that may originate from a few or many addresses with the intention of overwhelming server resources and starving legitimate users. A particularly elusive but increasingly common DoS attack on web servers mimics flash crowds. An attacker uses tens of thousands of zombies, otherwise regular machines that have been compromised by trojans or viruses, to issue requests that simply load web pages or make database queries at the server. These requests are indistinguishable from legitimate requests, yet happen in such volumes that the server runs out of resources and legitimate users are denied service. We direct the readers to Michael Walfish's thesis [178] for the rationale behind why attackers mount such an attack. We call such DDoS attacks CyberSlam, after the first FBI case involving DDoS-for-hire [137]. Many DDoS extortion attacks [117], recent DDoS-for-hire attacks [68, 117, 137], and the attack on Estonian government websites [14–16] are all instances of CyberSlam.

Traditional DoS techniques are unable to defend against CyberSlam. Existing Intrusion Detection Boxes filter out malicious traffic only when such traffic is different in content from legitimate traffic [35, 133]. A possible solution would be to track all the big consumers at a web-server, perhaps by using login password pairs and metering consumption. Many sites do not use passwords or login information, and even when they do, passwords could be easily stolen from the hard disk of a compromised machine. Further, as we will show later the password checking mechanism itself is vulnerable to a denial-of-service attack.

In Chapter 2, we present a solution to CyberSlam called Kill-Bots. Kill-Bots uses Reverse Turing Tests that humans can easily solve but are as yet unsolvable by computers to separate

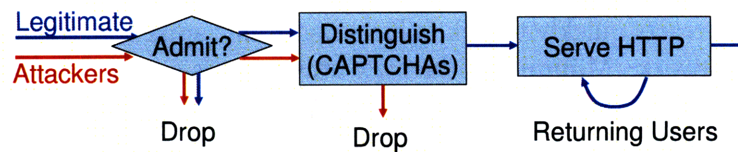


Figure 1-2 – Kill-Bots Queuing Model: KillBots distinguishes between Legitimate and Attacker Requests and provides Differentiated Service.

requests from legitimate users and attackers into two classes. For example, whenever the web server is overloaded it presents all users with a CAPTCHA [176], a graphical puzzle with text embedded in an image. Humans solve the CAPTCHA and are served the pages they request. Attacking robots on the other hand cannot solve the CAPTCHA and are denied access to content. Kill-Bots presents a HTTP cookie so that legitimate users who have solved puzzles get access for a short period of time without solving repeatedly. Fig. 1-2 presents a schematic of how Kill-Bots works.

Kill-Bots casts the solution as a resource management problem. How should a server divide its resources between authenticating arrivals and serving requests from already authenticated users? If a server uses all its resources serving puzzles, then it might run out of resources to serve legitimate users already in the system. If the server spends too little resources in serving puzzles, it might go idle because there are not enough legitimate users in the system. Kill-Bots drops requests from un-authenticated users and adapts its admission probability to authenticate just enough users to keep the server busy. Further, Kill-Bots does this without requiring any prior knowledge about server workloads, attack rates or attack arrival patterns.

Kill-Bots adapts to unexpected attacks by solving the above resource management problem in an online fashion. When the server becomes overloaded, Kill-Bots starts serving puzzles to all un-authenticated requests. Kill-Bots adaptively re-adjusts its admission probability, i.e., the fraction of requests to give puzzles out to, to efficiently manage resources during the attack. We have built a Kill-Bots prototype in Linux and shown via experiments over the Internet that compared to a server that does not use Kill-Bots, the prototype withstands attack rates two orders of magnitude higher, while maintaining response times around their values with no attack.

1.2 Robustness as a Learning Problem

We now shift focus to the significant internal complexity in enterprise and campus networks. Most existing enterprise management tools are box-centric and hence cannot help understand why a user sees bad performance. These tools focus on individual routers, servers, or links in the network. They raise alerts when anything goes wrong with anyone of these boxes, such as high-load at a server, too-many-drops at a link, yet offer no insights on which of these alerts causes the poor performance. Instead, of worrying about the individual components, we take an end-to-end perspective. Our approach is to *learn the implicit rules underlying communication in the network*; use these rules to build *functional representations* of how the dependent components are involved in a user activity and; *reason about most likely explanations of user's performance problems.*

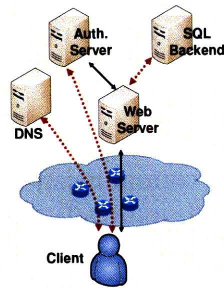


Figure 1-3 – Canonical Intranet Web Connection. We show the activities involved in fetching a web page from an intranet web server inside Microsoft. The dotted arrows represent actions that need not happen on every web request.

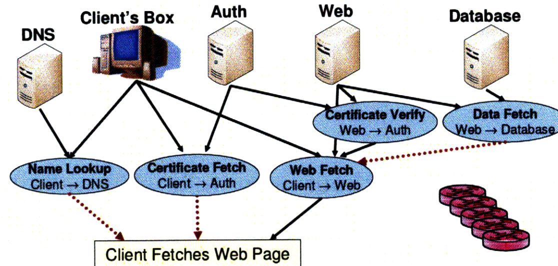


Figure 1-4 – Functional Representation of Dependencies. We show a simple *dependency graph* representation of this activity. For clarity, we omit network dependencies, i.e., the routers and switches between each of these components.

We address two specific challenges. First, it is not clear what the user's performance depends on. Let us take the canonical example of a user downloading a web-page from an intranet site (Figure 1-3). Before connecting to the web server, a client has to first lookup the server's name with DNS and fetch credentials from an Authentication server. Further before responding to the client, the web server verifies these credentials and may fetch relevant data from a database backend. Any one of these servers, or the routers and switches on the paths to these servers may be responsible for a user's poor performance. How can we automatically learn these dependencies? To further complicate the issue, the dependencies are distributed. The client only knows it needs to talk to DNS and Auth, it has no idea what the server does, at its backend, before responding to a request. Even worse, some dependencies are probabilistic. If the client has cached the previously looked up server's address (or user credentials), it need not repeat the lookup (or certificate fetch) every time it connects to the web-server. In Chapter 4, we present eXpose a technique to learn such dependencies from packet traces with no prior knowledge about the network.

Suppose that you knew precisely what the dependencies were, even then it is hard to figure out which of the dependencies causes the user's poor performance. For example, suppose the server is at 90% utilization and the router dropped a few packets, which of these is the most likely explanation for the user's long response time? Thus, the second challenge is to localize the causes of faults and performance problems. In Chapter 5, we present fault localization techniques that convert the discovered dependencies into probabilistic functional representations and apply inference techniques to reason about most likely explanations of faults.

Past work has advocated discovering dependencies. However, while several startups, like Relicore [33], Collation [11] and nLayers [29] perform "application dependency mapping" [17], not much is publicly known about their specific approaches. Some of these tools tend to work only for a few well known applications [18], or for applications that have pre-defined fingerpints [18]; others require customization to mine appropriate Windows Registry entries and application specific configuration [12] and still others focus on discovering which applications exist in the network [18]. Our dependency discovery is unique in terms of discovering dependencies without any prior knowledge about applications or servers in the network. Further, to the best of our knowledge, we are the first to couple dependency discovery with a probabilistic model that captures real-world dependencies and a fault local-

ization technique that uses this model to find causes of real problems in working networks.

1.2.1 Learning Communication Rules

In Chapter 4, we present eXpose, a tool that extracts significant communication rules in a network trace, without being told what to look for. A communication rule is an implicit relationship such as $Flow_i.new \Rightarrow Flow_j.new$ indicating that whenever a new $Flow_i$ connection happens, a new $Flow_j$ connection is likely to happen. For example, eXpose would deduce rules like a DNS connection often precedes new connections, an AFS client talks to the root-server (port 7003) before picking files up from the appropriate volume-servers (port 7000), an End-Point-Mapper RPC call precedes mail fetch from Microsoft Exchange Servers, and viruses such as the mySQLbot probe flood the entire network looking for vulnerable mySQL servers.

The key insight, here, is simple– if a group of flows consistently occurs together, the group is likely dependent. Of course, correlation does not always imply dependence, but this is the best one can do lacking knowledge of the applications and the network layout. The challenge lies in applying this insight to a network trace where millions of flows show up every few hours. To do this, eXpose selectively biases the potential rules it evaluates and does not evaluate rule types that are unlikely to yield useful information. Second, eXpose abstracts away extraneous flow details to make useful patterns more discernible. Third, eXpose uses an appropriate statistical measure to score the candidate rules and mines efficiently. Finally, eXpose aggregates the discovered rules into a small number of useful clusters that an administrator can corroborate and use.

Not all dependencies are visible at the granularity of a flow. For example, suppose whenever a client talks to a sales server, the server fetches data from a backend database. Yet no individual client accesses the server often enough to create a significant rule. To capture such rules that are otherwise unidentifiable, eXpose introduces templates that systematically abstract away parts of flows. For example, one of our templates replaces the client's IP address with a wild-card character creating a *generic* whenever any client talks to the sales server. Leveraging these generics, eXpose searches for rules like $* : Sales \Rightarrow Sales : Database$, meaning whenever *any one* of the clients talks to the Sales server, the Sales server talks to its Database. We show how to apply templates without any prior knowledge about the hosts involved in the trace and find that these templates improve the expressiveness of our rules considerably.

Deployments of eXpose on the server facing links of two edge networks, in the CSAIL lab at MIT and in the Microsoft Research for a couple of months, show that eXpose discovered rules for the major applications such as email, web, file-servers, instant messengers, peer-to-peer and multimedia distribution in each of the two edge networks. Further, eXpose discovered rules for enterprise configurations and protocols that are deployed and used in mainstream operating systems that we did not know existed, such as Nagios [26] monitors and link-level multicast name resolution [23].

1.2.2 Localizing Faults and Performance Problems

In Chapter 5, we show how to construct a functional representation of the dependencies discovered above and present algorithms that reason about most likely explanations for faults.

Our functional representation, called the Inference Graph, is a labeled, directed graph whose goal is to capture which components affect which user actions and in what manner. Figure 1-4 shows a much simplified example for a single user activity. Edges point from a *parent* node to a *child* node, denoting that the child node is dependent on all of the parent nodes. The graph encodes a joint probability distribution function for *how* the state of a child node is governed by the states of its parents. For example, one simple function is *or*, indicating that whenever any one of the parents is faulty, the child will be faulty. See Chapter. 5 for how we capture unequal contributions from parents—$Parent_1$ (e.g., the connection to the web server) affects the $Child$ (e.g., the client's browsing behavior) more than $Parent_2$ (e.g., the connection to the DNS server), and also more complicated real-world dependencies such as failover and selection.

Nodes that have no parents, shown at the top of the figure, are called root-cause nodes and correspond to independently failing entities, e.g., individual servers and links. Nodes that have no children, shown at the bottom of the figure, are called observation nodes and correspond to directly measurable events, e.g., a user's response time to the web server. The value of such a representation is that given observations of multiple user activities, we can figure out the most likely explanation for the observation, i.e., which combination of root-causes is likely faulty.

How do we obtain the most likely explanation for an observation? Probabilistic inference techniques [129, 135] are a potential solution. Given the states of certain nodes in a probabilistic graph, in this case which users see what performance, inference algorithms compute the most likely state assignment for the remaining nodes in the graph, in this case which root-causes are likely faulty. Unfortunately traditional inference algorithms do not scale to the size of the graphs we encounter in practice. Microsoft's Enterprise Network has tens of thousands of clients, servers and routers, with many shared dependencies such as routers on the paths to data-centers and services such as DNS and WINS that are common across multiple activities. Such high degree of sharing is beneficial. Observations of one user reading a web page at some server can help debug why another user takes too long to fetch a file from a networked file server–for example, that none of the shared components are to be blamed. Yet the scale of real networks, and the high degree of shared dependencies complicates the graph. Our inference graphs are both big and highly connected and traditional inference techniques scale particularly poorly on highly connected graphs.

In a field deployment, within the Microsoft Research Network, we show that our approximate inference algorithm, is effective at identifying performance problems and narrows down the root-cause to a small number of suspects. Further, we show via simulations that our algorithm is robust to noise in the Inference Graph and its multi-level probabilistic model helps localize faults more accurately than prior approaches that use a deterministic model.

Chapter 2

Defending Against DDoS Attacks That Mimic Flash Crowds

We first focus on protecting network resources from malicious traffic. Denial of service attacks are increasingly mounted by professionals to extort money or obtain commercial advantages [137]. Attackers use aggressive worms that can infect up to 30,000 new machines per day [93, 165] and group these infected zombies into botnets. Botnets of thousands of compromised machines are reported to be available for renting by the hour on IRC channels [93, 118, 167]. To circumvent detection, botnet equipped attackers are moving away from pure bandwidth floods. Instead, they profile the victim server and launch stealthy DoS attacks that mimic legitimate Web browsing behavior. In particular, [137] reports that a Massachusetts businessman "paid members of the computer underground to launch organized, crippling DDoS attacks against three of his competitors". The attackers used Botnets of more than 10,000 machines. When the simple SYN flood failed, they launched an HTTP flood, downloading many large images from the victim server. "At its peak, the onslaught allegedly kept the victim company offline for two weeks." In another instance, attackers ran a massive number of queries through the victim's search engine, bringing the server down [137]. We call such attacks that target higher layer server resources like sockets, disk bandwidth, database bandwidth and worker processes [66, 117, 137], Cyberslam, after the first FBI case involving DDoS-for-hire [137]. The MyDoom worm [66], many DDoS extortion attacks [117], and recent DDoS-for-hire attacks are instances of Cyberslam [68, 117, 137].

2.1 Overview

Kill-Bots mitigates Cyberslam by casting protection as a resource allocation problem. Current servers allow each client to compete for a share of resource by merely sending a request. The more requests a client can issue, the larger the share of server resources the client can use up and so, a client's consumption is only limited by his own computational and network link capacity. If an attacker has enough machines, this service model lets an attacker deny service to legitimate users by simply overwhelming the server with requests.

Instead, Kill-Bots allows access only upon solving a Reverse Turing Test and distributes resources evenly among everyone who solves a test. A Reverse Turing Test, such as a graphi-

cal CAPTCHA [176], is easily solved by humans, but cannot be solved by machines. Hence, a client is now limited by the rate at which he can solve CAPTCHAs – a human cost, that we believe is more scarce (and expensive) than computational or network bandwidth.

We believe that this idea – requiring human work prior to granting resources – will be a recurrent theme going forward. Predating our work, free email providers such as Hotmail used CAPTCHAs to ensure that only humans can register accounts. Postdating our work, blogging sites and wikis require solving a CAPTCHA before creating an entry or commenting on entries.

Kill-Bots is fundamentally different in its use of reverse turing tests. Rather than protecting long-lived resources like email accounts or blog posts, Kill-Bots use CAPTCHAs to protect fine-grained resources, i.e., to decide whether or not to respond to a HTTP request. So how would an attacker react? He would simply fetch so many CAPTCHAs or hog so much server state that the CAPTCHA delivery and answer checking mechanism is overwhelmed.

The first major contribution of Kill-Bots is that it protects the CAPTCHA authentication mechanism that is itself vulnerable to a denial of service attack. Kill-Bots sends a puzzle from within the kernel and without giving an unauthenticated client access to TCBs or socket buffers. This makes puzzle delivery quick–few resources have to be allocated, no context-switches or data copies to user space are needed and no new web-server processes or threads need to be spawned. Even better, puzzle delivery is stateless, i.e., all state required for authentication is stored in the messages themselves. Kill-Bots modifies the server's TCP stack so as to send a one or two packet puzzle at the end of the TCP handshake without maintaining any connection state, and while preserving TCP congestion control semantics.

Reverse Turing Tests have a bias against human users who cannot or will not solve them, and of course against legitimate machine generated requests such as web crawlers. A user study that required external visitors of our group's web page to solve CAPTCHAs before they could view the page, showed that over 40% of the presented puzzles went un-answered. So, merely by forcing a website to use reverse turing tests, an attacker successfully denies service to 40% of the site's business.

The second major contribution of Kill-Bots is to mitigate the bias of reverse turing tests against human users and legitimate machines who don't answer the reverse turing test. Our solution builds on a simple observation. Legitimate users, both human and web crawlers, either solve the puzzle, or try to reload a few times and, if they still cannot access the server, give up and may come back later. In contrast, the attacker's machines cannot solve the puzzle and have one of two alternatives. They can mimic legitimate users and leave after a couple of tries, in which case there is no longer an attack. Or in order to continue the attack, they can keep sending new requests without solving the puzzles given to them. Kill-Bots uses this difference in behavior to separate legitimate users and attackers. Kill-Bots uses a Bloom filter to track the number of un-answered puzzles a machine has received. Legitimate users will only have a few, whereas the longer the attack persists, the more the number of unanswered puzzles a zombie would have received. Once a client receives more than a configurable threshold (e.g., 32) of puzzles without answering, Kill-Bots discards all further requests from the client.

A couple of points are worth noting. First, Kill-Bots protects against an attack whose success depends on mimicking legitimate users– the attacker requests are indistinguishable from legitimate ones in content, by a solution technique that makes mimicking legitimate

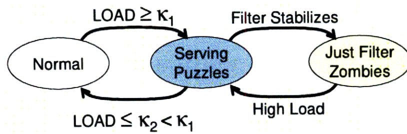


Figure 2-1 – Kill-Bots transitions between normal, serving puzzles and relying only on the zombie filter.

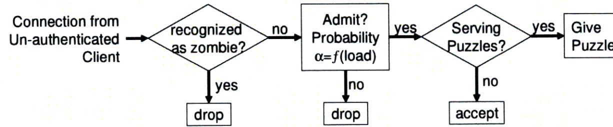


Figure 2-2 – Kill-Bots Overview. Note that new clients are authenticated with a puzzle only when Kill-Bots is in the **serving puzzles** stage.

users result in there being no attack– each attacker machine will be limited to make no more requests than a legitimate user who does not answer puzzles hits the reload button. Second, building on the resource allocation analogy, Kill-Bots separates users into two classes – (a) those who either answer reverse turing tests or if they don't answer the test, issue new requests at a small rate on par with humans refreshing web pages and (b) those who do not answer tests and yet issue requests at a very high rate. Requests from users in the first class compete fairly for service, whereas those from requests in the second class are dropped.

How should a server divide its resources between these two classes, i.e., between serving already authenticated users and authenticating new arrivals? This is general problem for any authentication system. Devoting excess resources to authentication might leave the server unable to fully serve the authenticated clients, and hence, wastes server resources on authenticating new clients that it cannot serve. On the other hand, devoting excess resources to serving authenticated clients without authenticating and taking in new clients will lead to idle periods wherein there are no clients to serve.

The third major contribution of Kill-Bots is to manage resources better via admission control. The strategy is simple – when the server is overloaded, it authenticates and admits exactly as many new clients in each time window as necessary for the server to be completely utilized. To do this, Kill-Bots computes the admission probability α that maximizes the server's goodput (i.e., the optimal fraction of new clients that are to be authenticated). Kill-Bots also provides a controller that allows the server to converge to the desired admission probability using simple measurements of the server's utilization. Admission control is a standard mechanism for combating server overload [94, 175, 180], but Kill-Bots examines admission control in the context of malicious clients and connects it with client authentication.

Figures 2-1 and 2-2 summarize Kill-Bots. Kill-Bots only kicks in when the server is overloaded. When a new connection arrives, it is first checked against the list of zombies, i.e., addresses that have exceeded the limit on unanswered puzzles. If the IP address is not recognized as a zombie, Kill-Bots admits the connection with probability $\alpha = f(load)$. If the list of zombies is still under flux, admitted connections are served a graphical puzzle. If the client solves the puzzle, it is given a Kill-Bots HTTP cookie which allows its future connections, for a short period, to access the server without being subject to admission control and without having to solve new puzzles. When the set of detected zombies stabilizes (i.e., the filter does not learn any new addresses crossing the threshold), Kill-Bots no longer issues puzzles; admitted connections are immediately given a Kill-Bots HTTP cookie, thereby allowing legitimate users access to the server even when they don't solve graphical puzzles.

The value of any protection technique boils down to reducing the cost to a server for each unit of resource that an attacker has. If the server were to only use CAPTCHAs, its

cost per attacker request is to serve the puzzle–two kilobytes of static content. Now, this isn't a big savings if the website was only serving small images (.gif, .jpeg). But, if the server has dynamic content that involves computation, or content that requires a database, or large files (OS images$\approx$ $1GB$, .pdf publications$\approx$ $1MB$) or video content, as most websites of interest do, serving a small puzzle instead of the attacker's request is a substantial saving. Moving along, if the web server also limits the number of un-answered puzzles per machine, it does much better. For e.g., suppose that Kill-Bots allows each attacking IP 32 un-answered puzzles, then even when the web-server is just one x86 2GHz P4 box, Kill-Bots can detect and block an attack from botnets of tens of thousands of nodes within a couple of minutes.

To summarize, the key aspects of Kill-Bots are:

- **Kill-Bots addresses graphical tests' bias against users who are unable or unwilling to solve them.** In contrast to prior work that ignores the bias against blind and inexperienced humans [127], Kill-Bots is the first system to employ graphical tests to distinguish humans from automated zombies, while limiting their negative impact on legitimate users who cannot or do not want to solve them.
- **Kill-Bots makes puzzle based authentication quick and stateless.** All state required for authentication is wrapped into the message exchanges. This protects the authentication mechanism from being itself vulnerable to denial of service.
- Kill-Bots improves performance, regardless of whether server overload is caused by DDoS attacks or true Flash Crowds, making it the **first system to address both DDoS and Flash Crowds within a single framework**. This is an important side effect of using admission control, which allows the server to admit new connections only if it can serve them.

We implemented Kill-Bots in the Linux kernel and evaluated it in the wide-area network using PlanetLab. On a standard 2GHz Pentium IV machine with 1GB of memory and 512KB L2 cache running a mathopd [24] web-server on top of a modified Linux 2.4.10 kernel, Kill-Bots serves a graphical test in 31μs; tracks the number of un-answered puzzles of a machine in less than 1μs; and can survive DDoS attacks of up to 6000 HTTP requests per second without affecting response times. Compared to a server that does not use Kill-Bots, our system survives attack rates two orders of magnitude higher, while maintaining response times around their values with no attack. Furthermore, in our Flash Crowds experiments, Kill-Bots delivers almost twice as much goodput as the baseline server and improves response times by two orders of magnitude. These results are for an event driven OS that relies on interrupts. The per-packet cost of taking an interrupt is fairly large $\approx$ $10\mu s$ [113]. We expect better performance with polling drivers [125].

2.2 Threat Model

Kill-Bots aims to improve server performance under Cyberslam attacks, which mimic legitimate Web browsing behavior and consume higher layer server resources such as CPU, memory, database and disk bandwidth. Prior work proposes various filters for bandwidth floods [50, 90, 109, 119]; Kill-Bots does not address these attacks. Attacks on the server's DNS entry or on the routing entries are also outside the scope of this chapter.

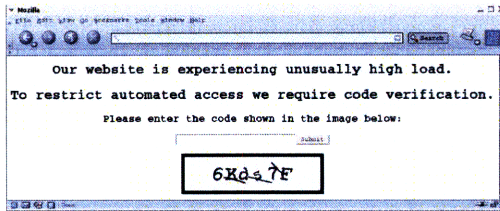


Figure 2-3 – Screen-shot of a graphical puzzle

```
<html>
  <form method = "GET" action="/validate">
      <img src = "PUZZLE.gif">
      <input type = "password" name = "ANSWER">
      <input type = "hidden" name = "TOKEN" value = "[]">
  </form>
</html>
```

Figure 2-4 – HTML source for the puzzle

In our threat model, an attacker may control an arbitrary number of machines widely distributed across the Internet, giving him access to significant CPU and memory resources in total. But, the attacker does not have physical access to the server, cannot sniff packets on the server's local network or on a link that carries traffic for a large number of legitimate users. Finally, the zombies cannot solve the graphical test and the attacker is not able to concentrate a large number of humans to continuously solve puzzles.

2.3 Separating Legitimate Users from Attackers

During periods of overload, Kill-Bots authenticates clients before granting them service. A Kill-Bots Web-server can be in one of the three modes, NORMAL, SERVE_PUZZLES, or JUST_FILTER as shown in Figure 2-1. Kill-Bots uses different authentication mechanisms in the latter two stages.

2.3.1 Activating the Authentication Mechanism

By default, a server begins in the NORMAL mode. When the Web server perceives resource depletion beyond an acceptable limit, κ_1, it shifts to the SERVE_PUZZLES mode. In this mode, every new connection has to solve a graphical test before obtaining service. When the user correctly solves the test, the server grants access for the duration of an HTTP session. Connections that began before the server switched to the SERVE_PUZZLES mode continue to be served normally until they terminate. However, the server will timeout these connections if they last longer than a certain duration (our implementation uses 5 minutes). The server transitions back to NORMAL mode when the server load goes down to its normal range and crosses a particular threshold $\kappa_2 < \kappa_1$. Kill-Bots estimates server load using an exponentially weighted moving average. The values of κ_1 and κ_2 will vary depending on the normal server load. For example, if the server is provisioned to work with 40% utilization, then one may choose $\kappa_1 = 70\%$ and $\kappa_2 = 50\%$.

A couple of points are worth noting. First, the server behavior is unchanged in the NORMAL mode, and thus the system has no overhead in the common case of no attack. Second, a stop-and-go attack that forces Kill-Bots to switch back and forth between these two modes is no worse than an equivalent attack that happens in one shot because the cost for switching is minimal. The only potential expense is the need to timeout very long connections that started in the NORMAL mode. Long connections that started in an earlier SERVE_PUZZLES mode are not timed out as their users have already been authenticated.

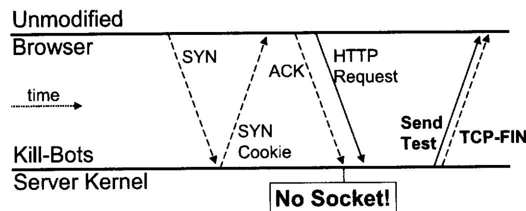


Figure 2-5 – Kill-Bots modifies server's TCP stack to send tests to new clients without allocating a socket or other connection resources.

2.3.2 CAPTCHA-Based Authentication

In the SERVE_PUZZLES mode, Kill-Bots authenticates clients using graphical tests, i.e., CAPTCHAs [176]. The basic operation is simple. When a new HTTP request arrives, Kill-Bots sends out a graphical test such as the one shown in Figure 2-3. Humans can solve these graphical tests easily but machines cannot. If the client responds, Kill-Bots validates the answer and grants access. It would be inconvenient if legitimate users had to solve a puzzle for every HTTP request or every TCP connection. Hence, Kill-Bots gives users who solve the test correctly a HTTP Cookie, which allows the user to re-enter the system for some time, T (our implementation uses $T = 30\text{min}$).

Two challenges remain, however. First, if puzzle delivery and answer validation takes much time or lets an attacker hog state on the server, authentication itself will be vulnerable to denial of service. Second, the amount of service a user obtains should be proportional to the human work he has expended. For example, solving one puzzle and replaying the answer packet over and over should not let the user obtain more than a puzzle's worth of resources.

(a) Quick and Stateless Puzzle Delivery: Kill-Bots sends a graphical test and validates the corresponding answer without allocating any TCBs, socket buffers, or worker processes on the server. We achieve this by a minor modification to the server TCP stack. As shown in Figure 2-5, similarly to a typical TCP connection, a Kill-Bots server responds to a SYN packet with a SYN cookie. The client receives the SYN cookie, transmits a SYNACKACK and the first data packet that usually contains the HTTP request. Unlike a typical connection, the Kill-Bots kernel does not create a new socket upon completion of the TCP handshake and instead discards the SYNACKACK packet. Since the first data packet from the client repeats the same acknowledgment sequence number as the SYNACKACK, Kill-Bots can check that the handshake succeeded; thereby effectively deferring creation of state until more is known about the client.

When the server receives the client's data packet, it checks whether it is a puzzle answer. A puzzle answer has a HTTP request of the form *GET /validate?answer=_something_*. If the packet is not an answer, the server sends out a new graphical test, embedded in an HTML form (see Figure 2-4). Our implementation uses CAPTCHA images that fit within one or two packets ($< 3KB$). After sending out the puzzle, the server immediately closes the connection by sending a FIN packet. When the user answers the graphical test, the HTML form (Figure 2-4) that wraps around the puzzle generates a HTTP request reporting the answer to the server. All the information required to validate this answer, such as which puzzle the answer corresponds to, is embedded as hidden fields in the form and is reported back along with the answer. If the answer is valid, the kernel creates a socket and delivers the request to the application server.

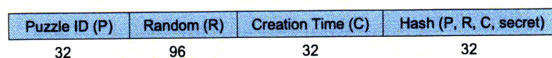


Figure 2-6 – Kill-Bots Token

We note that the above scheme preserves TCP congestion control semantics, does not require modifying the client software, and stows away all the state required for authentication within the exchanged messages. By authenticating new arrivals from within the kernel, Kill-Bots avoids context switches, data copies and spawning of new processes, thereby reducing the cost of authentication. Further, resource exhaustion attacks such as when an attacker establishes sockets but does not send any data [69, 150], or hogs server sockets by never sending a FIN [69, 150] are not possible with Kill-Bots.

(b) Providing Service Proportional to Human Effort: Kill-Bots leverages cryptographic support to provide proof of human effort. When the Kill-Bots server issues a puzzle, it creates a Token as shown in Figure 2-6. The token consists of a 32-bit puzzle ID P, a 96-bit random number R, the 32-bit creation time C of the token, and a 32-bit collision-resistant hash of P, R, and C along with the server secret.

The token is embedded (in hidden fields) in the HTML form that wraps around the puzzle (Figure 2-6). The client's browser reports back the Kill-Bots token to the server along with the user's answer. Besides verifying the answer to the test, the server validates the token by recomputing the hash. Further, the server checks the Kill-Bots token to ensure the token was created no longer than 4 minutes ago. The server creates a Kill-Bots HTTP cookie and grants access only when all these checks are successful. The cookie is created from the token by updating the token creation time and recording the token in the table of valid Kill-Bots cookies. Subsequent user requests arrive accompanied with this cookie and are allowed only when the cookie is valid.

Unforgeable: Can a user forge a cookie and therefore gain access without solving a puzzle? The Kill-Bots token, and cookie are both unforgeable and tamper-proof. The random number R in the cookie is generated at random from a vast space (2^{96} possibilities), making it unlikely that an attacker can guess a valid number. Even better, the cookie contains a hash over all its entries and a private *secret* string making it harder for an attacker to forge, and also prevents tampering with entries in the cookie.

No Benefit in Replaying: An attacker may solve one puzzle and attempt to replay the "answer" packet to obtain many Kill-Bots cookies. Recall that when Kill-Bots issues a cookie for a valid answer, the cookie is an updated form of the token (Fig 2-6). Hence, replaying the "answer" yields the same cookie. The cookie also contains a secure hash of the time it was issued and is only valid during a certain time interval. If an adversary tries to replay a session cookie outside its time interval it gets rejected.

No Benefit in Copying: What if the attacker solves one graphical test and distributes the HTTP cookie to a large number of bots? Kill-Bots introduces a notion of per-cookie fairness to make service proportional to the amount of human work expended. Each correctly answered graphical test allows the client to execute a maximum of 8 simultaneous HTTP requests. Distributing the cookie to multiple zombies makes them compete among themselves for these 8 connections. Most legitimate web browsers open no more than 8 simultaneous connections to a single server [95].

We defer discussion of how Kill-Bots addresses other attacks to Section 2.5. At this point,

it suffices to say that Kill-Bots leverages cryptography to ensure that clients obtain service proportional to the amount of human work they expend in solving graphical tests.

2.3.3 Authenticating Users Who Do Not Answer CAPTCHAs

An authentication mechanism that relies solely on CAPTCHAs has two disadvantages. First, the attacker can force the server to continuously send graphical tests, imposing an unnecessary overhead on the server. Secondly, and more importantly, humans who are unable or unwilling to solve CAPTCHAs may be denied service.

To deal with this issue, Kill-Bots distinguishes legitimate users from zombies by their reaction to the graphical test rather than their ability to solve it. When presented with a graphical test, legitimate users may react as follows: (1) they solve the test, immediately or after a few reloads; (2) they do not solve the test and give up on accessing the server for some period, which might happen immediately after receiving the test or after a few attempts to reload. The zombies have two options; (1) either imitate human users who cannot solve the test and leave the system after a few trials, in which case the attack has been subverted, or (2) keep sending requests though they cannot solve the test. However, by continuing to send requests without solving the test, the zombies become distinguishable from legitimate users, both human and machine.

In the SERVE_PUZZLES mode, Kill-Bots issues puzzles and tracks how often a particular IP address has failed to solve a puzzle. It maintains a Bloom filter [60] whose entries are 8-bit counters. Whenever a client is given a graphical puzzle, its IP address is hashed and the corresponding entries in the Bloom filter are incremented. In contrast, whenever a client comes back with a correct answer, the corresponding entries in the Bloom filter are decremented. Once all the entries corresponding to an IP address reach a particular threshold χ (our implementation uses χ=32), the server drops all further packets from that IP.

When the attack starts, the Bloom filter has no impact and users are authenticated using graphical puzzles. Yet, as the zombies make more requests, receive puzzles and do not answer them, their counters pile up. Once a client has χ unanswered puzzles, it will be blocked. When the server notices that the Bloom filter is not catching any new zombie IPs, i.e., no new IPs have crossed the χ threshold recently, it no longer issues puzzles. Instead it relies solely on the Bloom filter to block requests from the zombie clients. We call this the JUST_FILTER mode (see Figure 2-1). If subsequently the load increases, the server resumes issuing puzzles.

In our experiments, the Bloom filter detects and blocks a sustained attack from upto tens of thousands of zombies within a few minutes. In general, the higher the attack rate, the faster the Bloom filter can detect the zombies and block their requests. A full description of the Bloom filter is in §2.6.

2.3.4 Criterion to Distinguish Legitimate Users and Zombies

Precisely stated, Kill-Bots' litmus test for legitimacy is the following: *Any client that either answers a graphical test correctly, or if it does not, makes no more than the number of requests a human who does not answer the graphical test would by hitting the "reload" button is considered legitimate.* The latter includes legitimate tools that issue machine generated HTTP requests such as web crawlers and also humans who don't or cannot answer graphical tests.

Variable	Description
λ_s	Arrival rate of legitimate HTTP sessions
λ_a	Arrival rate of attacking HTTP requests
α	Admission Probability. Drop probability=$1-\alpha$.
$\frac{1}{\mu_d}$	Mean cost to pick a request and optionally drop it
$\frac{1}{\mu_p}$	Mean cost to serve a puzzle
$\frac{1}{\mu_h}$	Mean cost to serve an HTTP request
q	Mean # of requests per legitimate session
ρ_p	Fraction of server time spent in authenticating clients
ρ_h	Fraction of server time spent in serving authenticated clients
ρ_d	Fraction of server time spent dropping requests
ρ_i	Fraction of time the server is idle

Table 2.1 – Variables used in the analysis

Attacker Strategies: But, what if zombies mimic crawlers, i.e., they issue no more requests than a human who does not answer the graphical test? Our experiments show that attack rates of about 6000 requests per second, each of which obtains a graphical test, are necessary to overwhelm a website that is powered by just one 2GHz machine. This means that an attacker with a bot-net of $100,000$ machines can make $\chi = 32$ requests from each bot before the bloom filter recognizes and blocks the bot. Using the best possible strategy, if he issues exactly 6000 requests each second, the attack would persist for 9 minutes of attack time. A website with a little more computational power, say a couple of servers, will learn all zombie addresses much sooner.

2.4 Managing Resources Efficiently

A web site that performs authentication to protect itself from DDoS has to divide its resources between authenticating new clients and servicing those that have already been authenticated. Devoting excess resources to authentication might leave the server unable to fully service the authenticated clients, thereby wasting resources on authenticating new clients that it cannot serve. On the other hand, devoting excess resources to serving authenticated clients may cause the server to go idle if it does not authenticate enough new clients. Thus, there is an optimal authentication probability, α^*, that maximizes the server's goodput.

2.4.1 Results of the Analysis

In Appendix A., we model a server that implements an authentication procedure in the interrupt handler. This is a typical location for packet filters and kernel firewalls [27, 122, 149]; it allows dropping unwanted packets as early as possible. Our model is fairly general and is independent of how the authentication is performed. The server may be checking certificates, verifying their passwords, or asking them to solve a puzzle.

The model lets us compute the optimal probability with which new clients should be authenticated. Below, we summarize these results and discuss their implications. Table 2.1 describes our variables.

When a request from an unauthenticated client arrives, the server authenticates it with

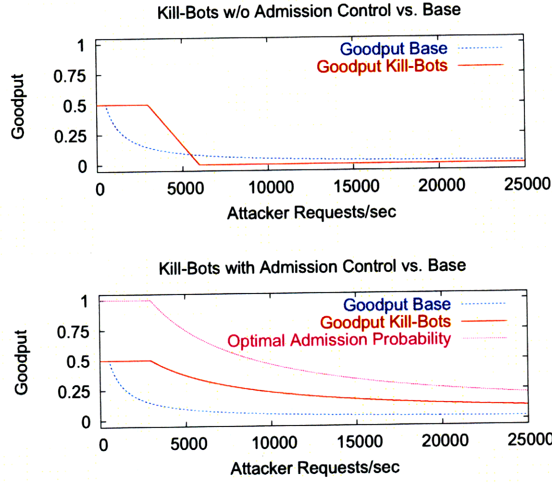


Figure 2-7 – Comparison of the goodput of a base/unmodified server with a server that uses authentication only (TOP) and a server that uses both authentication & admission control (BOTTOM). Server load due to legitimate requests is 50%. The graphs show that authentication improves goodput under attack and is even better with admission control, particularly at high attack rates.

probability α and drops it with probability $1 - \alpha$. The optimal value of α is the value that maximizes the server's goodput, i.e., the CPU time spent on serving HTTP requests. This is:

$$\alpha^* = \min\left(\frac{\mu_p}{(Bq+1)\lambda_s + \lambda_a}, 1\right), \quad \text{where } B = \frac{\mu_p}{\mu_h}, \tag{2.1}$$

where λ_a is the attack request rate, λ_s is the legitimate users' session rate, $\frac{1}{\mu_p}$ is the average time taken to serve a puzzle, $\frac{1}{\mu_h}$ is the average time to serve an HTTP request, and q is the average number of requests in a session. This yields an optimal server goodput of,

$$\rho_g^* = \min\left(\frac{q\lambda_s}{\mu_h}, \frac{Bq\lambda_s}{(Bq+1)\lambda_s + \lambda_a}\right). \tag{2.2}$$

In comparison, a server that does not use authentication has goodput:

$$\rho_g^b = \min\left(\frac{q\lambda_s}{\mu_h}, \frac{q\lambda_s}{q\lambda_s + \lambda_a}\right). \tag{2.3}$$

To compare the two approaches, note that authenticating new users costs much less than serving them, i.e., $\mu_p >> \mu_h$. Hence, $B >> 1$, and the server with authentication can survive attack rates that are B times larger without loss in goodput.

Also, compare the optimal goodput, ρ_g^*, with the goodput of a server that implements authentication without admission control (i.e., $\alpha = 1$) given by:

$$\rho_g^a = \min\left(\frac{q\lambda_s}{\mu_h}, \max\left(0, 1 - \frac{\lambda_a + \lambda_s}{\mu_p}\right)\right). \tag{2.4}$$

For large attack rates, $\lambda_a > \mu_p - Bq\lambda_s$, the goodput of the server with no admission control decreases linearly with the attack rate and in fact goes to zero at $\lambda_a = \mu_p - \lambda_s$. With

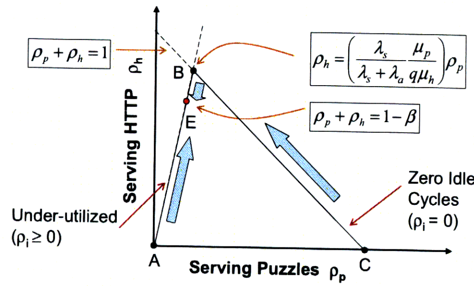


Figure 2-8 – Phase plot showing how Kill-Bots adapts the admission probability to operate at a high goodput

admission control, the goodput of the server decreases much more gracefully.

Figure 2-7 illustrates the above results. A Pentium-IV, 2.0GHz 1GB RAM, machine serves 2-pkt puzzles at a peak rate of 6000/sec ($\mu_p = 6000$). Assume, conservatively, that each HTTP request fetches 15KB files ($\mu_h = 1000$), that a user makes 20 requests in a session ($q = 20$) and that the normal server load is 50%. By substituting these values in Equations 2.2, 2.3 and 2.4, Figure 2-7 compares the goodput of a server that does not use authentication (base server) with the goodput of a server that uses authentication only ($\alpha = 1$), and with a server that uses both authentication and admission control ($\alpha = \alpha^*$). The top graph shows that authentication improves server goodput under attack. The bottom graph shows the additional improvement from admission control.

2.4.2 Adaptive Admission Control

How can we make a server function at the optimal admission probability? Computing α^* from Eq. 2.1 requires values for parameters that are typically unknown at the server and change over time, such as the attack rate, λ_a, the legitimate session rate, λ_s, and the number of requests per session, $\frac{1}{q}$.

To deal with the above difficulty, Kill-Bots uses an adaptive scheme. Based on simple measurements of the server's idle cycles, Kill-Bots adapts the authentication probability α to gradually approach α^*. Let ρ_i, ρ_p, ρ_h denote the fractions of time the server is idle, admitting arrivals, serving puzzles and serving HTTP requests respectively. We ignore ρ_d here for simplicity. We have:

$$\rho_h + \rho_p + \rho_i = 1. \tag{2.5}$$

If the current authentication probability $\alpha < \alpha^*$, the authenticated clients are too few and the server will spend a fraction of its time idle, i.e., $\rho_i > 0$. In contrast, if $\alpha > \alpha^*$, the server authenticates more clients than it can serve and $\rho_i = 0$. The optimal probability α^* occurs when the idle time transitions to zero. Thus, the controller should increase α when the server experiences a substantial idle time and decrease α otherwise.

However, the above adaptation rule is not as simple as it sounds. We use Figure 2-8 to show the *relation* between the fraction of time spent on authenticating clients ρ_p and that spent serving HTTP requests ρ_h. The line labeled "Zero Idle Cycles" refers to the states in which the system is highly congested $\rho_i = 0 \rightarrow \rho_p + \rho_h = 1$. The line labeled "Underutilized" refers to the case in which the system has some idle cycles, i.e., $\alpha < \alpha^*$. In this case, a fraction α of all arrivals are served puzzles. The average time to serve a puzzle is $\frac{1}{\mu_p}$. Thus,

the fraction of time the server serves puzzles is $\rho_p = \alpha\frac{\lambda_s+\lambda_a}{\mu_p}$. Further, an α fraction of legitimate sessions have their HTTP requests served. Thus, the fraction of time the server serves HTTP is $\rho_h = \alpha\frac{q\lambda_s}{\mu_h}$, where $\frac{1}{\mu_h}$ is the per-request average service time, and q is the average number of requests in a session. Consequently,

$$\forall\alpha < \alpha^* : \quad \rho_h = \left(\frac{q\lambda_s}{\lambda_s + \lambda_a}\frac{\mu_p}{\mu_h}\right)\rho_p,$$

which is the line labeled "Underutilized" in Figure 2-8. As the fraction of time the system is idle ρ_i changes, the system state moves along the solid line segments A→B→C. Ideally, one would like to operate the system at point B which maximizes the system's goodput, $\rho_g = \rho_h$, and corresponds to $\alpha = \alpha^*$. However, it is difficult to operate at point B because the system cannot tell whether it is at B or not; all points on the segment B–C exhibit $\rho_i = 0$. It is easier to stabilize the system at point E where the system is slightly underutilized because small deviations from E exhibit a change in the value of ρ_i, which we can measure. We pick E such that the fraction of idle time at E is $\beta = \frac{1}{8}$.

Next, we would like to decide how aggressively α can be adapted. Substituting the values of ρ_p and ρ_h from the previous paragraph in Eq. 2.5 yields:

$$\forall\alpha < \alpha^* : \quad \alpha\left(\frac{\lambda_a + \lambda_s}{\mu_p} + \frac{q\lambda_s}{\mu_h}\right) = 1 - \rho_i.$$

Hence, $\forall\alpha[t], \alpha[t+\tau] < \alpha^*$:

$$\frac{\alpha[t+\tau]}{\alpha[t]} = \frac{1-\rho_i[t+\tau]}{1-\rho_i[t]} \Rightarrow \frac{\Delta\alpha}{\alpha[t]} = \frac{\Delta\rho_i}{1-\rho_i[t]},$$

where $\alpha[t]$ and $\alpha[t+\tau]$ correspond to the values at time t and τ seconds later. Thus, every $\tau = 10s$, we adapt the admission probability according to the following rules:

$$\Delta\alpha = \begin{cases} +\gamma_1\alpha\frac{\rho_i-\beta}{1-\rho_i}, & \rho_i \geq \beta \\ -\gamma_2\alpha\frac{\beta-\rho_i}{1-\rho_i}, & 0 < \rho_i < \beta \\ -\gamma_3\alpha. & \rho_i = 0 \end{cases} \tag{2.6}$$

where γ_1, γ_2, and γ_3 are constants, which Kill-Bots set to $\frac{1}{8}$, $\frac{1}{4}$, and $\frac{1}{4}$ respectively. The above rules move α proportionally to how far the system is from the chosen equilibrium point E, unless there are no idle cycles. In this case, α is decreased aggressively to go back to the stable regime around point E. Appendix C. contains a stability analysis of this controller.

2.5 Security Analysis

This section discusses how Kill-Bots' handles attacks from a determined adversary.

(a) Socially Engineered Attack on CAPTCHAs: In a socially-engineered attack, the adversary tricks a large number of humans to solve puzzles on his behalf. Recently, spammers em-

ployed this tactic to bypass graphical tests that Yahoo and Hotmail use to prevent automated creation of email accounts [31]. The spammers ran a porn site that downloaded CAPTCHAs from the Yahoo/Hotmail email creation Web page, forced its own visitors to solve these CAPTCHAs before viewing porn, and used the answers to create new email accounts.

Kill-Bots is much more resilient to socially engineered attacks. In contrast to email account creation where the client is given an ample amount of time to solve the puzzle, puzzles in Kill-Bots expire 4 minutes after they have been served. Thus, the attacker cannot accumulate a store of answers from human users to mount an attack. Indeed, the attacker needs a *continuous* stream of visitors to his site to be able to sustain a DDoS attack. Further, Kill-Bots maintains a loose form of fairness among authenticated clients, allowing each of them a maximum of 8 simultaneous connections. To grab most of the server's resources, an attacker has to maintain many more authenticated malicious clients than the number of legitimate users. This means that the attacker needs to own a server at least as popular as the victim Web server. Such a popular site is an asset. It is unlikely that the attacker will jeopardize his popular site to DDoS an equally or less popular Web site.

Furthermore, note that each request served by Kill-Bots is accompanied by a unique HTTP Cookie that was obtained by solving a puzzle. Kill-Bots can measure the resources consumed by each cookie and de-prioritize cookies that have already issued many more requests than *normal*. Here *normal* is the median number of requests over all cookies plus a few standard deviations. By associating a cookie with each puzzle solved, Kill-Bots can more accurately track resources consumed by one user and penalize the attacking humans. Finally, we note that security is a moving target; by forcing the attacker to resort to socially engineered attacks, we made the attack harder and the probability of being convicted higher.

(b) Polluting the Bloom Filter: The attacker may try to spoof his source address, masquerade as if he were a legitimate user, and pollute the Bloom filter for that legitimate user's IP address. This may cause Kill-Bots to confuse the legitimate user for a zombie and drop his requests. This attack, however, is not possible because Kill-Bots uses SYN cookies to prevent IP spoofing and the Bloom filter entries are modified *after* the SYN cookie check succeeds (Figure 2-10).

(c) False-Positives in the Bloom Filter: False-Positives are when the Bloom Filter mistakes legitimate IP addresses for zombies and can happen in one of two ways.

Lossy Data Structure: First, there is the organic reason – since the Bloom filter is a lossy approach to maintain membership in a set (here the set of zombies), there is a small probability that non-members are mistaken to be members (i.e., legitimate users confused as zombies). Bloom filters are characterized by the number of counters N and the number of hash functions k that map keys onto counters. Since a potentially large set of keys (32-bit IPs), are mapped onto much smaller storage (N counters), there is a non-zero probability that all k counters corresponding to a legitimate user pile up to $\chi = 32$ due to collisions with zombies. Assuming a distinct zombies and uniformly random hash functions, the probability a legitimate client is classified as a zombie is approximately $(1 - e^{-ka/N})^k \approx (\frac{ka}{N})^k$ [60]. Using $N = 2^{23}$, $k = 5$ and supposing there are 75000 zombies, the false positive probability is an insignificant $1.8 * 10^{-7}$.

Web Proxies and NATs: A second type of false positives is due to an aliasing problem. Web proxies [83] and Network Address Translators (NATs) [77] multiplex a single IP address among multiple users. If all clients behind the proxy are legitimate users, then sharing the

IP address has no impact. In contrast, if a zombie shares the proxy IP with legitimate clients and uses the proxy to mount an attack on the Web server, the bloom filter may block all subsequent requests from the proxy IP address. To ameliorate such fate-sharing, Kill-Bots increments the Bloom counter by 1 when giving out a puzzle but decrements the Bloom counters by $x \geq 1$ whenever a puzzle is answered. Kill-Bots picks x based on server policy. For $x > 1$, the proxy IP will be blocked only if the zombie's traffic forwarded by the proxy/NAT is at least $x - 1$ times the legitimate traffic from the proxy. Further, the value of x can be adapted based on server load. For example, Kill-Bots starts with a high value of x, attempting to be lenient to proxies and NATs. If the server load remains high after the Bloom filter stops learning IP addresses that cross χ, Kill-Bots decreases the value of x because it can no longer afford to serve a proxy that has such a large number of zombies behind it. The onus then is on the proxy or NAT to police its internal machines.

(d) IP Address Harvesting: What if an attacker can artificially inflate the number of IP addresses he can send and receive traffic at? Since the bloom filter gives out $\chi = 32$ puzzles before blocking an IP that does not answer any of its puzzles, will gaining access to many more IPs prolong an attack? We note that harvesting whether it involves picking up unused IP addresses on a local subnet by polluting the ARP entries of the upstream router [178] or picking up unused address blocks by advertising spurious BGP routes [140] involves much more effort and expertise than simply issuing HTTP requests. In a sense, by forcing the attacker to resort to these harder-to-engineer attacks, Kill-Bots improves security. Furthermore, a simple extension lets Kill-Bots ameliorate such harvesting. Note that all known forms of harvesting, including the above, involve IP addresses with locality, either those within a university subnet or within a BGP route advertisement. Normally, there is a low chance that a significant fraction of machines behind a subnet are all zombies. Hence, it is fair to have a policy that blocks an entire subnet if say more than 10% of its IPs behave as zombies. Kill-Bots can be extended to do this via an additional bloom filter to identify subnets that have more than a certain fraction of their IPs behaving as zombies.

(e) Database Attack: An adversary might try to collect all possible puzzles and the corresponding answers. When a zombie receives a puzzle, it searches its database for the corresponding answer, and sends it back to the server. To prevent this attack, Kill-Bots uses a large number of puzzles and periodically replaces puzzles with a new set. Further, there are many possible graphical puzzles. Building a database of all puzzles and their answers, potentially distributing the database to all zombies, and searching the database within the lifetime of a puzzle, is challenging.

(f) Concerns regarding in-kernel HTTP header processing: Kill-Bots does not parse HTTP headers; it pattern matches the arguments to the GET and the Cookie fields against the fixed string *validate* and against a 192-bit Kill-Bots cookie respectively. The pattern-matching is done in-place, i.e., without copying the packet, and is cheap, i.e., takes $< 8\mu s$ per request (see Section §2.7.2).

(g) Breaking the CAPTCHA: Prior work on automatically solving simple CAPTCHAs exists [128], but such programs are not available to the public for security reasons [128]. However, when one type of CAPTCHAs is broken, Kill-Bots can switch to a different kind. For example, we have seen linguistic CAPTCHAs in use – wherein the server asks text based questions, such as "what was the day before yesterday?" that humans can easily answer but are hard to automate as they require parsing sentences and understanding semantics.

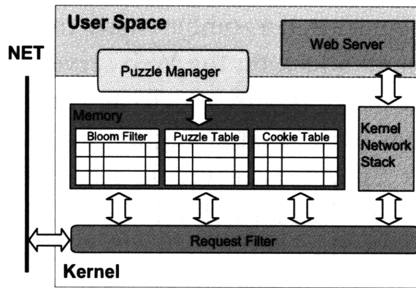


Figure 2-9 – A Modular representation of the Kill-Bots code.

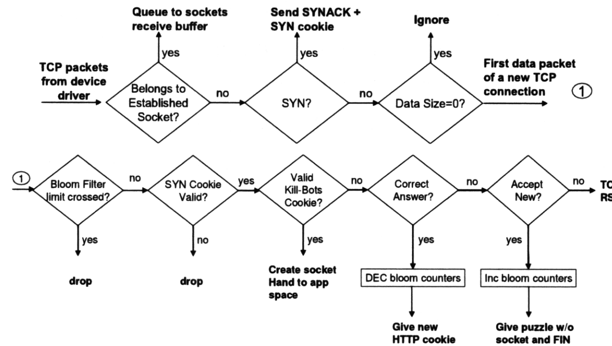


Figure 2-10 – The path traversed by new sessions in Kill-Bots. This code-path is implemented by the Request Filter module.

2.6 Kill-Bots System Architecture

Figure 2-9 illustrates the key components of Kill-Bots, which we briefly describe below.
(a) The Puzzle Manager consists of two components. The first component is a user-space stub that asynchronously obtains new puzzles. Generating graphical puzzles is relatively easy [22], and can either be done on the server itself in periods of inactivity (at night) or on a different dedicated machine. Also, puzzles may be purchased from a trusted third party. The second component is a kernel-thread that periodically loads new puzzles from disk into the in-memory Puzzle Table.
(b) The Request Filter (RF) processes every incoming TCP packet addressed to port 80. It is implemented in the bottom half of the interrupt handler to ensure that unwanted packets are dropped as early as possible. It is responsible for admitting requests, sending the graphical test, checking answers, providing cookies to authenticated clients, and ensuring each correct answer allows access only to a limited set of resources.

Figure 2-10 provides a flowchart representation of the RF code. When a TCP packet arrives for port 80, the RF first checks whether it belongs to an established connection in which case the packet is immediately queued in the socket's receive buffer and left to standard kernel processing. Otherwise, the RF checks whether the packet starts a new connection (i.e., is it a SYN?), in which case, the RF replies with a SYNACK that contains a standard SYN cookie. If the packet is not a SYN, the RF examines whether it contains any data; if not, the packet is dropped without further processing. Next, the RF performs two inexpensive tests in an attempt to drop unwanted packets quickly. It hashes the packet's source IP address and checks whether the corresponding entries in the Bloom filter have all exceeded χ

unsolved puzzles, in which case the packet is dropped. Otherwise, the RF checks that the acknowledgment number is a valid SYN cookie.

If the packet passes all of the above checks, it is the first data packet from a new TCP connection and the RF looks for three distinct possibilities. First, the packet might be from an unauthenticated client, and thus it goes through admission control (the "*Accept New?*" decision box) and is dropped with probability $1 - \alpha$. If accepted, the RF sends a puzzle and terminates the connection immediately. Second, the packet might be from a client that has already received a puzzle and is coming back with an answer. In this case, the RF verifies the answer and assigns the client an HTTP cookie, which allows access to the server for a period of time (the "Correct Answer?" decision box). Finally, the packet may be from an authenticated client that has a Kill-Bots HTTP cookie and is coming back to retrieve more objects (the "*Valid Cookie?*" decision box), in which case the packet contains the cookie. If the cookie is valid, Kill-Bots passes the new connection to the web-server, otherwise treats it as if the packet was from an unauthenticated client. If none of the above is true, the RF drops this packet. These checks are ordered according to their increasing cost to shed attackers as cheaply as possible.

(c) The Puzzle Table maintains the puzzles available to be served to users. To avoid races between writes and reads to the table, we divide the Puzzle Table into two memory regions, a write window and a read window. The Request Filter fetches puzzles from the read window, while the Puzzle Manager loads new puzzles into the write window periodically in the background. Once the Puzzle Manager loads a fresh window of puzzles, the read and write windows are swapped atomically.

(d) The Cookie Table maintains the number of concurrent connections for each HTTP cookie (limited to 8).

(e) The Bloom Filter counts unanswered puzzles for each IP address, allowing the Request Filter to block requests from IPs with more than ξ unsolved puzzles. Our prototype implementation sets $\xi = 32$, the number of counters N and the number of hash functions k of the bloom filter to $N = 2^{20}$ and $k = 2$. Each counter uses eight bits (count from 0 to 256), causing the bloom filter to fit within $1MB$. We believe that much larger bloom filters should be feasible in practice.

2.7 Evaluation

We evaluate a Linux-based kernel implementation of Kill-Bots in the wide-area network using PlanetLab.

2.7.1 Experimental Environment

(a) Web Server: The web server is a 2GHz P4 with 1GB RAM and 512KB L2 cache running an unmodified mathopd [24] web-server on top of a modified Linux 2.4.10 kernel. We chose mathopd because of its simplicity. The Kill-Bots implementation consists of (a) 300 lines of modifications to kernel code, mostly in the TCP/IP protocol stack and (b) 500 additional lines for implementing the puzzle manager, the bloom filter and the adaptive controller. To

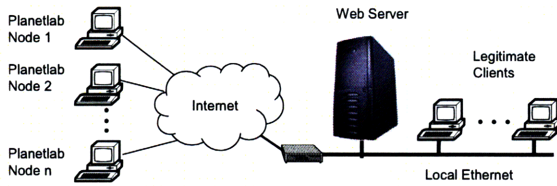


Figure 2-11 – Our Experimental Setup.

Function	CPU Latency
Bloom Filter Access	.7 μs
Processing HTTP Header	8 μs
SYN Cookie Check	11 μs
Serving puzzle	31 μs

Table 2.2 – Kill-Bots Microbenchmarks

obtain realistic server workloads, we replicate both static and dynamic content served by two websites, the CSAIL web-server and a Debian mirror.

(b) Modeling Request Arrivals: Legitimate clients generate requests by replaying HTTP traces collected at the CSAIL web-server and a Debian mirror. Multiple segments of the trace are played simultaneously to control the load generated by legitimate clients. A zombie issues requests at a desired rate by randomly picking a URI (static/dynamic) from the content available on the server.

(c) Experiment Setup: We evaluate Kill-Bots in the wide-area network using the setup in Figure 2-11. The Web server is connected to a 100Mbps Ethernet. We launch Cyberslam attacks from 100 different nodes on PlanetLab using different port ranges to simulate multiple attackers per node. Each PlanetLab node simulates up to 256 zombies—a total of 25,600 attack clients. We emulate legitimate clients on machines connected over the Ethernet, to ensure that any difference in their performance is due to the service they receive from the Web server, rather than wide-area path variability.

(d) Emulating Clients: We use a modified version of WebStone2.5 [39] to emulate both legitimate Web clients and attackers. WebStone is a benchmarking tool that issues HTTP requests to a web-server given a specific distribution over the requests. We extended WebStone in two ways. First, we added support for HTTP sessions, cookies, and for replaying requests from traces. Second, to have the clients issue requests at the specified rate independent of how the web-server responds to the load, we rewrote WebStone's networking code using libasync [121], an asynchronous socket library.

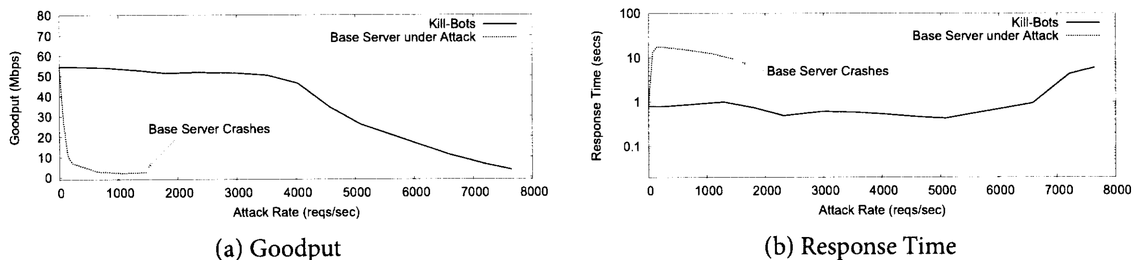
Metrics

We evaluated Kill-Bots by comparing the performance of a base server (i.e., a server with no authentication) with its Kill-Bots mirror operating under the same conditions. Server performance is measured using these metrics:

(a) Goodput of Legitimate Clients: The number of bytes per second delivered to *all* legitimate client applications. Goodput ignores TCP retransmissions and is averaged over 30s windows.

(b) Response Times of Legitimate Clients: The elapsed time before a request is completed or timed out. We timeout incomplete requests after 60s.

(c) Cumulative Number of Legitimate Requests Dropped: The total number of legitimate requests dropped since the beginning of the experiment.



(a) Goodput (b) Response Time

Figure 2-12 – Kill-Bots under Cyberslam: Goodput and average response time of legitimate users at different attack rates for both a base server and its Kill-Bots version. Kill-Bots substantially improves server performance at high attack rates.

2.7.2 Microbenchmarks

We ran microbenchmarks on the Kill-Bots kernel to measure the time taken by the various modules. We used the x86 rdtsc instruction to obtain fine-grained timing information. rdtsc reads a hardware timestamp counter that is incremented once every CPU cycle. On our 2GHz web-server, this yields a resolution of 0.5 nanoseconds. The measurements are for CAPTCHAs of 1100 bytes.

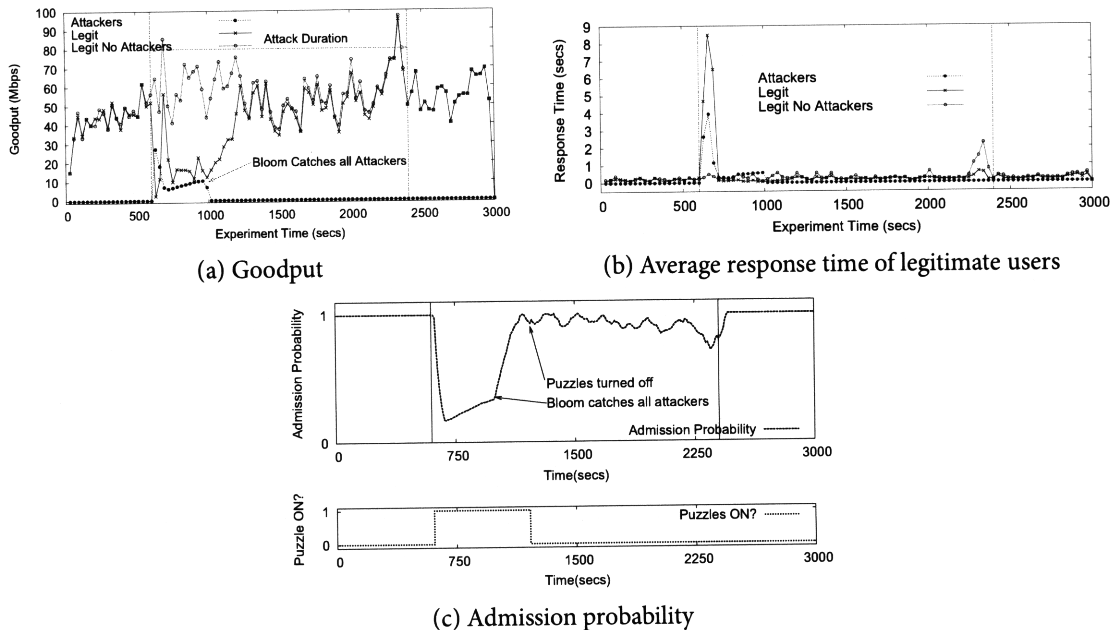
Table 2.2 shows our microbenchmarks. The overhead for issuing a graphical puzzle is $\approx$ $40\mu s$ (process http header +serve puzzle), which means that the CPU can issue puzzles faster than the time to transmit a 1100B puzzle on 100Mb/s Ethernet. However, the authentication cost is dominated by standard kernel code for processing incoming TCP packets, mainly the interrupts ($\approx 10\mu s$ per packet [113], about 10 packets per TCP connection). Thus, the CPU is the bottleneck for authentication and as shown in §2.7.5, performing admission control based on CPU utilization is beneficial.

Note also that checking the Bloom filter is much cheaper than other operations including the SYN cookie check. Hence, for incoming requests, we perform the Bloom filter check before the SYN cookie check (Figure 2-14). In the JUST_FILTER stage, the Bloom filter drops all zombie packets; hence performance is limited by the cost for interrupt processing and device driver access. We conjecture that using polling drivers [113, 125] will improve performance at high attack rates.

2.7.3 Kill-Bots under Cyberslam

We evaluate the performance of Kill-Bots under Cyberslam attacks, using the setting described in §2.7.1. We also assume only 60% of the legitimate clients solve the CAPTCHAs; the others are either unable or unwilling to solve them. This is supported by the user study in §2.7.6.

Figure 2-12 compares the performance of Kill-Bots with a base (i.e., unmodified) server, as the attack request rate increases. Figure 2-12(a) shows the goodput of both servers. Each point on the graph is the average goodput of the server in the first twelve minutes after the beginning of the attack. A server protected by Kill-Bots endures attack rates multiple orders of magnitude higher than the base server. At very high attack rates, the goodput of the Kill-Bots server decreases as the cost of processing interrupts becomes excessive. Figure 2-12(b) shows the response time of both web servers. The average response time experienced by



(a) Goodput

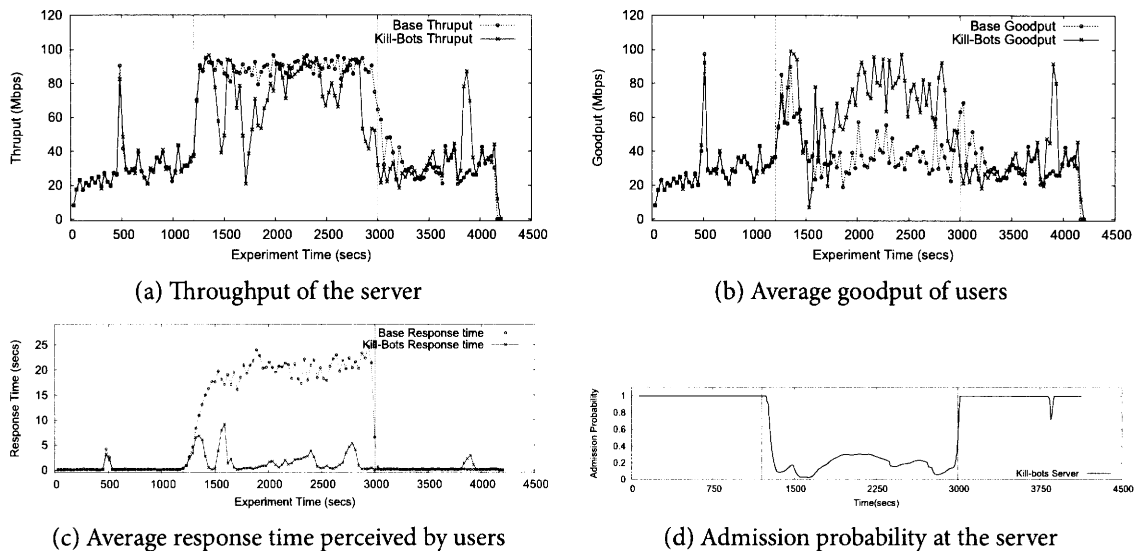
(b) Average response time of legitimate users

(c) Admission probability

Figure 2-13 – Comparison of Kill-Bots' performance to server with no attackers when only 60% of the legitimate users solve puzzles. Attack lasts from 600s to 2400s. (a) Goodput quickly improves once bloom catches all attackers. (b) Response times improve as soon as the admission control reacts to the beginning of attack. (c) Admission control is operational in both the SERVE_PUZZLES and the JUST_FILTER stages.

legitimate users increases dramatically when the base server is under attack. In contrast, the average response time of users accessing a Kill-Bots server is only moderately affected by the ongoing attack for even much higher attack rates.

Figure 2-13 shows the dynamics of Kill-Bots during a Cyberslam attack, with $\lambda_a = 4000$ req/s. The figure also shows the goodput and mean response time with no attackers, as a reference. The attack begins at $t = 600$s and ends at $t = 2400$s. At the beginning of the attack, the goodput decreases (Figure 2-13(a)) and the mean response time increases (Figure 2-13(b)). Yet, quickly the admission probability decreases (Figure 2-13(c)), causing the mean response time to go back to its value when there is no attack. The goodput however stays low because of the relatively high attack rate, and because many legitimate users do not answer puzzles. After a few minutes, the Bloom filter catches all zombie IPs, causing puzzles to no longer be issued (Figure 2-13(c)). Kill-Bots now moves to the JUST_FILTER stage, stops issuing graphical tests and drops all requests from IP addresses that are recognized by the Bloom filter as zombies. This causes a large increase in goodput (Figure 2-13(a)) due to the admission of users who were earlier unwilling or unable to solve CAPTCHAs and also due to the reduction in authentication cost. In this experiment, despite the ongoing Cyberslam attack, Kill-Bots' performance in the JUST_FILTER stage ($t = 1200s$ onwards), is close to that of a server not under attack. Note that the normal load significantly varies with time and the adaptive controller (Figure 2-13(c)) reacts to these changes $t \in [1200, 2400]s$, keeping response times low while providing reasonable goodput.



(a) Throughput of the server

(b) Average goodput of users

(c) Average response time perceived by users

(d) Admission probability at the server

Figure 2-14 – Kill-Bots under Flash Crowds: The Flash Crowd event lasts from t=1200s to t=3000s. Though Kill-Bots has a slightly lower throughput, its Goodput is much higher and its average response time is lower.

2.7.4 Kill-Bots under Flash Crowds

We evaluate the behavior of Kill-Bots under a Flash Crowd. We emulate a Flash Crowd by playing our Web logs at a high speed to generate an average request rate of 2000 req/s. The request rate when there is no flash crowd is 300 req/s. This matches Flash Crowd request rates reported in prior work [95, 106]. In our experiment, a Flash Crowd starts at $t = 1200s$ and ends at $t = 3000s$.

Figure 2-14 compares the performance of the base server against its Kill-Bots mirror during the Flash Crowd event. The figure shows the dynamics as functions of time. Each point in each graph is an average measurement over a 30s interval. We first show the total throughput of both servers in Figure 2-14(a). Kill-Bots has slightly lower throughput for two reasons. First, Kill-Bots attempts to operate at β=12% idle cycles rather than at zero idle cycles. Second, Kill-Bots uses some of the bandwidth to serve puzzles. Figure 2-14(b) reveals that the throughput figures are misleading; though Kill-Bots has a slightly lower throughput than the base server, its goodput is substantially higher (almost 100% more). This indicates that the base server wasted its throughput on retransmissions and incomplete transfers. Figure 2-14(c) provides further supporting evidence–Kill-Bots drastically reduces the average response time.

That Kill-Bots improves server performance during Flash Crowds might look surprising. Although all clients in a Flash Crowd can answer the graphical puzzles, Kill-Bots computes an admission probability α such that the system only admits users it can serve. In contrast, a base server with no admission control accepts more requests than it can serve during a flash crowd. This leads to queues getting filled up, inflated response times and eventually half-served requests are dropped either because hard resource constraints such as the number of pending requests are crossed or because user requests time out. All of this leads to wasted resources in the base server. Figure 2-14(d) supports this argument by showing how the admission probability α changes during the Flash Crowd event to allow the server to shed

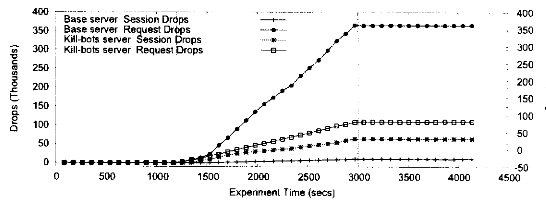


Figure 2-15 – Cumulative numbers of dropped requests and dropped sessions under a Flash Crowd event lasting from $t = 1200s$ to $t = 3000s$. Kill-Bots adaptively drops sessions upon arrival, ensuring that accepted sessions obtain full service, i.e. have fewer requests dropped.

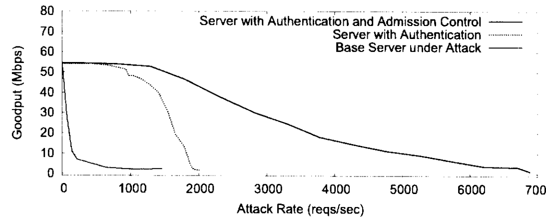


Figure 2-16 – Server goodput substantially improves with adaptive admission control. This figure is similar to Figure 2-7 but is based on wide-area experiments rather than analysis. (For clarity, the Bloom filter is turned off in this experiment.)

away the extra load.

Finally, Figure 2-15 shows the cumulative number of dropped requests and dropped sessions during the Flash Crowd event for both the base server and the Kill-Bots server. Interestingly, the figure shows that Kill-Bots drops more sessions but fewer requests than the base server. The base server accepts new sessions more often than Kill-Bots but keeps dropping their requests. Kill-Bots drops sessions upon arrival, but once a session is admitted it is given a Kill-Bots cookie which allows it access to the server for 30min.

Note that Flash Crowds is just one example of a scenario in which Kill-Bots only needs to perform admission control. Kill-Bots can identify such scenarios which are characterized by high server load but few bad bloom entries. In such a scenario, Kill-Bots decouples authentication from admission control by no longer issuing puzzles; instead every user that passes the admission control check gets a Kill-Bots cookie.

2.7.5 Benefit from Admission Control

In §2.4, using a simple model, we showed that admission control lets an authentication mechanism perform better at high attack rates. Figure 2-16 provides experimental evidence that confirms the analysis. The figure compares the goodput of a version of Kill-Bots that uses only puzzle-based authentication, with a version that uses both puzzle-based authentication and admission control. We turn off the Bloom filter in these experiments because we are interested in measuring the goodput gain obtained only from admission control. The results in this figure are fairly similar to those in Figure 2-7 showing that admission control increases server resilience at high attack rates.

2.7.6 User Willingness to Solve Puzzles

We conducted a user study to evaluate the willingness of users to solve CAPTCHAs. We instrumented our research group's Web server to present puzzles to 50% of all external ac-

Case	%Users
Answered puzzle	55%
Interested surfers who answered puzzle	74%

Table 2.3 – The percentage of users who answered a graphical puzzle to access the Web server. We define interested surfers as those who access two or more pages on the Web site.

cesses to the *index.html* page. Clients that answer the puzzle correctly are given an HTTP cookie that allows them access to the server for an hour. The experiment lasted from Oct. 3'rd to Oct. 7'th, 2004. During that period, we registered a total of 973 accesses to the page, from 477 distinct IP addresses.

We compute two types of results. First, we filter out requests from known robots, using the User-Agent field, and compute the fraction of clients who answered our puzzles. We find that 55% of all clients answered the puzzles. It is likely that some of the remaining requests are also from robots that don't use well-known User-Agent identifiers, so this number underestimates the fraction of humans that answered the puzzles. Second, we distinguish between clients who check only the group's main page and leave the server, and those who follow one or more links. We call the latter *interested surfers*. We would like to check how many of the interested surfers answered the graphical puzzle because these users probably bring more value to the Web site. We find that 74% of interested users answer puzzles. Table 2.3 summarizes our results. These results may not be representative of users in the Internet, as the behavior of user populations may differ from one server to another.

2.8 Related Work

Related work falls into the following areas.

(a) Denial of Service: Much prior work on DDoS describes specific attacks (e.g., SYN flood [144], Smurf [67], reflector attacks [134] etc.), and presents detection techniques or countermeasures. In contrast to Kill-Bots, prior work focuses on lower layers attacks and bandwidth floods. The back-scatter technique [126] detects DDoS sources by monitoring traffic to unused segments of the IP address space. Trace-back [151] uses in-network support to trace offending packets to their source. Many variations to the trace-back idea detect low-volume attacks [159, 161, 181]. Others detect bandwidth floods by the mis-match in the volumes of traffic [90] and some push-back filtering to throttle traffic closer to its source [119]. Anderson et. al. [50] propose that routers only forward packets with capabilities. Juels and Brainard [97] first proposed computational client puzzles as a SYN flood defense. Some recent work uses overlays as distributed firewalls [47, 73, 109]. Clients can only access the server through the overlay nodes, which filter packets.

(b) Cyberslam Defense: WebSOS [127] proposes to use graphical tests in order to allow access to a secure overlay that then indirectly routes packets to the server. Kill-Bots differs by using CAPTCHAs only as an intermediate stage to identify the offending IPs thereby allowing users who do not or cannot solve CAPTCHAs access to the server. Further, Kill-Bots combines authentication with admission control and focuses on an efficient kernel implementation. Post-dating our work, is a body of research for novel approaches to address this attack [61, 73, 178, 183]. TVA [183] augments today's Internet with a capabilities infras-

tructure, wherein traffic senders obtain capabilities from intended receivers and stamp their packets with these capabilities. The network delivers only those packets that have appropriate capabilities and drops all others. Portcullis [61] solves denial-of-service attacks that could be mounted on the capability-providing mechanism. Speak-up [178] is a novel idea that makes legitimate users send more requests when the server is under attack, so that if the server were to simply drop requests agnostic-ally, i.e., without distinguishing legitimate users from attackers, legitimate users will end up with more server resources than the attacker. Speak-up leverages the fact that all the legitimate users together have more network bandwidth than the botnets. However, speak-up does burden network links a lot more (with all the duplicate requests) than most other Cyberslam defenses. Phalanx [73] is similar to WebSOS, in terms of redirecting requests through an indirection overlay that filters attackers; yet handles many un-resolved issues in WebSOS.

(c) CAPTCHAs: Our authentication mechanism uses graphical tests or CAPTCHAs [176]. Several other reverse Turing tests exist [71, 111, 148]. CAPTCHAs are currently used by many online businesses (e.g., Yahoo!, Hotmail).

(d) Flash Crowds and Server Overload: Prior work [94, 180] shows that admission control improves server performance under overload. Some admission control schemes [56, 175] manage OS resources better. Others persistently drop TCP SYN packets in routers to tackle Flash Crowds [95]. Still others shed extra load onto an overlay or a peer-to-peer network [163, 166]. Kill-Bots couples admission control with authentication to address both flash-crowds and DDoS attacks.

(e) Cryptographic Puzzles: Some prior work allows the client access to the server only after the client performs some computational work, such as reversing a hash [76, 97]. PuzzleNet [82] implements such a scheme. If the attacker has access to a sizable botnet, which is often the case [118, 137], then he has enough computational power to invert so many hashes and flood the server with requests. In this case, having clients solve a hash only defers the attack for a later time.

2.9 Generalizing & Extending Kill-Bots

We place Kill-Bots in context of general approaches and examine some new use cases.

Rate-Limiting at the Granularity of a User: Kill-Bots is different from traditional rate-limiting schemes that attempt to identify and filter the heavy resource consumers. Traditional schemes run into the problems of accounting and aliasing. First, it is hard to account for all the CPU, memory, database, remote procedure calls and kernel-state that a user consumes over time at the server. Not only does this require detailed knowledge of the internals of the web-server and its operating system, but also which of these resources may be the bottleneck at any time depends on the server workload and might change over time. Second, even if one could identify the bottleneck resource and accurately measure resource consumption, it is not clear which user to associate this measurement with. Most sites do not have login/password pairs as users frown on registering with websites. An IP address can be shared among possibly unknown number of users due to Web Proxies and Network Address Translators. Hence, some IPs legitimately use a lot more resources than others and

an IP address is a poor measure of resource consumption. Third, the state-of-the-art method of giving users a unique HTTP cookie that identifies their requests can be bypassed if multiple users simply share their cookies (e.g., with Google Mail cookies [20]). Rather than accounting for each of the myriad resources, Kill-Bots forces users to solve a Reverse Turing Test. Hence, more often that not, the bottleneck exposed to the attacker is the server's CPU — an attacker can only force the server to issue puzzles, which exercises server CPU and bandwidth on the server's outward facing link. For legitimate users who solve the puzzle, Kill-Bots issues an unforgeable cookie as a proof-of-work and hence, circumvents the aliasing problem. When necessary, Kill-Bots can serve requests with cookies that have already used up a lot of resources at a lower priority.

Serve-and-Cap: We recently came across an intriguing variation of using Kill-Bots. Under overload, rather than issue CAPTCHAs to every un-authenticated client, a server first estimates the amount of requests being issued by each IP address or HTTP Cookie. The server allows low volume users to continue accessing the server, but users making requests beyond a certain rate are asked to serve a graphical puzzle, and if they do not solve the test, are denied service. Google's search page uses this logic. For example, clients that make more than a few tens of searches within a second, are asked to solve a graphical test before being allowed further service.

Kill-Bots on a Cluster: Can Kill-Bots help defend a website that runs on a cluster, i.e., when client requests are served by one of several identical servers? The straightforward approach would be to deploy Kill-Bots on all the servers. While this is feasible, simply replicating the bloom filter and the cookie table, will unintentionally relax some guarantees. For example, solving a puzzle will allow a client up to 8 simultaneous connections on each of one (or a few)the servers. Fortunately however, most such website deployments already use load balancing front-ends that distribute incoming requests across the servers. Hence, we suggest that some components of Kill-Bots – specifically, the Bloom filter which counts the number of unanswered puzzles and the Cookie table which limits the number of connections per solved puzzle, be implemented on the front-end. The more resource intensive components such as admission control, serving puzzles and checking answers can still be performed on the distributed cluster. Note that this separation of functionality requires minimal coordination among the servers, or between the servers and the front-end load-balancer.

Kill-Bots as a Service: Taking the separation of functionality a step further, a third-party provider such as Akamai [10] might provide Kill-Bots as a service. In this use-case, the provider might set up a distributed overlay that invokes Kill-Bots, i.e., client's traffic is routed through the overlay which first authenticates clients and only forwards authenticated requests through to the website. The website in turn accepts only requests forwarded by the overlay. Phalanx [73] and WebSOS [127] solve the remainder of problems associated with managing such an overlay securely; Kill-Bots complements these by providing a better method to authenticate new users to the overlay.

2.10 Limitations & Open Issues

A few limitations and open issues are worth discussing.

First, Kill-Bots has a few parameters that we have assigned values based on experience.

For example, we set the Bloom filter threshold $\xi = 32$ because even legitimate users may drop puzzles due to congestion or indecisiveness and should not be punished. There is nothing special about 32, we only need a value that is neither too big nor too small. Similarly, we allow a client that answers a CAPTCHA a maximum of 8 parallel connections as a trade-off between the improved performance gained from parallel connections and the desire to limit the loss due to a compromised cookie.

Second, Kill-Bots assumes that the first data packet of the TCP connection will contain the GET and Cookie lines of the HTTP request. In general the request may span multiple packets, but we found this to happen rarely.

Third, the Bloom filter needs to be flushed at some slow rate since compromised zombies may turn into legitimate clients. The Bloom filter can be cleaned either by resetting all entries simultaneously or by decrementing the various entries at a particular rate. In the future, we will examine which of these two strategies is more suitable.

2.11 Conclusion

The Internet literature contains a large body of research on denial of service solutions. The vast majority assume that the destination can distinguish between malicious and legitimate traffic by performing simple checks on the content of packets, their headers, or their arrival rates. Yet, attackers are increasingly disguising their traffic by mimicking legitimate user's access patterns, which allows them to defy traditional filters. This chapter focuses on protecting Web servers from DDoS attacks that masquerade as Flash Crowds. Underlying our solution is the assumption that most online services value human surfers much more than automated accesses. We present a novel design that uses CAPTCHAs to distinguish the IP addresses of the attack machines from those of legitimate clients. In contrast to prior work on CAPTCHAs, our system allows legitimate users to access the attacked server even if they are unable or unwilling to solve graphical tests. We implemented our design in the Linux kernel and evaluated it in Planetlab.

Requiring proof of human work prior to service is increasingly common at blogs, credit card applications and even search engines. We expect the idea—*constrain harmful traffic to the number of continuously active human attackers that can be mustered for the attack* will reoccur in different forms. The lessons learnt from Kill-Bots include (a) admission control complements authentication nicely when apportioning limited resources among competing input types and (b) engineering systems to handle harmful traffic as quickly as possible while maintaining as little state as possible keeps servers resilient to significant attack volumes.

Chapter 3

Responsive Yet Stable Traffic Engineering

We now shift focus to resources in the network backbones operated by Internet Service Providers (ISPs). ISPs are in the business of carrying user's traffic on their backbone network. The ISP knows its network topology, i.e., the routers, links and their capacities, and wants to map user's traffic onto the underlying topology. Finding a good mapping that avoids hot-spots and provides good performance to users is a key part of an ISP's business. Besides finding a good mapping, the challenge lies in adapting the mapping when unpredictable traffic spikes or link failures happen before end users notice poor performance. In this chapter, we present TeXCP an online traffic engineering scheme that helps ISPs manage their resources better in the face of unpredictable changes.

3.1 Overview

Intra-domain Traffic Engineering (TE) is an essential part of modern ISP operations. The TE problem is typically formalized as minimizing the maximum utilization in the network [51, 52, 85, 124]. This allows the ISP to balance the load and avoid hot spots and failures, which increases reliability and improves performance. Furthermore, ISPs upgrade their infrastructure when the maximum link utilization exceeds a particular threshold (about 40% utilization [92]). By maintaining lower network utilization for the same traffic demands, traffic engineering allows the ISP to make do with existing infrastructure for a longer time, which reduces cost.

Recent years have witnessed significant advances in traffic engineering methods, from both the research and operational communities [51, 78, 85, 173]. TE methods like the OSPF weight optimizer (OSPF-TE) [85, 86] and the MPLS multi-commodity flow optimizer [124] have shown significant reduction in maximum utilization over pure shortest path routing. Nonetheless, because of its *offline* nature, current TE has the following intrinsic limitations:

- It might create a suboptimal or even inadequate load distribution for the realtime traffic. This is because offline TE attempts to balance load given the long term traffic demands averaged over multiple days (potentially months). But the actual traffic may differ from the long term demands due to BGP re-routes, external or internal failures, diurnal variations, flash crowds, or attacks.

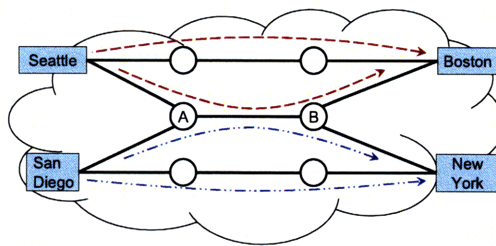


Figure 3-1 – For each Ingress-Egress (IE) pair, a TeXCP agent at the ingress router balances the IE traffic across available paths in an online fashion.

- Its reaction to failures is suboptimal. Offline TE deals with network failures by pre-computing alternate routings for a limited set of failures [86]. Since the operator cannot predict which failure will occur, offline TE must find a routing that works reasonably well under a large number of potential failures. Such a routing is unlikely to be optimal for any particular failure. As a result, current TE may fail to prevent congestion when unanticipated or combination failures occur, even though the network may have enough capacity to handle the failure.

The natural next step is to use online traffic engineering, which reacts in realtime to traffic changes and failures. It is challenging to build an online TE scheme that responds quickly to changes in traffic, yet does not lead to oscillations, as demonstrated by the instability of the early ARPAnet routing [110]. Prior online TE methods are either centralized [59, 62] or assume an oracle that provides global knowledge of the network [78], and most lack a stability analysis [155, 174]. There is a need for an online TE protocol that combines practical implementation, clear performance advantage, and stable behavior. Furthermore, we need to understand the performance gap between online and offline TE.

This chapter presents TeXCP, [1] a distributed responsive and stable online traffic engineering protocol. Our approach simplifies the design and analysis of online TE by splitting the problem into two components. First, a load-balancer takes as input the state of the network and shifts traffic from one path to another to minimize the utilization. Second, each path in the network has a closed-loop feedback controller that collects network feedback and ensures traffic stability on the path. Making the feedback controller work at a *faster time scale* than the load balancer achieves multiple goals: 1) The feedback controller is easy to stabilize by building on recent ideas in applying closed-loop feedback control to congestion control [107]. 2) The feedback controller stabilizes the network before the load balancer makes new decisions, giving the allusion of instantaneous movement of traffic. 3) As a result, the load balancer is easier to design and analyze because it can ignore the complexity of the underlying network.

TeXCP works as follows. In an ISP network, like the one in Figure 3-1, each ingress router may have traffic demands for a particular egress router or set of routers, e.g., traffic demands from Seattle to Boston, Boston to New York, etc. To each ingress-egress (IE) pair, we assign a TeXCP agent that resides at the ingress router and uses multiple paths or tunnels to deliver traffic from the ingress to the egress. The TeXCP agent uses light-weight explicit feedback from the core routers to discover path utilization. It adaptively moves traffic from

[1] pronounced *tex* as in latex, cee-pee

Phrase	Definition
IE flow	The traffic flow from an ingress to an egress router along a particular path
Active path	A path on which the TeXCP agent is sending traffic (i.e., $x_{sp} > 0$)
Path Utilization	Maximum link utilization along a path
Network (or Max-) Utilization	The maximum utilization over all links in the network
Routing	The set of routes that traffic for all IE pairs takes through the network

Table 3.1 – Definitions of terms used in the chapter.

over-utilized to under-utilized paths. Traffic is split among paths at the granularity of a flow, to avoid reordering TCP packets. TeXCP's load movements are carefully designed such that, though done independently by each edge router based on local information, the system balances load throughout the network. We model TeXCP under standard assumptions and show that it is stable. Further, our analysis provides a systematic approach to set parameters to constant values that work independent of the traffic demands and the failure scenarios.

Using simulations of multiple tier-1 topologies from Rocketfuel [34], many different traffic matrices, and changing network conditions, we evaluate TeXCP and study the performance gap between online and offline TE methods. Our results show that:

- For the same traffic demands, a network that uses TeXCP can support the same utilization and failure resilience as a network that uses traditional offline TE, but with a half or a third the capacity. This significantly reduces cost for the ISP.
- The network utilization under TeXCP is always within a few percent of the optimal value, independent of failures or deviations from the traffic matrix. In comparison, InvCap weight setting [4], a widely used TE method, results in an average network utilization 80% higher than optimal in the base case, and twice as high under failures. OSPF-TE, a state-of-the-art offline TE technique [85, 86], achieves a network utilization that is about 20% away from optimal, in the base case, but it is highly sensitive to failures and deviations in traffic load, sometimes creating a utilization that is twice or thrice the optimal.
- Compared to MATE [78], a prior online TE proposal, TeXCP converges faster, achieves a better load balance, and does not assume an oracle that provides global knowledge of the network.
- TeXCP automatically prunes additional paths whose usage does not reduce the maximum utilization in the network, and prefers shorter paths over longer paths.
- Finally, as explained in §3.6, TeXCP is easy to implement without any modifications to current router technology.

Our **terminology and variables** are described in Tables 3.1 and 3.2.

3.2 Problem Formalization

In an ISP network, like the one in Figure 3-1, each IE pair s has input traffic rate R_s and multiple paths P_s that can be used to deliver the traffic from ingress to egress. A fraction of the IE traffic x_{sp} is routed along path p. The problem is, how to split the traffic of each

Variable	Definition
R_s	Total Traffic Demand of IE pair s
P_s	Set of paths available to IE pair s
r_{sp}	Traffic of IE pair s sent on path p. i.e., $R_s = \sum r_{sp}$
x_{sp}	Fraction of IE, s, traffic sent on path p, called path weight.
u_{sp}	The utilization of path p observed by IE pair s
u_l	The utilization of link l
C_l	The capacity of link l
P_l	Set of paths that traverse link l
$\overline{u}_s$	Weighted average utilization of paths used by IE pair s

Table 3.2 – The variables most-used in the chapter. All of these variables are functions of time.

IE pair across the paths available to the pair such that the maximum link utilization in the network is minimized, i.e.,

$$\min_{x_{sp}} \max_{l \in L} u_l, \tag{3.1}$$

subject to the constraints:

$$u_l = \sum_s \sum_{p \in P_s,\, p \ni l} \frac{x_{sp} \cdot R_s}{C_l}, \tag{3.2}$$

$$\sum_{p \in P_s} x_{sp} = 1, \quad \forall s, \tag{3.3}$$

$$x_{sp} \geq 0, \quad \forall p \in P_s, \forall s. \tag{3.4}$$

Equation 3.1 states the optimization problem showing that we want to find $\{x_{sp}\}$, the traffic split ratios that minimize the maximum utilization across all links, $l \in L$. Equations 3.2– 3.4 are the constraints to the optimization problem; Equation 3.2 denotes that link utilization, u_l, is the total traffic on the link divided by its capacity, C_l; Equation 3.3 ensures that the traffic sent by an IE pair sums up to its demands; Equation 3.4 states that the traffic share on any path cannot be negative.

3.3 TeXCP Design

In a TeXCP network, the edge and core routers collaborate to balance the load and route around failures. The vast majority of the new functionality is at the edge routers, and can be built in software. The ingress router in each IE pair runs a TeXCP agent. The ISP configures each agent with a set of paths that it can use to deliver its IE traffic, and pins the paths using a standard protocol like RSVP-TE [53]. The TeXCP agent probes each path to discover its utilization and failure state, and splits the IE traffic across these paths to minimize the max-utilization. It adapts the split ratios in realtime to changing network conditions. The different TeXCP agents work independently, without exchanging information. Their combined effort optimizes the maximum utilization and balances the load.

Before going into the details, we note that TeXCP works independent of which IP prefixes are mapped to which ingress-egress pair. For example, BGP can freely reroute an IP prefix and change its egress point(s) in the network. TeXCP in turn, would treat this as a change in traffic demands and re-balance the routing. Furthermore, although our description focuses on point-to-point traffic, TeXCP seamlessly handles point-to-multipoint traffic; for example, by instantiating a virtual IE pair whose traffic flows from the ingress point to a virtual egress point. The virtual egress is connected to each of the physical routers at the multiple exit points of the traffic.

3.3.1 Path Selection

The ISP configures each TeXCP agent with a set of paths that it can use to deliver traffic between the corresponding IE pair. By default, TeXCP picks the K-shortest paths that connect the ingress to the egress router, where a path length is set to its propagation delay. Though preferable, these paths need not be link-disjoint.

A few points are worth noting. First, the choice of the per-IE paths is based solely on the topology and is independent of the state of each path (e.g., congestion, failure). Thus, paths are computed offline and are rarely re-computed. Second, a TeXCP agent uses the realtime congestion and failure state of a path to determine whether to use a path and how much traffic to send on the path. Thus, the actual number of paths used by a TeXCP agent may be much smaller than K and depends on whether increasing the number of active paths decreases the max-utilization. The default is $K = 10$ paths.

3.3.2 Probing Network State

To balance the traffic of an IE pair, a TeXCP agent corresponding to the IE pair, tracks the utilization of each path available to that pair (i.e., the maximum link utilization along each path). The TeXCP agent maintains a probe timer which fires every T_p seconds where T_p is larger than the maximum round trip time in the network and defaults to 100ms. Smaller values of T_p make TeXCP converge faster whereas larger values decrease the overhead (§3.6). When the timer fires, the TeXCP agent sends a small probe on each of its paths. A router that sees a probe packet checks whether the utilization reported in the packet is smaller than the utilization of its output link, in which case it overwrites the utilization in the probe with its own. The egress node at the end of the path unicasts the contents of the probe packet to the ingress node, which delivers it to the appropriate TeXCP agent. This packet (i.e., the probe's ack) goes directly to the ingress node and is not processed by intermediate routers.

Note that probe packets, like ICMP packets, do not follow the fast data path, and hence need no modifications to router hardware. Slow path processing may add an extra one or two milliseconds of delay to the probe at each router [91]. We show that this is negligible for our purpose (§3.5). Finally, we use probes only for ease of explanation; an equivalent approach with much lower overhead is described in §3.6.

Probe Loss is an indication of path failure, or congestion along the path. In TeXCP, probes have sequence numbers. If a previous probe is not acknowledged by the next time the probe timer fires, the agent exponentially increases its estimate of corresponding path utilization to $\max(1, \rho u_{sp})$, where u_{sp} is the path utilization from the previous T_p, and ρ is a

parameter > 1, that defaults to 1.2. As a result, failed and highly congested paths are quickly recognized, causing TeXCP to divert traffic from them to less congested paths.

3.3.3 The Load Balancer

Each TeXCP agent runs a load balancer that splits the traffic of an IE pair, s, among its available paths, P_s, with the objective of maintaining the max-utilization in the network as low as possible.[2]

The load balancer maintains a decision timer, which fires every T_d seconds. For stability reasons (see §3.4), the decision interval is set to five times the probe interval, i.e., $T_d \geq 5T_p$.

Our distributed load balancer works iteratively; every time the decision timer fires, the load balancer at TeXCP agent s computes a change in the fraction of IE traffic sent on path p denoted by Δx_{sp}. This change must satisfy the following constraints:

(1) At equilibrium, traffic assignment should not change, i.e., $\Delta x_{sp} = 0, \forall s, \forall p \in P_s$.
(2) Conservation of traffic implies $\sum_{p \in P_s} x_{sp} = 1, \forall s$.
(3) A path whose rate is zero cannot have its rate decreased, i.e., $x_{sp} = 0 \Rightarrow \Delta x_{sp} \geq 0$.
(4) Unless the system is at equilibrium, each update should decrease the maximum utilization, i.e., if path p has the largest utilization at TeXCP agent s, then either $x_{sp} = 0$ or $\Delta x_{sp} < 0$.

Taking the above into account, we adopt the following load balancing algorithm. When the decision timer fires, the TeXCP agent updates the fraction of traffic on its paths as follows:

$$\Delta x_{sp} = \begin{cases} \frac{r_{sp}}{\sum_{p'} r_{sp'}}(\overline{u}_s - u_{sp}) & \forall p, u_{sp} > u_s^{min} \\ \frac{r_{sp}}{\sum_{p'} r_{sp'}}(\overline{u}_s - u_{sp}) + \epsilon & p, u_{sp} = u_s^{min}. \end{cases} \tag{3.5}$$

The term r_{sp} is the draining rate of path $p \in P_s$, which is computed as the number of bits sent on the path since the last time the decision timer fired divided by the decision interval T_d. The term u_{sp} is the most recent value of the path's utilization reported by the probes, u_s^{min} is the minimum utilization across all paths controlled by this agent, $0 < \epsilon \ll 1$ is a small positive *constant*, and $\overline{u}_s$ is the average utilization normalized by the rates, i.e.,:

$$\overline{u}_s = \frac{\sum_{p \in P_s} r_{sp} \cdot u_{sp}}{\sum_{p' \in P_s} r_{sp'}}. \tag{3.6}$$

Before assigning traffic to paths, we normalize the traffic fractions, x_{sp}, to ensure they are

[2] For the interested reader, we note that the standard approach to solve optimizations such as the one in Equations 3.1– 3.4 is to make small iterative adjustments in the direction of gradient descent of the cost function. This approach does not apply here, since the cost function $\max u_l$ is not differentiable everywhere. Further, even when differentiable, the derivative is zero for all paths not traversing the max-utilization link. Picking an alternate cost function that mimics $\max u_l$, such as σu_l^j where $j \gg 1$, avoids this problem but slows convergence.

positive and sum up to 1:

$$\hat{x}_{sp} = max(0, x_{sp} + \Delta x_{sp}), \quad (3.7)$$

$$x_{sp} = \frac{\hat{x}_{sp}}{\sum \hat{x}_{sp'}}. \quad (3.8)$$

Equation 3.5 is fairly intuitive. In particular, setting $\Delta x_{sp} \propto (\overline{u}_s - u_{sp})$ means that any path whose utilization is above the average utilization should decrease its rate whereas any path whose utilization is below the average should increase its rate. But this is not enough. Paths with larger minimum capacity need more traffic to achieve the same utilization as smaller capacity paths. To cope with this issue, Equation 3.5 sets $\Delta x_{sp} \propto r_{sp}$. This makes the change in a path's traffic proportional to its current traffic share, which is the contribution of this TeXCP agent to the path utilization.

Additionally, Equation 3.5 uses the normalized average utilization rather than the standard average to deal with inactive paths, i.e., paths for which $r_{sp} = 0$. Assume one of the paths controlled by the TeXCP agent has a very high utilization compared to the others, but the TeXCP agent is not sending traffic on that path ($r_{sp} = 0$). The TeXCP agent cannot move traffic from that path because it is not sending any traffic on it. Yet, the high utilization of this inactive path might make the un-normalized average utilization much higher than all other utilizations. This may prevent the TeXCP agent from moving traffic away from other highly utilized paths, which halts the optimization. To prevent such a situation, Equation 3.6 normalizes the average utilization by the rates.

The last point to note about Equation 3.5 is the use of ϵ. Since TeXCP performs online load balancing, it must react when changing network conditions reduce load on a previously congested path. Without ϵ, $r_{sp} = 0 \Rightarrow \Delta x_{sp} = 0$, which means that if at some point the algorithm decided a particular path is highly utilized and should not be used, it will never use it again even if the utilization decreases later. To prevent this situation, Equation 3.5 always moves a small amount of traffic to the path with the minimum utilization (unless all paths have the same utilization). ϵ is a small positive *constant* chosen according to Theorem 3.4.2 (Equation 38), to ensure convergence.

Guarantees: We would like to highlight the nature of the equilibrium that TeXCP's load balancer leads to and defer proof of convergence for later (see §3.4). From Equation 3.5, observe that $\Delta x_{sp} = 0$ only when one of these two conditions hold:

1. $r_{sp} = 0$ **and** $u_{sp} > \overline{u}_s$: This denotes the case when the TeXCP agent stops using a path because the path's utilization is larger than all the paths currently being used by the agent.

2. $u_{sp} = \overline{u}_s$: This denotes the case wherein the path's utilization is equal to the normalized utilization.

To summarize, at equilibrium, a TeXCP agent distributes traffic on a set of paths all of which have utilization equal to $\overline{u}_s$ (the bottom condition above). Further, all the paths unused by a TeXCP agent have utilization strictly larger than that of paths used by TeXCP (the top condition above). This equilibrium is promising, because it confirms to the Nash criterion [88]. No TeXCP agent can lower its utilization by unilaterally moving away from the equilibrium.

In Section 3.4, we prove that TeXCP converges to the equilibrium and comment on its optimality.

Finally, for practical reasons, we would like to use a small number of paths for each IE pair, even if that makes the max-utilization slightly higher than the optimal achievable value. Thus, TeXCP stops using a path when its rate falls below a threshold (e.g., less than 10% of the total traffic of the IE pair). More formally:

$$u_{sp} > 0.9\, u_s^{max} \text{ and } x_{sp} < \sigma \Rightarrow x_{sp} = 0, \tag{3.9}$$

where u_s^{max} is the maximum utilization over all paths controlled by agent s, and σ is a small constant set by default to 0.1.

3.3.4 Preventing Oscillations and Managing Congestion

On the face of it, realtime load balancing appears simple; all one has to do is to iteratively move traffic from the over-utilized paths to the under-utilized paths until the utilization is balanced. Unfortunately, this is over-simplistic. Early experiences in the ARPAnet show that adaptive routing protocols which react to congestion can lead to persistent oscillations and instability [110]. Stability remains a challenge for the success of any adaptive protocol, particularly when it has to respond quickly [157].

Figure 3-1 shows an example of how oscillations might occur in a distributed adaptive load balancing protocol. There are two TeXCP agents, one controls the traffic from Seattle to Boston, and the other controls the traffic from San Diego to New York. Each agent has two paths, as shown. At first, the middle path, link A-B, is under-utilized and both agents decide to shift traffic to it. Each agent moves traffic without knowing that the other is doing the same, and as a result the total traffic on link A-B becomes larger than anticipated by any of the TeXCP agents. This causes both agents to move traffic away from the middle path, leading to under-utilization; and the cycle repeats resulting in traffic oscillations. The actual situation is harder as each link could be shared by hundreds of IE pairs. Network delays and drops complicate the situation further as the TeXCP agents move traffic based on obsolete knowledge of utilization and may substantially overshoot the desired target.

Solution Idea: TeXCP addresses this oscillation problem by borrowing ideas from recent congestion control protocols, in particular XCP [107]. There is a subtle yet important connection between congestion control and load balancing. Both deal with the send rate of flows; congestion control deals with flows from senders to receivers; load balancing deals with flows from ingress to egress routers. As described above, uncoordinated actions of multiple TeXCP agents that share a path may cause traffic oscillations. Similarly, in congestion control, if multiple sources that share a bottleneck link adjust their rates independently, they can cause the utilization of the bottleneck link to oscillate. For example, if the total increase exceeds the spare bandwidth, the bottleneck link gets congested causing all the sources to back off together, which in turn leads to under-utilization [84]. Similarly to XCP, TeXCP solves the oscillation problem by using explicit feedback from the routers. In particular, the router at link A-B issues feedback such that the traffic increase by the two TeXCP agents never overshoots the capacity of link A-B. We now explain the feedback computation in detail.

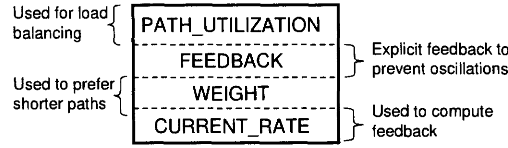


Figure 3-2 – The Probe Packet used by TeXCP. Routers update path utilization and issue feedback as per Equations 3.13, 3.14.

Solution Detail: In TeXCP, the router at the head of each link (a) computes the appropriate feedback that prevents overshooting on the link,[3] (b) divides the feedback among the IE flows traversing the link, and (c) returns a per IE-flow feedback to the TeXCP agents.

(a) Computing aggregate feedback: Every T_p, core routers compute the link utilization and the explicit feedback. Link utilization, u_l, is the average amount of traffic sent down the link in the last T_p, whereas aggregate feedback, Φ, is an upper bound on how much the *aggregate* traffic on the link should increase or decrease by. The core router makes the increase proportional to the spare bandwidth, S (S = capacity − load), and the decrease proportional to the queue size, Q. Hence,

$$\Phi = \alpha \cdot T_p \cdot S - \beta \cdot Q, \tag{3.10}$$

where α and β are constant parameters chosen according to Theorem 3.4.1 to ensure stability (see §3.4).

(b) Computing per-IE-flow feedback: Next, the core router divides the aggregate feedback, Φ, among the IE flows traversing the link. TeXCP adopts a Max–Min allocation, i.e., when all IE pairs have enough traffic demands to consume their allocated bandwidth, the bandwidth is divided equally. But, if an IE flow does not have enough demand to consume its fair share, the extra bandwidth is divided fairly among those IE flows which do have demands, and so on. Indeed by imposing a notion of fairness among IE pairs, we prevent congested IE pairs from starving others (e.g., a huge increase in the traffic from Seattle to Boston does not starve the San Diego to Boston traffic, even on the shared links).

How should a router divide the feedback among the IE flows to achieve Max-Min fairness? The standard approach is to use Additive-Increase Multiplicative-Decrease (AIMD). Note that to achieve fairness the core router needs to move traffic from over-consuming agents to under-consuming agents even if the aggregate feedback is zero, for example, when the link is completely utilized. To achieve this, a TeXCP core router always re-adjusts a small fraction γ of its total traffic load, where γ defaults to 10%. We denote the re-adjustment feedback by h.

$$h = \max\left(0, \gamma\phi_l - |\Phi|\right) \tag{3.11}$$

where ϕ_l is the aggregate load on link l. The core router, based on its congestion state, computes both an additive feedback δ_{add} and a multiplicative feedback δ_{mult} for each TeXCP

[3]To the interested reader, we note that our feedback is a simplified version of XCP's [107] feedback. Core routers distribute feedback to probe packets in contrast to XCP's use of headers in data packets. Also, core routers function with less information; they do not need the throughput/cwnd of individual flows since probes arrive at a constant rate. Also, instead of per-flow RTT, routers use T_p, a constant upper bound on RTT.

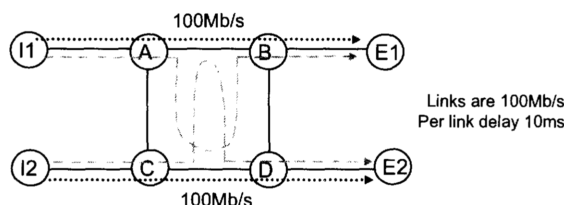


Figure 3-3 – Simple topology with many optimal traffic splits. Both IE pairs have the same traffic demands and all routings other than sending traffic on the shortest paths results in suboptimal delays.

agent:

$$\begin{aligned} \delta_{add} &= +\frac{h + \max(0, \Phi)}{N}, \\ \delta_{mult} &= -\frac{h + \min(0, \Phi)}{\phi_l}, \end{aligned} \tag{3.12}$$

where ϕ_l is the aggregate load on link l and N is the number of IE flows. Since each IE flow sends one probe every T_p, a router estimates N by counting the number of probes during T_p.[4] As we explain next, TeXCP agents using this link obtain feedback equaling $\delta_{add} + \delta_{mult} * g_{sp}$, where g_{sp} is the agent's current traffic rate.

(c) Sending the feedback to the TeXCP agents: Core routers communicate the feedback to the TeXCP agents by annotating their probes (§3.3.2). Each TeXCP agent, s, maintains an additional per path variable, g_{sp}, which denotes the agent's current traffic rate on this path and advertises this rate in the probe. The probe also contains other fields, as shown in Figure 3-2, which get overwritten by each core router along the path as follows:

$$\text{PATH_UTILIZATION} = max(\text{PATH_UTILIZATION},\ u_l) \tag{3.13}$$

$$\text{FEEDBACK} = min(\text{FEEDBACK},\ \delta_{add} + \delta_{mult} * g_{sp}) \tag{3.14}$$

When the probe reaches the egress router, the router unicasts these values to the corresponding TeXCP agent at the ingress (§3.3.2). The feedback in the probes governs the allowed rate along any path, G_{sp}, which is updated as follows

$$G_{sp} = G_{sp} + \text{FEEDBACK}, \quad \text{and} \tag{3.15}$$

$$g_{sp} = \min(G_{sp}, r_{sp}) \tag{3.16}$$

The above equation shows that the actual rate on path p, g_{sp}, is the minimum of the allowed rate, G_{sp}, and the demand, $r_{sp} = x_{sp} * R_s$. If the demands are larger than the allowed rate along a path, the extra traffic is queued at the edge router and dropped if the buffer overflows.

3.3.5 Shorter Paths First

Minimizing the maximum utilization in a network given traffic demands need not have a unique solution—i.e., there may be multiple traffic split ratios $\{x_{sp} : p \in P_s, s \in S\}$ that achieve optimal max-utilization. For example, Figure 3-3 shows two IE pairs, each with a

[4] Indeed, the router may know the number of IE flows without any computation; if paths are pinned with MPLS, then N is just the number of transit LSPs.

short and a long path. The maximum utilization does not change whether both IE pairs split traffic equally between the paths or send only on the shortest path.

We would like a TeXCP agent to choose shorter routes over longer ones as long as that choice does not increase the maximum utilization of the network. Thus, we bias the allocation of link bandwidth such that IE flows which have the link on their shortest path obtain a larger share of the bandwidth. This is easy to achieve in our system. Instead of using Max–Min fairness in allocating the explicit feedback (§3.3.4), a core routers uses Weighted Max–Min fairness, where longer paths p have smaller weights w_p. In particular, the core router replaces the sum of IE flows, N, in Equation 3.12 with a weighted sum as follows:

$$\delta_{add} = \frac{w_p}{\sum_{p' \in P_l} w_{p'}} \cdot (h + \max(0, \Phi)) \quad (3.17)$$

$$\delta_{mult} = -\frac{h + \min(0, \Phi)}{\phi_l}. \quad (3.18)$$

Note the multiplicative feedback does not change as it is already a proportional factor. We use Equations 3.17, 3.18 in all our experiments.

The TeXCP agent estimates the propagation delay d_{sp} on path p by measuring the probe RTT and assigns weight $w_p = \frac{1}{d_{sp}^3}$. An extra field in the probe packet communicates this to the core router (Figure 3-2).

3.4 Analysis

To make the analysis tractable, we adopt a fluid model and assume queues and traffic are unbounded. We also assume that each path has a single bottleneck and that the TeXCP load balancers are synchronized. Finally, we assume that IE traffic demands change at time scales much larger than the dynamics of the TeXCP load balancer ($\gg T_d$), and thus do not affect stability. These assumptions, though simplistic, are commonly used in the literature [107, 116] to make analysis manageable. Simulations in §3.5 show that TeXCP works properly even when these assumptions do not hold.

Proof Idea: Our approach is to decouple the effect of network delays from load balancing. We ensure that when a TeXCP load balancer shifts traffic from one path to another, the impact of this traffic shift on link utilizations stabilizes before any load balancer makes a new decision. If this is the case, we can analyze the stability of the load balancer ignoring network delays and assuming that traffic changes take effect instantaneously. Thus, our stability analysis has the following 3 steps.

Step 1: First, we prove that explicit feedback stabilizes the per-IE flow rates, and as a result, the utilizations are stable.

Theorem 3.4.1 *Let $d > 0$ be the round trip delay in the ISP network, if the parameters T_p, α, and β satisfy:*

$$T_p > d, \quad 0 < \alpha < \frac{\pi}{4\sqrt{2}} \quad \text{and} \quad \beta = \alpha^2\sqrt{2},$$

ISP (AS#)	Where?	Rocketfuel		PoPs	
		routers	links	cities	links
Ebone (1755)	Europe	87	322	23	38
Exodus (3967)	Europe	79	294	22	37
Abovenet (6461)	US	141	748	22	42
Genuity (1)	US	-	-	42	110
Sprint (1239)	US	315	1944	44	83
Tiscali (3257)	Europe	108	306	50	88
AT&T (7018)	US	-	-	115	296

Table 3.3 – Rocketfuel topologies used in evaluation.

then the aggregate per-IE flow on a link is stable independently of the delay, capacity, and number of IE flows.

PROOF: See Appendix §D. ■

Step 2: Second, we show that explicit feedback brings path utilization to within 90% of the desired value before the load balancer makes a new decision, i.e., by the next T_d. In particular, we pick $\alpha = 0.4$ and $\beta = 0.226$ in accordance with Theorem 3.4.1. In the worst case, the utilization has to ramp up from $u_l = 0$ to full utilization (or the opposite). For α=0.4, starting with $u_l = 0$, five iterations of Equation 3.10 cause the utilization to be more than 90%[5]. Thus, by picking $T_d = 5T_p$, the load balancer allows the explicit feedback enough time to bring the utilization to about 90% of the value dictated by the previous decision, before making a new decision.

Step 3: Finally, using steps 1 and 2 to assume that traffic shifts take effect instantaneously, we prove that independently acting TeXCP load balancers do not cause oscillations.

Theorem 3.4.2 *The TeXCP load balancer is stable and converges to a state in which every TeXCP agent sees a balanced load on all of its active paths (paths on which the agent sends traffic). Further, all inactive paths at an agent (paths on which the agent sends no traffic) have higher utilization than the active paths.*

PROOF: See Appendix §E. ■

Note that Theorem 3.4.2 proves the system is stable, shows no oscillations, and balances the load, but does not prove optimal max-utilization. An example scenario wherein preferring shortest paths is necessary for the load-balancer to reach the optimal solution is in Appendix §F. We conjecture that giving preference to shorter paths, as we do, is important for achieving optimality. Simulations in §3.5 with multiple tier-1 ISP topologies, many different traffic matrices and changing network conditions, show that TeXCP is always within few percent of the optimal max-utilization, and is much closer to optimal than alternative approaches.

3.5 Performance

We evaluate TeXCP and compare it with prior work.

[5]In 5 iterations, a spare bandwidth of 100% goes to $(1 - 0.4)^5 < 10\%$.

Technique	Description	Distributed Computation?	Reacts to changes in traffic?	Robust to failures?
Oracle	LP solution for multi-commodity flow	No	No	No
TeXCP	in §3.3	Yes	Yes	Yes
OSPF-TE$_{Base}$	Optimal Link weights for one TM [85]	No	No	No
OSPF-TE$_{Failures}$	Optimal weights for a few failures [86]	No	No	Limited number of anticipated failures
OSPF-TE$_{MultiTM}$	Optimal weights over multiple TMs [85]	No	Optimizes over multiple demands	No
MATE	Described in [78]	Sim. needs global knowledge	Yes	Yes
InvCap	Common Practice	—	No	No

Table 3.4 – Various load balancing techniques.

3.5.1 Topologies & Traffic Demands

Internet Service Providers (ISPs) regard their topologies and traffic demands as proprietary information. Thus, similar to prior work [51, 139], we use the topologies inferred by Rocketfuel [34] shown in Table 3.3. To obtain approximate PoP to PoP topologies, we collapse the topologies so that "nodes" correspond to "cities". Rocketfuel does not provide link capacities; so we assign capacities to links based on the degree distribution of the cities. There is a marked knee in the degree distribution—i.e., cities are either highly connected (high-degree) or not. The high degree cities are probably Level-1 PoPs [92], with the rest being smaller PoPs. We assume that links connecting Level-1 PoPs have high capacity (10Gb/s) and that the others have smaller capacity (2.5Gb/s). This is in line with recent ISP case studies [9, 92].

Similarly to [51], we use the gravity model to compute estimated traffic matrices (TMs). This approach assumes that the incoming traffic at a PoP is proportional to the combined capacity of its outgoing links. Then it applies the gravity model [147] to extrapolate a complete TM. The TMs used in our experiments lead to max-utilizations in the range 25-75%. For lack of space, we omit similar results for bimodal TMs [51] and topologies generated using GT-ITM [2].

3.5.2 Metric

As in [51], we compare the performance of various load balancing techniques with respect to a particular topology and traffic matrix (TM) using the ratio of the max-utilization under the studied technique to the max-utilization obtained by an oracle, i.e.,:

$$Metric = \frac{\text{max-utilization}_{\ Tech.}}{\text{max-utilization}_{\ Oracle}}. \tag{3.19}$$

3.5.3 Simulated TE Techniques

We compare the following techniques (see Table 3.4):

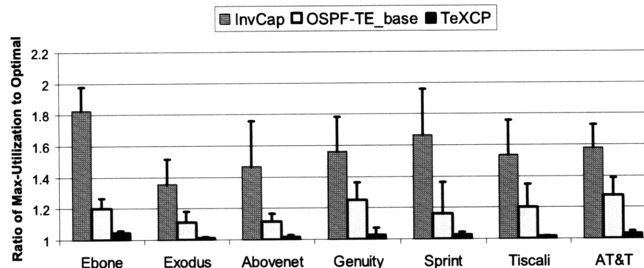


Figure 3-4 – When traffic matches TM, TeXCP results in a max-utilization within a few percent of the optimal, and much closer to optimal than OSPF-TE or InvCap. Figure shows both average (thick bars) and maximum (thin bars) taken over 40 TMs.

(a) Oracle: As the base case for all our comparisons, we use Matlab's linprog solver to compute the optimal link utilization for any topology and traffic matrix. This is the standard off-line centralized oracle which uses instantaneous traffic demands and solves the multi-commodity flow optimization problem [124].

(b) TeXCP: We have implemented TeXCP in ns2 [6]. The implementation is based on Equations 3.5 and 3.18. The TeXCP probe timer is set to $T_p = 0.1$s, and thus $T_d = 0.5$s. TeXCP uses the constants $\alpha = 0.4$ and $\beta = 0.225$ as per Theorem 3.4.1. The processing time of a probe at a core router is uniformly distributed in [0,2]ms, consistent with Internet measurements of the delay jitter for packets processed on the slow path [91]. Packet size is 1KB, and buffers store up to 0.1s worth of packets.

(c) OSPF-TE: We implemented 3 versions of the proprietary OSPF weight optimizer. The first such approach, which we call **OSPF-TE**$_{Base}$, is from [85]. Given a traffic matrix, it searches for link weights that result in low max-utilization.[6] The second version, **OSPF-TE**$_{Failures}$, computes link weights that result in low max-utilization both for the base case and also when a few critical failures happen [86]. The third version, **OSPF-TE**$_{Multi-TM}$, simultaneously optimizes weights for multiple traffic matrices. Our implementation gives results consistent with those in [85, 86].

(d) MATE: We compare the performance of TeXCP with MATE, a prior online TE protocol [78]. MATE's simulation code is proprietary. Therefore, we compare TeXCP against MATE's published results [78], after consulting with the authors to ensure that the simulation environments are identical.

(e) InvCap: A common practice that sets the weight of a link as the inverse of its capacity and runs OSPF [4].

3.5.4 Comparison With the OSPF Optimizer

We would like to understand the performance gap between online and offline traffic engineering. No prior work provides a quantitative comparison of these two approaches. Hence, in this section, we compare TeXCP with the OSPF weight optimizer (OSPF-TE), one of the

[6]The goal of an OSPF-TE scheme is to obtain a set of link weights that result in low max-utilization if traffic were sent along the shortest weight paths (and spread evenly when multiple paths have the same weight). For the sake of tractability, these schemes minimize the total cost in the network rather than the max-utilization; where cost is assigned to each link based on a piece-wise linear function of the link utilization [85].

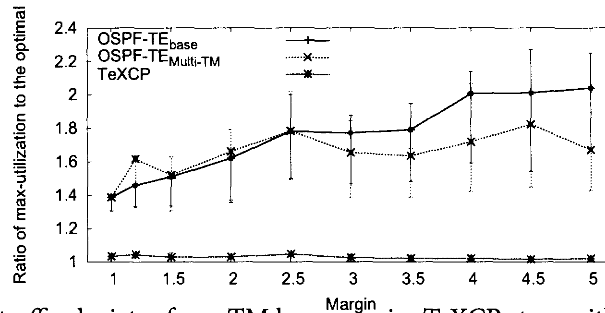


Figure 3-5 – When traffic deviates from TM by a margin, TeXCP stays within a few percent of the optimal max-utilization; OSPF-TE$_{Base}$ and OSPF-TE$_{Multi-TM}$ lead to much larger max-utilization.

more sophisticated and well studied offline TE techniques [85, 86]. Given a topology and a traffic matrix, OSPF-TE computes a set of link weights, which when used in the OSPF intra-domain routing protocol produce a routing with low max-utilization. We also compare against InvCap, a common practice that sets link weights to the inverse of link capacity.

(a) Static Traffic: First, we investigate the simplest case in which IE traffic demands are static, i.e., the actual realtime traffic completely matches the long term demands in the TM.

Figure 3-4 plots the ratio of the max-utilization under a particular technique to the optimal max-utilization for that topology and TM. The figure shows both the average taken over 40 different TMs and the maximum value. The figure shows that using a TE technique such as OSPF-TE or TeXCP can substantially reduce the network utilization. For example, for the AT&T network, using the default InvCap weights produces a network utilization that is on average 60% higher than the optimal. Using OSPF-TE produces an average network utilization that is 20% higher than the optimal, whereas with TeXCP the value is less than 5% higher than the optimal. These results have direct implications on the required link capacity–i.e., under static demands, a network that runs TeXCP can support the same required demands, under the same utilization, but with 55% less capacity than a network that uses InvCap weights and 15% less capacity than a network that uses OSPF-TE.

One might wonder why OSPF-TE does not achieve optimal max-utilization even though the traffic demands are static. Indeed, finding optimal link weights that minimize the max-utilization is NP-hard. OSPF-TE uses a heuristic to search for a good weight setting [85] but is not guaranteed to find the optimal one. TeXCP stays within a few percent of the optimal max-utilization; the small deviations from optimal are likely caused by the limited number of paths.

(b) Deviation from Long Term Demands: OSPF-TE does not re-balance the load when the realtime demands deviate from the long term averages provided in the TM. Theoretically, one can compute new weights using the new traffic demands, but in reality computing new weights takes time and accurately estimating changing traffic matrices is difficult [85]. More importantly, the new weights have to be pushed to the routers and the OSPF routing protocol has to rerun and converge to new routes. Rerunning OSPF with new weights may cause transient loops and substantial congestion [58] before routes converge, so ISPs avoid it if possible [92].

To cope with deviations from the TM, OSPF-TE uses a special tweak which we call OSPF-TE$_{Multi-TM}$. This technique optimizes the link weights for multiple traffic matrices at once

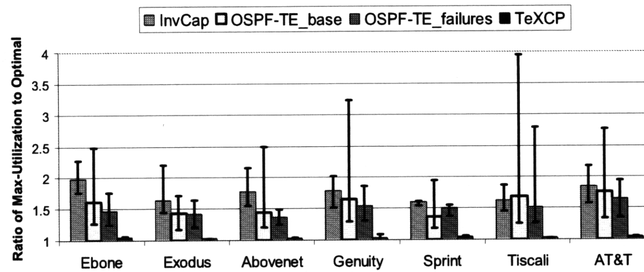


Figure 3-6 – Under failures, TeXCP's max-utilization is within a few percent of the optimal; InvCap, OSPF-TE$_{Base}$, and OSPF-TE$_{Failures}$ become highly suboptimal. Figure shows the $90th$ percentile (thick) and maximums (thin) taken over all link failures and over multiple TMs.

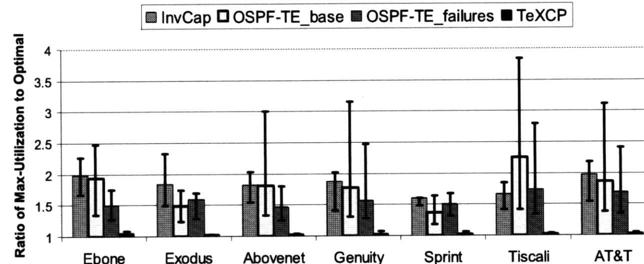


Figure 3-7 – Under multiple link failures, TeXCP's max-utilization is within a few percent of the optimal; InvCap, OSPF-TE$_{Base}$, and OSPF-TE$_{Failures}$ become highly suboptimal. Figure shows the $90th$ percentile (thick) and maximums (thin) taken over hundred three link failures and over multiple TMs.

(e.g., the peak hour TM, the off-peak TM, ...).

Figure 3-5 compares the max-utilization under OSPF-TE$_{Base}$, OSPF-TE$_{Multi-TM}$, and TeXCP as the actual traffic demands deviate from the long term average demands expressed in the traffic matrix. The figure is for the AT&T topology. The x-axis is the deviation margin; e.g., a margin of 1.5 means that the actual demands are randomly chosen to be at most 50% away from the long term demands [51]. The graph shows the average and standard deviation, over 40 TMs with that margin. The figure shows that as the actual traffic deviates from the traffic matrix (long term demands), the performance of OSFP-TE$_{Base}$ degrades substantially. This is expected as OSFP-TE is an offline protocol that cannot react to changing demands. In contrast, TeXCP reacts in realtime to the actual demands and its performance is always near optimal independent of the margin. Further, OSPF-TE$_{Multi-TM}$ is only marginally better than OSPF-TE$_{Base}$.

(c) Failures: ISPs provision their network with enough capacity to support the demands if a failure occurs (i.e., max-utilization stays $\ll$ 100%). We investigate the amount of over-provisioning required under various TE techniques.

Although the OSPF intra-domain routing protocol will re-route around an unavailable link, it does not re-balance the load after a failure. Thus, after a failure, the max-utilization may become very suboptimal. OSPF-TE$_{Failures}$ [86] addresses this issue by optimizing the weights over a set of critical single link failures, those that cause the largest increase in max-utilization upon failure. [7]

We compare the performance of TeXCP, OSPF-TE$_{Failures}$, and OSPF-TE$_{Base}$, under fail-

[7] Optimizing over five most critical failures, as recommended by [86], takes about one hour on a 2GHz, 4GB RAM P4 machine.

ures. Figure 3-6 plots the ratio of the max-utilization under a particular TE technique to the optimal max-utilization, for single link failures. The figure plots the $90'th$ percentile and the maximum value taken over all possible single link failures in the given topology. The figure shows that the 90'th percentile of max-utilization under a single link failure is much higher with OSFP-TE than TeXCP.[8] These results have interesting implications for capacity provisioning. The figure reveals that for the same level of failure provisioning, an ISP that uses OSPF-TE needs to buy double or triple the capacity needed under TeXCP.

Links do not fail in isolation. Physical causes such as fiber-cuts, power outages and router overheating may result in simultaneous failures of correlated sets of links, i.e., those comprising a Shared Risk Link Group (SRLG). Figure 3-7 shows the same results for the case of 3-links failing simultaneously. The 90'th percentile and the maximum value are computed over 100 different three link failure scenarios. The results are similar in nature to those in Figure 3-6. They again show that under failures, a network that runs TeXCP can support the same traffic demands as a network that runs OSPF-TE but with a half or third the capacity.

In summary:

- For the same traffic demands, a network that uses TeXCP can support the same utilization and failure resilience as a network that uses traditional offline TE, but with a half or a third the capacity. This creates a major cost reduction for the ISP.
- The max-utilization under TeXCP is always within a few percent of optimal, independent of failures or deviations from the TM. In comparison, InvCap results in an average max-utilization 60% higher than optimal in the base case, and twice as high under failures. OSPF-TE, achieves a max-utilization that is about 20% away from optimal in the base case, but is highly sensitive to failures and deviations from the TM, sometimes creating a utilization that is twice or thrice the optimal.

3.5.5 Comparison With MATE

We also compare TeXCP with MATE, a prior online TE proposal [78]. TeXCP borrows from MATE, and prior work on MPLS-TE [124, 174], the idea of building multiple ingress-to-egress tunnels and splitting traffic among them. TeXCP, however, differs from MATE in a few important aspects. First, the TeXCP load balancer (Equation 3.5) minimizes max-utilization while MATE minimizes the sum of the delays in the network. Unless the network is congested, the delay on a path is constant and equal to propagation delay. Second, TeXCP is fully distributed whereas MATE's simulations assume that ingress nodes have instantaneous knowledge of the whole network state. On the other hand, MATE does not need the core routers to report link utilization to ingresses.

We compare the performance of TeXCP with MATE. MATE's simulation code is proprietary. So, we compare TeXCP against MATE's published results [78], which we reproduce here for convenience. We run TeXCP in the simulation environment reported in [78]. In particular, we use the same topology, simulate traffic demands similarly using Poisson sources, use the same cross-traffic, and split traffic using the hashing method described in [78]. The topology, shown in Figure 3-8, consists of 3 ingress-egress pairs sharing 3 bottleneck links.

[8] The averages over all link failures, represented by the lower end of the thin error bars, also show the same trends. We highlight the 90'th percentiles as ISPs try to over-provision for most failures.

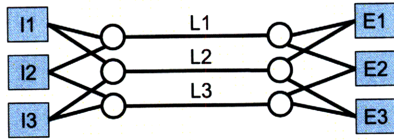


Figure 3-8 – Topology for comparing TeXCP against MATE.

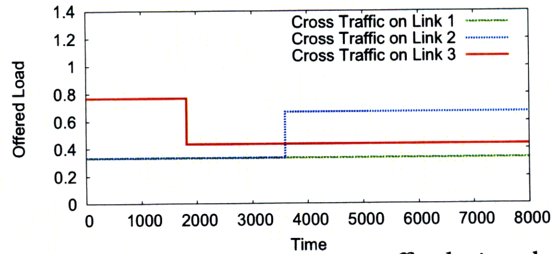
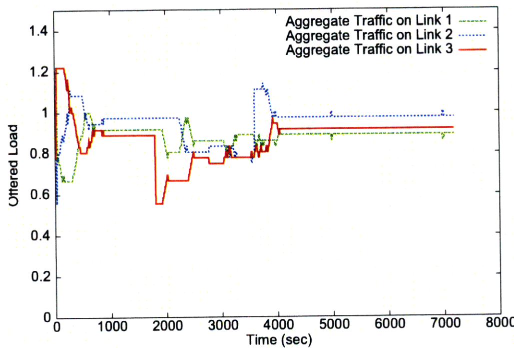
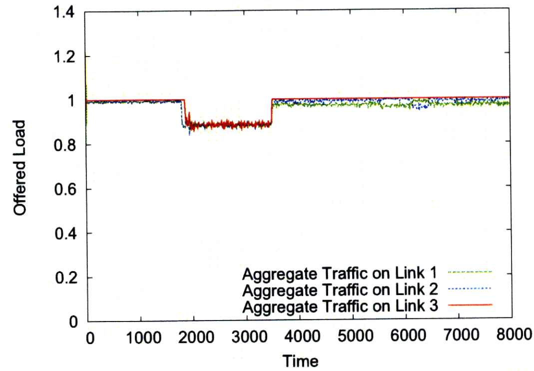


Figure 3-9 – Changes in cross traffic during the MATE simulation.



a) MATE, reproduced from [78]



b) TeXCP

Figure 3-10 – Comparison between the performance of TeXCP and MATE over the topology in Figure 3-8 and for the cross traffic in Fig 3-9. TeXCP converges faster to the optimal balance and exhibits a smoother curve.

Each bottleneck carries cross traffic uncontrolled by TeXCP. Figure 3-9 shows changes in cross traffic during the simulation. All simulation parameters, such as capacities, delays, queue sizes, number of bins etc., have been confirmed by one of the authors of MATE. Our results, in Figure 3-10, show that TeXCP is more effective at balancing the load than MATE and converges faster.

3.5.6 TeXCP Convergence Time

As with any online protocol, responsiveness is a key factor. How long does it take for TeXCP to re-balance the load after a failure or a traffic disturbance? The convergence time depends on many factors such as the topology, the maximum number of paths per IE pair, how far the initial traffic splits are from the balanced ones, etc. Figure 3-12 shows the CDF of the convergence time for the Sprint topology, where each IE pair is allowed a maximum of $K = 10$ paths. The graph shows convergence time in terms of the number of TeXCP iterations, i.e., number of T_d intervals. We generated the various samples by using 200 different TMs and starting at 20 different random initial traffic split ratios. The figure shows that TeXCP takes only 10-15 iterations to converge to within 10% of the optimal max-utilization, and a few dozens of iterations to converge to 5% of the optimal. Other topologies show similar trends.

Furthermore, TeXCP converges smoothly without oscillations as predicted by the anal-

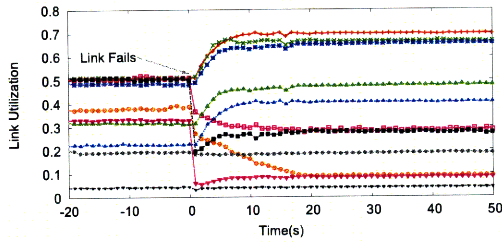


Figure 3-11 – Changes in link utilization during TeXCP convergence for a representative sample of links in the Sprint topology. The individual link utilizations converge smoothly without oscillations.

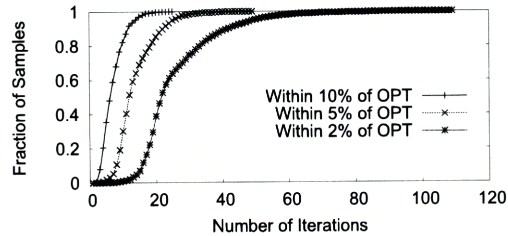


Figure 3-12 – The CDF of TeXCP convergence time for Sprint. TeXCP quickly converges to within 5% to 10% of the optimal.

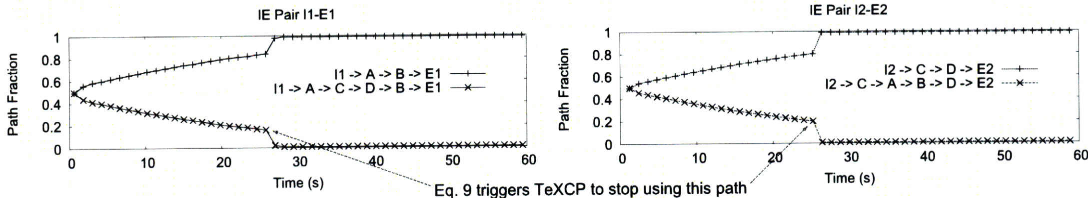


Figure 3-13 – As long as optimality is preserved, TeXCP stops using the longer paths and shifts all traffic to the shorter paths.

ISP (AS#)	# of paths used	
	avg	std
Ebone (1755)	4.275	1.717
Exodus (3967)	4.769	1.577
Abovenet (6461)	4.653	2.038
Genuity (1)	4.076	1.806
Sprint (1239)	4.175	1.935
Tiscali (3257)	4.525	1.980
AT&T (7018)	3.976	1.785

Table 3.5 – Though a TeXCP agent is configured with a maximum of K =10 paths, it achieves near-optimal max-utilization using many fewer paths.

ysis in §3.4. Figure 3-11 shows the link utilizations for a representative subset of the links in the Sprint topology when an unforeseen event (link failure) happens. It shows that the individual link utilizations steadily converge without oscillations. TeXCP experiments in §3.5.4 for other topologies and traffic demands show similar trends.

Finally, unlike OSPF convergence which might cause transient loops, the convergence time of TeXCP only affects how far we are from optimal utilization; no loops occur during that interval.

3.5.7 Number of Active Paths

Although we configure each TeXCP agent with the 10 shortest paths, TeXCP does not use all of these paths. It automatically prunes paths that do not help in reducing max-utilization. Table 3.5 shows the average number of paths used by each IE pair for the simulations in §3.5.4. In our simulations, on average, a TeXCP agent uses only 4 of the 10 paths that it was configured with.

3.5.8 Automatic Selection of Shorter Paths

We show an example simulation of how TeXCP automatically prunes longer paths, if they do not reduce the max-utilization. We simulate TeXCP on the simple network in Figure 3-3. The two IE pairs (I1, E1) and (I2, E2) have one unit of traffic demand each. There are two TeXCP agents, one for each I-E pair. Each TeXCP agent has one short path and one long path. Note that the max-utilization is the same regardless of whether the agents send all of their traffic on shorter paths or split it evenly between the two paths. Figure 3-13 shows the fraction of traffic each agent sends on the two paths. Despite being initialized to send equal traffic on the long and short paths, both TeXCP agents quickly prune the longer paths and prefer to route all traffic on the shorter paths.

3.6 Implementation & Deployment

This section examines some practical issues that an ISP would consider prior to deploying TeXCP in their routers. For this section, we assume IE paths are pinned using MPLS Label Switched Paths (LSPs) [146].

(a) Number of LSPs: TeXCP creates K LSPs for each IE pair ($K = 10$ in our experiments). It is common for large ISP backbones to have 200-300 egress points, resulting in 2K-3K LSP *heads* at an ingress router. This is somewhat higher than typically supported in today's backbone routers.[9] We observe, however, that ISPs have many fewer PoPs, 20-100 (Table 3.3). Since traffic on the ISP backbone essentially moves from one PoP to another, TeXCP tunnels need only be created between PoPs rather than between ingress-egress routers. This commonly used technique in MPLS networks [92] brings the number of LSP heads into line with current technology. We also note that a router can typically support an order of magnitude more *transit* MPLS LSPs than LSP *heads* because transit LSPs need smaller state and processing overhead.

(b) Traffic Splitting: TeXCP requires edge routers to split traffic among the LSPs connecting an IE pair. Current ISP-class routers can split traffic to a destination between as many as 16 LSPs, at as fine a resolution as desired [131].

(c) Utilization Estimates: TeXCP needs routers to estimate link utilization every T_p seconds. Current edge and core routers maintain counters of the amount of traffic (bytes and packets) sent on each interface. These counters can be read every T_p to get a utilization estimate. T_p=100ms allows counters enough time to stabilize.

[9] Official numbers of supported tunnels are hard to find in vendors' public documentation and tend to increase over time. One Cisco router used in ISP networks supports at least 600 tunnel heads, back in 2005.

(d) Estimating Feedback: Estimating the feedback is as easy as estimating the utilization. Equation 3.12 shows that all IE flows have the same feedback, be it positive or negative. A router needs to compute this one value for each outgoing interface, once every T_p. Also, the core router can offload the computation entirely, by pushing the variables (S, Q, N, ϕ_l) to the edges.

(e) Communication between the core and edge routers: On the first cut, each TeXCP agent sends one *probe/*T_p down all its LSPs. For small networks $n \in [30, 100]$ and low probe rates $T_p \in [.5, 1]s$, this results in a core router processing $1.2 - 8K$ probes/sec/interface, and may be feasible. For larger networks, this is just too many.

Instead, for large networks we suggest a cleaner approach. Each core router generates one *report* packet/T_p, for each outgoing interface, which contains the link utilization and the additive and multiplicative changes. Recall from Equations 3.17, 3.18, that all IE flows traversing the link receive the same changes, and a TeXCP agent can calculate its feedback based on its current rate. Thus, unicasting the report packet to every edge router is equivalent to updating probes of every LSP. Note, a core router only generates n reports/T_p, where n is the number of edge routers, for each outgoing interface. As in §3.6(a), we note that using PoP to PoP LSPs substantially reduces the amount of traffic. Also note that the edge routers need to do a little more work to consolidate per-link reports into per-LSP state, but that seems a good tradeoff.

(f) Router Modification: TeXCP needs one new functionality; the load balancer (Equation 3.5), which can be easily built in software in existing edge routers. All other functionality either already exists in current routers or is very similar to existing functionality.

3.7 Related Work

Several offline TE methods, like the OSPF weight optimizer [85, 86] and the multi-commodity flow formulation [124] were described in §3.5. Here we mention other relevant efforts.

Optimal Routing: Gallager's seminal work on minimum delay routing [89] began the field of optimal and constraint-based routing [59, 62, 78, 124, 169, 177]. This work studies routing as an optimization problem and usually relies on centralized solutions, global knowledge of the network state, or synchronized nodes.

Offline Oblivious Optimizers: Instead of estimating traffic matrices accurately [81], the MPLS Oblivious optimizer [51] finds a routing that balances load independent of the traffic matrix, i.e., minimizes the worst case across all possible traffic matrices. Similarly, two new schemes [112, 188] balance load in a traffic-insensitive manner by re-routing traffic through pre-determined intermediate nodes; an idea used in processor interconnection networks. Post-dating our work, COPE [179] creates multiple linear programs which given a topology and a set of different possible traffic demands, finds a routing that is no worse than optimal by a certain penalty value. The primary difference is that TeXCP is independent of the actual demands, and adapts in real-time to both traffic spikes and link failures. Further, it is not clear how COPE scales, for example, COPE involves solving a master LP, with one slave LP for each link in the topology [179], and it is unspecified how long it takes to solve this group of optimizations in practice.

Online Load Balancers: There are relatively few proposals for realtime adaptive multipath routing. We describe MATE [78] and compare it with TeXCP in §3.5.5. OSPF-OMP [173], an Internet draft, describes another technique that floods the realtime link load information and adaptively routes traffic across multiple paths.

3.8 Further Discussion

(a) TCP Re-ordering: Wouldn't multipath routing, especially if it moves traffic around in an online fashion, reorder TCP packets and hurt TCP congestion control? Typical approaches to split traffic across multiple paths either keep per-flow state to ensure that a flow would remain on the same path, or hash TCP flows into many buckets and assign buckets to the various paths [7, 8]. Neither of these approaches is satisfying. The former approach finds it hard to move traffic, because it is constrained to not move all the pre-existing flows. The latter approach still causes TCP reordering when it reassigns a bucket to a different path. A clean and complete online TE solution has to solve two problems. The first is the intrinsic problem of being responsive yet stable, which we address in this chapter. The second is to dynamically split flows without reordering TCP packets, which we address in our Flare paper [156]. We show that splitting TCP flows at the granularity of flowlets–bursts of packets within a TCP flow–balances load accurately without reordering packets and without keeping per-flow state.

(b) Utilization vs. Delay: Would a scheme that minimizes the max-utilization, like TeXCP does, excessively delay traffic by routing data over very long paths? We note that minimizing the maximum utilization in the network is a widely-used TE metric in research and practice [51, 52, 124]. It removes hot-spots and reduces congestion risks by spreading the load over available paths and allows a network to support greater demands. Of course, minimizing max-utilization may sometimes increase the delay. But, unlike contemporary TE [51, 85], TeXCP allows an ISP to bound this increase in delay by specifying which paths a TeXCP agent is allowed to use. For example, when an ISP operator configures the Boston-LA TeXCP agent with some paths, he is declaring that all of these paths have acceptable delays. Clearly the operator should not pick a path that goes to LA via Europe; and he does not need to, given the diversity in ISP topologies [168]. Using the Rocketfuel delay estimates, we have computed the average delay difference between the paths used by TeXCP and the shortest path weighted by the fraction of traffic on each path. For the experiments in §3.5.4, this number is in the [3.2, 10.6] ms range. Also, we re-ran the experiments in Figure 3-4 by restricting TeXCP to paths that are no more than $20ms$ longer than the shortest. In these runs, the weighted average delay difference goes down to [1.2, 2.7] ms. The max-utilization stays similar to Figure 3-4, except for the AT&T topology where it increases by a few percent.

(c) Interaction with BGP: Does an intra-domain traffic engineering scheme, like TeXCP, interact adversely with BGP, the inter-domain routing protocol? TeXCP has no adverse interaction with BGP, i.e., nothing bad happens even if every ISP uses TeXCP? The same traffic volume enters and exits an ISP at the same points. TeXCP only balances traffic *within* an ISP and has no globally-visible changes.

(d) Interaction with Failure Recovery: TeXCP does not replace failure discovery and link restoration techniques, such as SONET rings, DWDM optical protection and MPLS fast-

reroute [92]. Rather, TeXCP complements these techniques; it rebalances the load after a link fails and also after the link restoration technique re-routes traffic. Further, TeXCP helps in recovering from unanticipated or combination failures for which the ISP may not have a pre-computed backup path. For example, about half the ISPs run MPLS in their core. When a link fails, MPLS fast-reroute quickly patches the failed MPLS tunnel with an alternative segment, shifting the traffic to different physical links. This may unbalance the traffic and create hot spots. TeXCP rebalances the traffic allowing the recovery process to go smoothly.
(e) Centralized Online Traffic Engineering: It is fair to ask whether online traffic engineering needs to be distributed. To adapt load distribution when traffic spikes or when links fail, a centralized TE scheme needs to do a couple of different things. First, it needs to obtain real-time estimates of the traffic demands at all the ingresses, i.e., the traffic matrix (TM) and the up-down status of all links in the network [64, 123, 147, 187]. Second, the centralized controller needs to quickly re-compute an optimal routing for the current traffic matrix and network topology [63]. Third, the new optimal routing has to be pushed down to every router that forwards traffic in the network [182]. Finally, all routers in the network have to shift to the new routing in lock-step to avoid formation of routing loops from stale routes [162]. Each of these challenges has been documented earlier as hard to achieve. Further, to avoid single points of failure, centralized schemes typically replicate the centralized controller that computes routes and have to face the subtle challenges involved in synchronizing the replicas, i.e., they need to ensure that all replicas observe the same state of the network and react similarly. Post-dating our work, Tesseract [182] and RCP [63, 80] make progress towards addressing these challenges.
(f) Interaction with Reactive Transport Protocols: Wouldn't online traffic engineering interact adversely with end-to-end reactive mechanisms like TCP that adapt their rates based on the available bandwidth on the path? This is a subtle question, and one that we believe needs more work to answer. Anderson et. al. [49] present a theoretical analysis that no adverse interaction would happen as long as the TE scheme satisfies some simple conditions, for example, that the TE scheme should distribute traffic fairly among users.

3.9 Concluding Remarks

This chapter advocates online traffic engineering, a technique to dynamically adapt the routing when traffic changes or links fail, rather than optimizing the routing for long term average demands and a pre-selected set of failures. We present TeXCP, an online distributed TE protocol, show that it is stable and that it balances the load and keeps network utilization within a few percent of the optimal value. We also show that TeXCP outperforms traditional offline TE, particularly when the realtime demands deviate from the traffic matrix or unanticipated failures happen.

Although this chapter focuses on traffic engineering, we note a larger trend away from the traditional single-path, congestion-insensitive, Internet routing and towards adaptive routing. This includes overlays [48], adaptive multi-homing [45], and traffic engineering. Recent studies show that overlay routing increases traffic variability making it much harder to estimate the TM needed for offline TE optimization [108, 139]. In that sense, our work complements recent work on overlay routing because online TE does not need to estimate

TMs. Ideally, one would like online TE to balance the load within ISP networks, exploiting intra-AS path diversity [168] to avoid intra-AS bottlenecks (which, according to [46], account for 40% of the bottlenecks). Overlays, on the other hand, are good at adapting end-to-end routes to avoid congested peering links.

Chapter 4

Learning Communication Rules

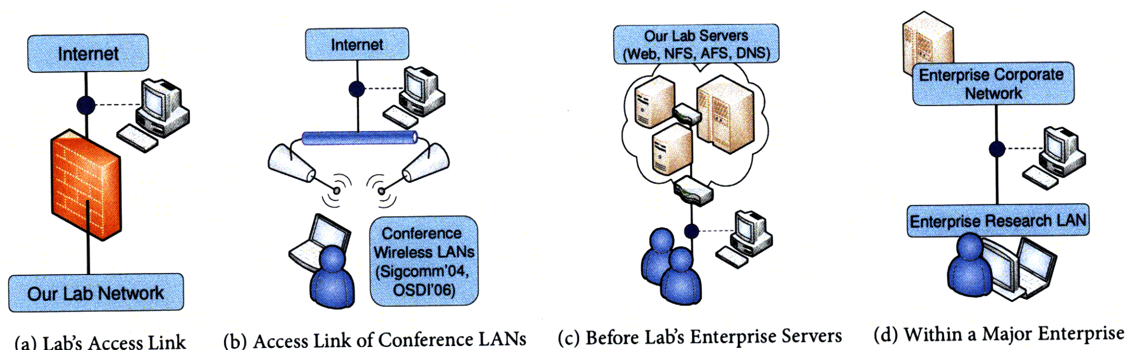
In this chapter, we shift focus to learning dependencies in edge networks, i.e., campus and enterprise-wide networks. Existing traffic analysis tools focus on traffic volume. They identify the heavy-hitters—flows that exchange high volumes of data, yet fail to identify the structure implicit in network traffic—do certain flows happen before, after or along with each other repeatedly over time? Since most traffic is generated by applications (web browsing, email, p2p), network traffic tends to be governed by a set of underlying rules. Malicious traffic such as network-wide scans for vulnerable hosts (mySQLbot) also presents distinct patterns.

This chapter presents eXpose, a technique to learn the underlying rules that govern communication over a network. From packet timing information, eXpose learns rules for network communication that may be spread across multiple hosts, protocols or applications. Our key contribution is a novel statistical rule mining technique that extracts significant communication patterns in a packet trace without explicitly being told what to look for. Going beyond rules involving flow pairs, eXpose introduces templates to systematically abstract away parts of flows thereby capturing rules that are otherwise unidentifiable. Deployments within our lab and within a large enterprise show that eXpose discovers rules that help with network monitoring, diagnosis, and intrusion detection with few false positives. [1].

4.1 Overview

Perhaps the single defining aspect of edge networks today is that they are complex to manage. Today's enterprise and campus networks are built from multiple applications, protocols and servers which interact in unpredictable ways. Once the network is set-up, there are few tools that let an administrator keep track with what is going on in the network. Configuration errors seep in, software gets upgraded and servers get phased out leaving the administrator with the unenviable job of ensuring that traffic in the network conforms to a plan. Of course, scripting cron jobs and correlating server logs to figure out what's going on is a tedious option that does not scale [142].

[1] Throughout this chapter, our lab refers to the CSAIL lab at MIT, and large enterprise refers to the Microsoft Enterprise Network.



(a) Lab's Access Link (b) Access Link of Conference LANs (c) Before Lab's Enterprise Servers (d) Within a Major Enterprise

Figure 4-1 – We evaluated eXpose on traces collected at each of the locations above (shown by the blue circle). The locations included *access links* at a large university and at two different conference wireless LANs and, *links carrying enterprise traffic* towards servers at a university and at a large company.

We advocate an alternative approach to manage the complexity in edge networks. Suppose we shift focus away from individual servers and configurations and focus directly on packets on the wire. Suppose that from a packet trace, we could learn communication rules that underlie a majority of the user activities present in that trace. Such a broad picture would provide an administrator with a reality-check; he can see how traffic for major applications traverses the network, identify configuration errors or unwarranted communication, and perhaps detect malicious activity. This chapter is a first step towards this goal.

Existing trace analysis tools however cannot reveal to an administrator the communication patterns in his network. They focus on traffic volume and do not reveal the implicit structure in edge network traffic. Tools like MRTG [130] and NetFlow Analyzers focus on the heavy hitters, i.e., flows that exchange a lot of data. They may report that 30% of the traffic is to the web-server and that 70% of the traffic uses TCP. Advanced tools like AutoFocus [79] adapt their granularity of search. They can reveal IP subnets (e.g., 10.0.0.0/8) or port ranges that contribute lots of traffic.

This chapter introduces eXpose, a new analysis technique that extracts significant communication rules in a network trace, without being told what to look for. A communication rule is an *implies relationship* such as $Flow_i.new \Rightarrow Flow_j.new$ indicating that whenever a new $Flow_i$ connection happens, a new $Flow_j$ connection is likely to happen. For example, eXpose would deduce rules like a DNS connection often precedes new connections, an AFS client talks to the root-server (port 7003) before picking files up from the appropriate volume-servers (port 7000), an End-Point-Mapper RPC call precedes mail fetch from Microsoft Exchange Servers, and viruses such as the mySQLbot probe flood the entire network looking for vulnerable mySQL servers.

Our key insight is simple–if a group of flows consistently occurs together, the group is likely dependent. Of course, correlation does not always imply dependence, but this is the best one can do lacking knowledge of the applications and the network layout. The challenge lies in applying this insight to a network trace where millions of flows show up every few hours. To do this, eXpose selectively biases the potential rules it evaluates and does not evaluate rule types that are unlikely to yield useful information. Second, eXpose abstracts away extraneous flow details to make useful patterns more discernible. Third, eXpose uses an appropriate statistical measure to score the candidate rules and mines efficiently. Finally,

Term	Meaning
Flow	A five-tuple <local IP, local Port, remote IP, remote Port, Protocol>
Flow Activity	A flow is either *absent, present* or *new.* Note *new* $\subset$ *present.*
Rule $X \Rightarrow Y$	X, Y are tuples of <flow, flow_activity>. The rule $X \Rightarrow Y$ says that whenever flow $X.flow$ has $X.activity$, flow $Y.flow$ is more likely to have $Y.activity$ than normal.
Activity Matrix	Rows denote time windows, columns denote flows in the trace. Each entry holds the activity of a flow (column) in the corresponding time window (row)

Table 4.1 – Definitions used in this chapter

eXpose aggregates the discovered rules into a small number of useful clusters that an administrator can corroborate and use.

Not all dependencies are visible at the granularity of a flow. For example, suppose whenever a client talks to a sales server, the server fetches data from a backend database. Yet no individual client accesses the server often enough to create a significant rule. To capture such rules that are otherwise unidentifiable, eXpose introduces templates that systematically abstract away parts of flows. For example, one of our templates replaces the client's IP with a wild-card character creating a *generic* whenever any client talks to the sales server. Leveraging these generics, eXpose searches for rules like $* : Sales \Rightarrow Sales : Database$, meaning whenever *any one* of the clients talks to the Sales server, the Sales server talks to its Database. We show how to apply templates without any prior knowledge about the hosts involved in the trace and find that these templates improve the expressiveness of our rules considerably.

eXpose has three unique characteristics. First, eXpose extracts communication rules with zero guidance, i.e., without being told what to look for. Hence, eXpose can identify communication rules that are unknown to the administrator, including configuration errors, anomalies, and exploits. Second, while prior work [54, 103] focuses on individual applications, a single source-destination pair [103], or the dependencies for clients accessing a given server [54], eXpose learns patterns that may involve multiple applications, servers or protocols. Extracting this broad picture by applying prior techniques that learn rules one application at a time or one server at a time does not scale and may miss out on important dependencies if the admin forgets to point the tool towards the correct servers or applications. Third, eXpose is in a sense future-proof. By making minimal assumptions about applications, protocols and servers, eXpose's techniques to learn rules can deal with evolving networks.

We deployed eXpose on the server-facing links of two edge networks, in the CSAIL lab at MIT and in the Microsoft Research network for two months. We also ran eXpose on traces collected on the access-links (i.e., Internet facing links) of our lab at MIT and two conference hot-spots. We corroborated rules output by eXpose with the admins at the corresponding locations and report rules that we could not explain as false positives. Our results show that:

- eXpose discovered rules for enterprise configurations and protocols that are deployed and used in mainstream operating systems that we did not know existed, such as Nagios [26] monitors and link-level multicast name resolution [23].
- eXpose discovered rules for the major applications such as email, web, file-servers, instant messengers, peer-to-peer and multimedia distribution in each of the two edge networks.

- eXpose detected configuration errors leading to bug fixes.
- eXpose detected malicious traffic, machines infected by trojans, mysql bot scans and ssh scans targeting key routers.
- eXpose mines for rules in time that is much smaller than the duration of traces (0.1% – 23.8%). This means that although our current implementation works offline by feeding off packet traces, an online implementation is feasible.

We begin by describing eXpose's learning algorithm.

4.2 Learning Communication Rules

Our goal is the following: Given a packet trace, discover the communication rules inside the corresponding network. We define a communication rule as a dependency between flow activities (see Table 4.1). A communication rule $X \Rightarrow Y$ occurs in the trace if flow activity X implies flow activity Y. For example, X may be a new http connection from a client to a web-server, which implies Y, a new connection from the client to the DNS server. The rules themselves are probabilistic to mimic the underlying phenomena. For example, new connections trigger DNS lookups only upon a miss in the local cache. Also rules may not be symmetric, for example, HTTP connections trigger DNS but not all DNS requests are caused by HTTP, hence the rule $DNS \Rightarrow HTTP$ may not be significant. Some communication rules are at the granularity of flows such as whenever Host1 talks to Host2, Host2 talks to Host3; whereas others are more abstract, such as whenever *any client* talks to the web-server W, the web-server talks to its backend B. We call the latter *generic rules* and define simple templates for such generic rules. Our tool discovers all significant instantiations of these templates in the given packet trace. Some of the rules we discover describe the normal behavior of the network, while others identify attacks and mis-configurations in the network.

4.2.1 From Dependency to Association

How can we detect flow dependencies in a trace? Lacking knowledge of the applications and the network layout, it is impossible to say that two activities seen in a trace are dependent. The best one can do is to find correlated occurrences in the trace. At a high-level, our idea is to identify flow groups that co-occur with frequency significantly greater than chance and repeat consistently over time.

But there are simply too many flows in a trace, for example, in both the campus and enterprise network that we collected traces at (see Figure 4-1), we saw more than 10^6 flows within a few hours. Examining every pair of flows for dependence doesn't scale, let alone examining larger groups of flows.

To design a scalable solution, we make a simplifying assumption. Whether a flow is absent, present, or new at any point of time is only influenced by flows that have recently occurred. Hence, we partition the trace into time windows and look for dependencies that occur within the same time window (default window is 1s wide). This assumption lets us not worry about the actual time gap between pairs of flows, which often varies in practice; instead, it considers pairs of flows that repeatedly occur separated by a short period of time.

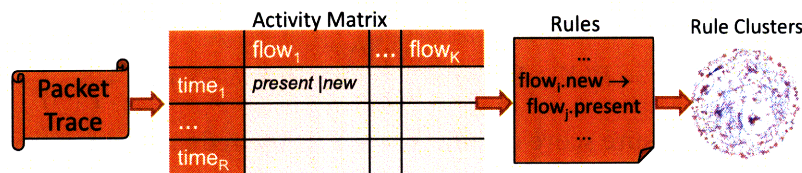


Figure 4-2 – Schematic of eXpose's work-flow.

We believe that most network dependencies are machine-generated and are indeed separated by a short period of time. For example, we learnt the communication rules for BitTorrent, viruses and, email clients (See §4.4) by looking within 1s-wide windows. A systematic evaluation of sensitivity to window size is in §4.4.5. We acknowledge, however, that some flow dependencies occur over longer periods. For example, the gap between a user reading a web-page and clicking on a link may exceed 1s. Extending our technique to look at dependencies over multiple time windows is possible but is outside the scope of this chapter.

Using this simplifying assumption, eXpose converts the input trace into an *activity matrix*. The rows of the activity matrix denote time windows in the trace, whereas the columns correspond to flows in the trace. Each entry in the matrix takes one of three values: new, $present$ or $absent$ denoting the activity of a flow (column) in a time window (row). We observe that this representation (Figure 4-2) is typical of a data-mining problem. Hence eXpose finds dependent flows by mining for association rules [74] in the activity matrix.

Flows and their activities: To be concrete, we identify a flow in terms of its source and destination IPs, ports, and protocol number. A flow is *present* in a time-window if it sends non-zero packets during that period. For connection-oriented flows like TCP that have an explicit start and end (syn, fin), a flow is *new* in the time window that overlaps its beginning. For datagram-based flows and also when traces don't have detailed headers, a flow is *new* if it becomes present after a period of inactivity. Note that if a flow is new, it is also present in that time window (*new* $\subset$ *present*).

4.2.2 Significance of a Communication Rule

How does one tell which dependencies are significant? One might be tempted to say that a communication rule, $X \Rightarrow Y$, exists if X and Y occur together often, that is if $Prob(X \wedge Y)$ is high. This intuition is incorrect, however, and causes both false positives and false negatives. Two dependent flows may always occur together in the trace, yet happen so rarely (low $Prob(X \wedge Y)$) that their dependence is not identified. On the other hand, an unrelated pair of flows can have a high joint probability simply because one of the flows is very popular.

We take an information-theoretic approach to identify statistically significant dependencies—pairs of flows that have much higher joint probabilities than merely due to chance. Specifically, we set up candidate rules of the type $X \Rightarrow Y$, where both X and Y are tuples of <flow, flow-activity> and use JMeasure [158], a known metric in the data-mining community, to score candidate rules.

$$JMeasure(X \Rightarrow Y) = I(Y; X = 1), \tag{4.1}$$

where $I(Y; X = 1)$ is the mutual information, i.e.:

$$I(Y;X=1) = P(X \wedge Y)\log\frac{P(Y|X)}{P(Y)} + P(X \wedge \neg Y)\log\frac{P(\neg Y|X)}{P(\neg Y)}.$$

Intuitively, the JMeasure score of rule $X \Rightarrow Y$ is the reduction in entropy of the random variable Y when the random variable X happens. If X and Y were independent, the JMeasure score would be zero. At the other extreme, if Y is completely dependent on X, the JMeasure takes the largest possible value, $P(X)\log\frac{1}{P(Y)}$.

Unlike other measures of statistical significance, such as the Chi-Square Test [96] and the F-Measure [96], the JMeasure score does encode the directionality of the relationship. This is crucial because we expect dependencies in practice to be uni-directional. For example, a HTTP connection may trigger a DNS connection but only a few DNS connections are due to HTTP.

Unfortunately, the JMeasure rule that works well for general data-mining comes short when identifying network dependencies for the following reasons.

(a) Negative Correlations: Reduction in entropy occurs in one of two ways. In the extreme, when X happens, Y may always happen or Y may never happen. Unfortunately, JMeasure does not differentiate between these two cases and scores both highly. While the first type of rules are interesting as they correspond to co-occurring flows which are likely to be true dependencies, the latter kind happens often in network traces and are usually not meaningful. There are so many flows that are much shorter than the duration of the trace (low $P(Y)$) that it is easy to find another flow X that happens only when Y does not occur, spuriously leading to a high JMeasure rule. To avoid such rules, we only use the positive correlation part of the JMeasure rule:

$$Score(X \Rightarrow Y) = P(X \wedge Y)\log\frac{P(Y|X)}{P(Y)}. \qquad (4.2)$$ [2]

(b) Long-Running Flows: Long-duration flows pose a challenge unique to mining network traces. Every one of our traces had flows that were contiguously active over a significant part of the trace–long downloads (FTP, SMB) and long-running shell sessions (telnet, ssh, remote-desktop). Given the prevalence of short duration flows, it is often the case that a short flow (X) happens only when a long running flow (Y) is active. The above scoring technique scores this pair highly, yielding a spurious rule. To avoid such rules we employ the following principles. First, we note that spurious rules like the above happen only when the activity on either side (both X and Y) is *present*. Long-running flows are *present* in many more time-windows than they are *new*, and a short flow that starts many times, i.e., has many *new* time-windows, is unlikely to completely coincide with a long flow. So, we prefer to report rules involving *present* activities only when there is little mis-match between the frequencies on either side: $1/3 \leq \frac{P(X)}{P(Y)} \leq 3$. Second, we do not report rules involving flows that happen in more than 90% of the time-windows of the trace. To say anything reasonable about such flows would require a longer trace.[3]

[2]Note, Equation 4.2 is symmetric, so we continue to use JMeasure to resolve directionality of rules.

[3]Examples of rules involving *present* flows are an ssh tunnel that forwards packets received from one peer to another and TOR routers involved in anonymizing overlays.

(c) Too Many Possibilities: Any statistical test, including our technique, is based on a simple principle. How likely is the null hypothesis, in this context, the hypothesis that X and Y are independent, given the score? If a rule has a score value that makes the probability of the null hypothesis vanishingly small, the rule is reported as statistically significant. Unfortunately, in network mining there is a complementary challenge. There are so many flows, and so many potential rules, that even if each null hypothesis has very small probability, overall the likelihood of false positives is quite large. eXpose selectively biases the possible rules it considers to reduce false positives. First, we note that a rule involving two flows can have upto four unique IPs–the sources and destinations of the two flows. It is unlikely that flows which have none of their ends in common are dependent (example: does IP A talk to IP B whenever IP C talks to IP D?). Hence, we only report rules involving flows that have at-least one IP address in common. Not only does this shift focus away from potential rules that are more likely to be false-positives (see §4.4.3 for evaluation), but it also greatly reduces the number of potential rules, improving scalability. Second, we note that most flows happen in such few time windows that there is too little data to predict relationships for such short flows. So, instead, we focus on the heavy tail– the K most active flows in the packet trace. K can be set by the network administrator, otherwise it defaults to 5000 flows. See §4.4.5 for an analysis of eXpose's sensitivity to K.

(d) Cut-off: eXpose only reports rules that have a significance score (Equation 4.2) greater than a threshold α which defaults to $.01 nats$. [4] To put this threshold in perspective, note that if two flows are completely dependent on each other, i.e., $P(X) = P(Y) = P(X \wedge Y)$, then both rules $X \Rightarrow Y$ and $Y \Rightarrow X$ will be output for all non-trivial values of $P(X)$. More interestingly, for general flows such as the flows from a client to a DNS and an HTTP server, the rule $HTTP \Rightarrow DNS$ will be output only when two conditions hold: (1) co-occurrence better than chance, i.e., $P(DNS|HTTP) > P(DNS)$ and, (2) frequent co-occurrence, i.e., $P(HTTP \wedge DNS) > \frac{.01}{\log P(DNS|HTTP) - \log P(DNS)}$. Note the interplay between the two conditions, the more likely a DNS flow is when an HTTP flow happens, the larger the gap between $P(DNS|HTTP)$ and $P(DNS)$, the less stringent the constraint on frequency of co-occurrence.

4.2.3 Generic Communication Rules

So far, we have been looking at dependencies between pairs of flows. But, not all dependencies are visible at this granularity. For example, suppose whenever a client talks to a sales server, the server fetches data from a backend database. Clearly this is an important dependency, yet it may be missed if no *single* client accesses the server frequently (recall that we score based on flow frequency). Our idea is to relax the granularity at which we look for dependencies. In particular, we report the above dependence as long as all the clients *together* access the server often enough. To achieve this we introduce generics.

The idea behind generics is similar to wild-cards in regular expression matching—relax some of the fields in a flow's five-tuple. As an example, whenever the flow $Client.SomePort : Sales.80$ is active in a time window, we introduce a generic $*.* : Sales.80$ as being active

[4] nats stands for units of entropy when natural logarithms are used. For example, $\log(2) = .693\ nats$.

in that time window. Further, we consider rules like

$$*.* : Sales.80 \Rightarrow Sales.* : Database.*, \tag{4.3}$$

and say that this rule happens whenever *some client* talks to Sales and Sales talks to the Database within the same time window. A more interesting example of a generic rule is:

$$*.* : WebServer.80 \Rightarrow *.* : DNS.53. \tag{4.4}$$

For rules that relax IPs on both sides, we say the rule happens in a time window only if the *missing* fields on both sides are the same. It does not matter which client(s) lead to the two relaxed flows, as long as there is *at least one client that talked to both* the web-server and the DNS in that time-window.

Templates for Generic Rules

When should generics be instantiated and how to combine them into communication rules? Note that relaxing exhaustively explodes the number of flows and rules. Instead, we define simple templates for relaxing flows and for combining the generics into rules. Whenever a flow matches a relaxation template, we instantiate the corresponding generic. Rule templates prescribe which combinations of generics and flows are worth considering as a communication rule. We introduce one relaxation template and two rule templates. Our tool automatically learns all significant rules that are formed through instantiations of these templates in the packet trace.

Relaxation Template to Create Generics: Whenever one end of a flow is a well-known port (80 for http, 53 for DNS, /etc/services has exhaustive list), we relax the IP at the opposite (client) end. The idea is to create a generic that abstracts away the client's IP and focuses on all accesses to the server. Instead of relying only on the standard list of well-known server ports, we learn from each trace the ports used by a large number of flows in that trace and consider them well-known. This lets us learn the ports used by peer-to-peer applications. Finally, ports that are not in the well-known list are considered the same for the purpose of matching.

Rule-Templates to build Rules from Generics and Flows: Our first template helps to identify server's backend dependencies. Analogous to the example in Equation 4.3, this template allows rules that combine a flow with a generic if the un-relaxed IP in the generic (i.e., the IP corresponding to the server) matches one of the IPs in the flow. The second template identifies dependencies that are visible to a client. Analogous to the example in Equation 4.4, this template allows rules involving two generics. Such a rule is active in a time window only if at-least one client accesses both the server IPs/Ports in that time window.

Scoring Generic Rules

Rules involving generics, such as the example in Equation 4.4, become more interesting as more unique clients conform to the rule. Hence, we supplement statistical significance by a *support* metric. The support of a rule involving one or more generics is the number of

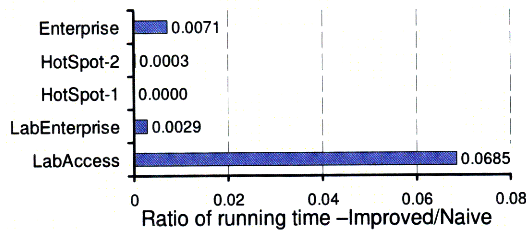


Figure 4-3 – Measurement results compare the computational cost of eXpose's mining algorithm with that of a baseline.

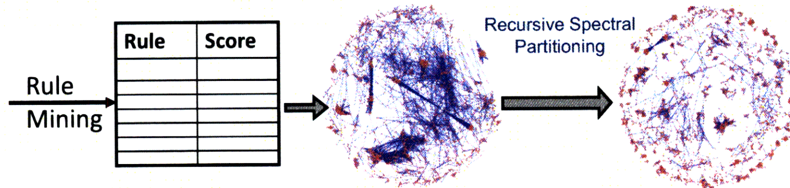


Figure 4-4 – Composing and Pruning Rules into Rule Clusters. The table on the left depicts the output of the rule miner, the graph in the middle represents rules discovered by mining. Each edge corresponds to a discovered rule and joins nodes corresponding to the <flow, activity> pair involved in the rule. The graph on the right depicts the same rule-set after recursive spectral partitioning. Eliminating low-scored rules between node-groups that are otherwise strongly connected breaks the rule-set into understandable pieces.

unique clients whose communication can be abstracted in the form of the rule. Clearly, a generic rule with support 1 is a trivial generalization. Hence, we only report generic rules that have support greater than a threshold β (default 3).

4.3 Algorithms to Mine for Rules

So far we introduced generics and the scoring function for statistical significance but how do we mine for communication rules? The costliest part of rule-mining, in our context, involves computing how often each candidate rule occurs in the trace. For a simple rule that involves only two flows, we need to count the time-windows when both flows happen; if the rule involves generics, we need to count time windows that contain flows matching the generics as described above.

Our key contribution here is a more efficient way to compute frequencies for all potential rules. Recall that we focus on the top K active flows (and generics) after discarding the really long flows (those that happen in more than 90% of the time windows). Suppose the trace consists of W consecutive time windows. Naively checking whether each of the $O(K^2)$ pairs of flows occur in each of the time-windows, takes $O(W * K^2)$ time. The square term, K^2, dominates the running time and can be quite long. Instead, we observe that in any time-window only a handful of the K flows are active. Thus, instead of counting frequencies of all K^2 pairs at each time-window, we need only count flow pairs that are active in a time window. If the wth window has S_w flows, our algorithm computes frequencies for all rules in $O(\sum_{w=1}^{W} {S_w}^2)$ time. Figure 4-3 shows the improvement in computational efficiency with this algorithm on our traces. For concreteness, we show the pseudo-code of our algorithm.

Algorithm 1 ExtractCommRules(Packet Trace)

```
 1: Find F – the set of top K active flows and generics              ▷ Selective Bias
 2: Compute Activity Matrix M for all flows in F                  ▷ As shown in Figure 4-2
 3: Use Rule Templates to Create Candidate Rule set R                        ▷ See §4.2.3
 4: for all Time Windows w ∈ rows (M) do
 5:   for all <flow, activity> tuples X, Y ∈ Window w do
 6:     if X ⇒ Y ∈ Candidate Rules R then
 7:       UpdateStats Freq(X ⇒ Y)                              ▷ Increment Counters
 8:     end if
 9:   end for
10: end for
11: for all X ⇒ Y rules in Candidate Rules R do                             ▷ Pruning
12:   if Score(X ⇒ Y) > α, Support(X ⇒ Y) > β then
13:     Output Rule X ⇒ Y as Significant
14:   end if
15: end for
```

4.3.1 Composing Communication Rules

The communication rules discussed above relate only a pair of flows. Our algorithm and statistical significance score naturally extend to rules involving more than two flows. To see this, note that if the left side of the rule $X \Rightarrow Y$ were to be replaced with a conjunction $X_1 \wedge X_2 \wedge \cdots \wedge X_N \Rightarrow Y$, both the algorithm and the significance score still apply.

The trade-off is computational complexity versus rule exhaustiveness. Looking for rules involving up to N flows would roughly take $O(W * K^N)$ time, since we would have to check each of the K^N flow groups generated from the top K flows on each of the W windows in the trace. On the other hand, doing so would help only when a set of flows are related but none of the pair-wise relationships are significant; for example, say F_3 is *present* only when both F_1 is *new* and F_2 is *present* and never otherwise. Here, $F_1.new \wedge F_2.present \Rightarrow F_3.present$ is a significant rule but neither of the pair-wise rules are significant enough. We believe in Occam's razor, the more complicated a rule, the less likely it would happen; hence we focus on pair-wise rules.

Complementary to rule exhaustiveness is the problem of grouping similar rules. Our analysis generates hundreds of statistically significant rules on a packet trace; hence it is worthwhile to attempt clustering similar rules into easier-to-understand groups. We found two techniques to be helpful. The first is rather simple. We transform the discovered rules into a rule-graph, wherein each rule is a directed edge joining the nodes corresponding to the <flow, activity> tuples that comprise the rule. We then take a transitive closure of the rule-graph to identify clusters of rules that involve related flows, for example, $X \Rightarrow Y_1$, $X \Rightarrow Y_2, Y_1 \Rightarrow Z$ will belong to the same transitive closure. The second technique is more complex and is motivated by our observation that the rule-graph consists of highly clustered components that are connected by "weak" edges, i.e., rules with low statistical significance. Hence, we apply spectral partitioning [171] to the rule-graph, to remove such weak edges between strongly connected components. Specifically, if A is the adjacency matrix of the

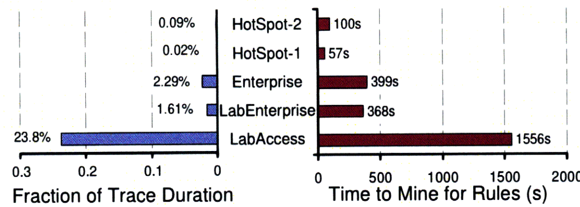


Figure 4-5 – eXpose mines for communication rules within a small fraction of the duration of the trace, so an online solution is feasible.

rule-graph, where each matrix entry is the significance score of the corresponding rule, we recursively partition the graph based on the second smallest eigen value of A's laplacian, as described below. Figure 4-4 illustrates the effect of spectral partitioning on rules from the

Algorithm 2 RecursiveSpectralPartition(RuleGraph A)

1: **if** $dim(A) < 10$ or A is $tightlyConnected$ **then** ▷ End of Recursion
2: return
3: **end if**
4: Laplacian $L = diag(colSums(A)) - A$
5: Find EigenVector ν for 2nd smallest eigen value of L
6: Split A's rows based on the sign of ν into A_{pos}, A_{neg}
7: Recursively SpectralPartition A_{pos} and A_{neg}

LabEnterprise trace. Graph nodes correspond to a flow or a generic in the trace, an edge between nodes indicates that eXpose discovered a rule linking the two nodes. The graph in the middle plots the output of eXpose's rule-mining algorithm–i.e., all the statistically significant rules, whereas the graph on the right plots the output of our recursive spectral partitioning algorithm. We found in practice that eliminating weak rules which strangle otherwise disconnected node groups reduces false positives.

4.3.2 Towards a Streaming Solution

As presented so far, eXpose runs offline on packet traces. But, can eXpose work in an online fashion, i.e., can we learn rules from packet streams? Clearly, the chief obstacle is whether rules can be mined quickly enough. Figure 4-5 shows how long eXpose takes to mine for rules in each of the five sample traces on a 1.8MHz Dual Core AMD Opteron with 16GB of RAM. Note that the time to mine rules is always much smaller than the duration of the trace and is often many orders of magnitude smaller.[5] Hence, an online implementation seems feasible. To flesh out a streaming version of eXpose, besides mining for rules, we need to compute rows of the activity matrix (§4.2.1) online, maintain counters for the currently active flows to determine their *new, present* status and frequency counters to identify the K highly active flows (§4.2.2); all of these seem solvable.

[5]eXpose takes longer to mine rules in the LabAccess trace because unlike the other traces, there is little locality in this trace with flows going off to many unique destinations in each time window.

Trace	Collected at...	Type	Length	Unique Flows	Size
LabAccess	Our Lab (CSAIL@MIT)'s access link to the Internet	Conn. Records	3 hrs	2,832,931	2 GB
LabEnterprise	Link facing internal Lab Servers	Conn. Records	7 hrs	909,563	2 GB
Enterprise	Link between Microsoft's Research and corporate LANs	Pkt Headers	3 hrs	622,959	30 GB
HotSpot-1	Access link for the SIGCOMM'04 wireless LAN	Pkt. Headers	3 days	204,318	604 MB
HotSpot-2	Access Link for the OSDI'06 wireless LAN	Pkt. Headers	3 days	603,127	2.1 GB

Table 4.2 – Dataset of traces we have tested eXpose with.

4.4 Evaluation

We deployed eXpose on the links carrying traffic towards internal servers at our lab and a major Enterprise. Further, we analyzed traffic traces from three additional locations–the access links at our lab, and at two wireless hot-spots (conference venues for SIGCOMM'04 and OSDI'06). Here, we describe the rules that we corroborated through discussions with admins at each of the locations and highlight sources for both false positives and false negatives.

4.4.1 Dataset

Our traces encompass packet headers (pcap format) and connection records (BRO [133] format) while some traces anonymize IPs. Since eXpose does not perform deep packet inspection, summary information showing the flows active in each time window suffices. Table 4.2 summarizes our traces. All our traces were collected at links carrying traffic for thousands of clients, but the connectivity patterns and traffic mix vary. Traffic on the access links is mostly web browsing and email access and exhibits a high degree of fan-out with connections going off to many unique IPs. Traffic on the enterprise links has greater diversity in applications but is directed towards fewer IPs. Traffic in the Enterprise trace is dominated by Windows/NT machines whereas the others show a wider mix of operating systems. eXpose uncovered many interesting flow rules in the traces. The differences in the rules from the various locations provide insights into the characteristics of each environment.

4.4.2 Metrics

One of the metrics we care about is breadth; we want to extract broad rules that represent patterns for a majority of the traffic. In each of our traces, we were able to discover dependencies for a significant proportion of user actions— web browsing, email, file-server access, instant messaging, peer-to-peer traffic and multimedia content. We care about both correctness and completeness of our rules. False negatives are patterns that are expected to exist in a trace but are not discovered by eXpose. We checked with administrators at both our lab and the corporate enterprise and report the patterns missed by eXpose. False positives are

Trace	# Flow Pairs	#Generics Added	#Rules Evaluated	#Rules Output	#Rule-Clusters
LabAccess	6.76×10^{12}	730	944,681	31,904	301
LabEnterprise	8.26×10^{11}	830	899,034	29,797	346
Enterprise	3.88×10^{11}	1,952	2,000,089	6,289	504
HotSpot-1	4.17×10^{10}	761	513,647	20,094	101
HotSpot-2	3.64×10^{11}	1,040	662,134	1,409	151

Table 4.3 – Progression from Packet Trace to Clusters of Communication Rules

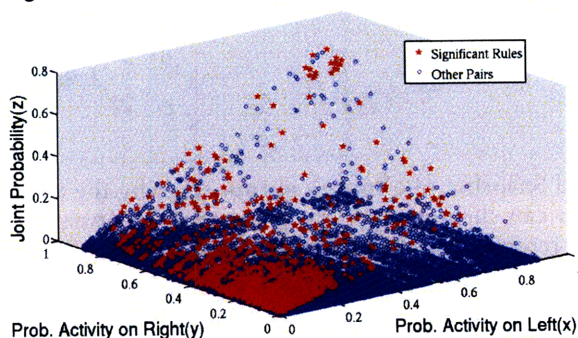


Figure 4-6 – Rules identified by eXpose from among the many possible rules. The figure has a circle (in blue) for each possible rule and a star (in red) for rules that are identified as significant. Significant rules come in different forms; simply looking for high joint probability (high z) is not enough.

flow-pairs that are scored highly by eXpose but have no reasonable explanation. We aggressively assume every rule we could not explain to be a false positive and mention the sources for such rules in our traces.

4.4.3 Nature of the Rule-Mining Problem

Identifying significant rules from among the many possibilities is tricky. For the LabEnterprise trace, Figure 4-6 plots in (blue) circles each of the potential rules and in (red) stars each of the rules eXpose identifies as significant. Clearly, significant rules come in different forms, some involve activities that happen rarely, both individually and together (near the (0,0,0) corner), others involve one rare activity and one frequent activity (near the (0,1,0) and (1,0,0) corners), and yet others involve a pair of frequent activities (close to the (1,1,1) corner). Simply looking for pairs with high joint probability (points with $z > const$) or looking for pairs with high conditional probability ($\frac{z}{x} > const$) does not suffice.

Before detailing the kinds of rules discovered by eXpose, we present the bigger picture. eXpose augments a packet trace with generics–abstract versions of the real flows, evaluates many potential flow/generic pairs to extract the significant rules and clusters together rules that are similar to one other. Table 4.3 shows for each of our traces the progression of eXpose through each of these phases.

In this context, it is easy to place our chief contribution—a technique to identify the few hundred significant patterns from among the $10^{10} - 10^{12}$ possibilities. To achieve this, eXpose selectively biases search by not evaluating rules that are unlikely to be useful (see Table 4.3, #Rules Evaluated vs. #Flow Pairs). Second, eXpose abstracts away extraneous flow details to make useful patterns more discernible (see Table 4.3, #Generics Added). Third, eXpose scores the candidate rules with an appropriate statistical measure and mines effi-

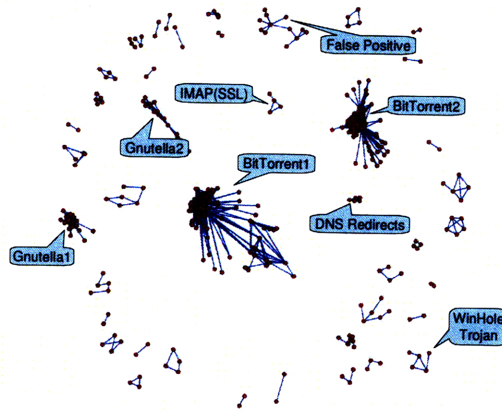


Figure 4-7 – Annotated Snapshot of **eXpose's output** showing the rules learnt by eXpose for the HotSpot-1 Trace. A user can click on a pattern to see more information about the corresponding rule. Nodes in the graph represent flow activities and edges representing rules join the two activities involved in each rule.

ciently (see Figure 4-5). Finally, eXpose aggregates the inferred rules into a small number of useful patterns that an admin can corroborate and use (see Table 4.3 #Rule Clusters vs. #Rules Output). Figure 4-7 shows an annotated snapshot of eXpose's output.

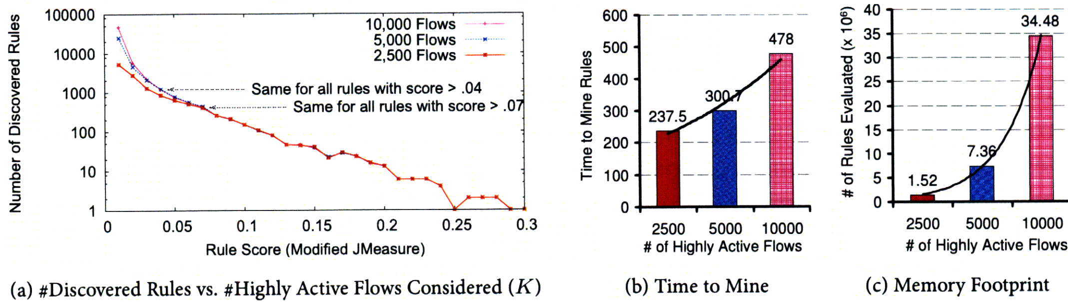
4.4.4 Evaluation in Controlled Settings

A key obstacle in evaluating a rule miner like eXpose is the lack of ground-truth information for traces in the wild. To circumvent this, we ran eXpose on a three hour trace from a single client desktop. We discovered all the expected communication rules including, the dependence with DNS, the accesses to the yp server during logins, and the rules for NFS access. Unexpectedly, we found dependencies for certain often browsed web-sites. eXpose found rules for how advertisements and images are synchronously fetched from other servers whenever the client browsed the main page of a web site. Further, we injected artificial traffic wherein pairs of flows at random would either be independent of each other or dependent. eXpose was successful at discovering rules for the dependent flows. We also crafted traffic that occurs together always but is separated by a time gap greater than eXpose's choice of time window size. As expected, eXpose did not discover these dependencies. Similar to prior work [54, 103], we share the belief that *dependent yet separated by long time gap* flow pairs are not common in practice and defer finding such pairs to future work.

4.4.5 Micro-Evaluation

We first evaluate some of eXpose's design choices.

Is Selective Biasing Useful? Recall that eXpose selectively biases the search to avoid pairs that are unlikely to be dependent. While the details of how eXpose picks rules to evaluate are elsewhere (§4.2.2), here we verify its usefulness. On the HotSpot-1 trace, eXpose finds 20,094 significant rules from among the 513,647 flow pairs that it evaluates for a *hit-rate* of 3.9×10^{-2}. Doing away with some of our biasing constraints (specifically the constraint that flow pairs have atleast one matching IP to be considered for a rule) caused eXpose to evaluate



(a) #Discovered Rules vs. #Highly Active Flows Considered (K) (b) Time to Mine (c) Memory Footprint

Figure 4-8 – Sensitivity of eXpose to the number of highly active flows (K, §4.2.2). Recall that eXpose mines over the top $K = 5000$ highly active flows by default. The more statistically significant a rule (higher score), the fewer the number of active flows eXpose has to consider to discover the rule!

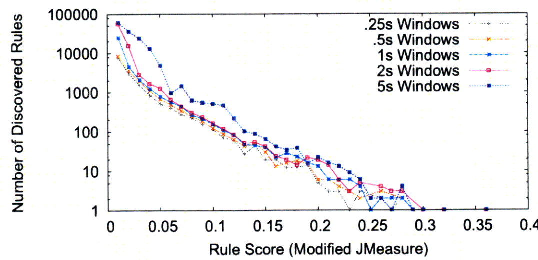


Figure 4-9 – Sensitivity of eXpose to the size of time windows used in discretizing the trace (§4.2.1). Window sizes in the [.25s, 2s] range seem to not change results appreciably, verifying the assumption that most dependent flows happen within a short duration of each other.

an order of magnitude more (5,409,049) flow pairs. Yet, only an additional 165 rules were found for a much smaller hit-rate of 3.3×10^{-5} and a large increase in the time to mine.

How Sensitive is eXpose to the Number of Active Flows? Recall that eXpose mines over the top $K = 5000$ highly active flows by default. But, do the discovered rules change appreciably if eXpose mines over many more or many fewer flows? Figure 4-8 plots the number of discovered rules and their scores when eXpose mines over the LabEnterprise trace with three different choices of K. As eXpose mines over more active flows, its search space broadens and it discovers more rules. But, the more statistically significant a rule (higher score), the fewer the number of active flows eXpose has to consider in order to reveal that rule! In Figure 4-8 (a), we see that when eXpose mines over twice the default number of highly active flows ($K = 10,000$), none of the new rules have score higher than .04. Figures 4-8(b, c) show the cost of expanding the search space. Since eXpose has to now evaluate many more rules, the time to mine for rules increases by 59% (from 300.7s to 478s) and the memory footprint increases by 368%. We believe that an administrator can choose the number of active flows to tradeoff higher resource cost for better fidelity in revealing the less significant rules. We picked $K = 5000$ as the default.

Why pick $1s$ wide time-windows? Figure 4-9 plots the number of rules discovered at each score level by eXpose on the LabEnterprise trace when discretizing at time windows of different sizes. We see that smaller windows lead to fewer rules while larger windows lead to more rules, at almost every score level. The reason is twofold. First, at larger time windows, eXpose would discover rules for more dependent flows, i.e., all those that are separated by no more than the time window. Second, since more flows are present in each of the larger time

windows on average, it is more likely that a flow pair will co-occur merely due to chance. Lacking ground truth it is hard to distinguish between these two cases—whether a rule that was newly discovered at a larger window size is a true rule that was missed at the smaller window size or is a false positive that was added merely by chance. Regardless, at all window sizes in the [.25s, 2s] range, eXpose reveals very similar rules, showing that dependencies that occur within a short period are highly stable.

4.4.6 Case Study: Patterns in the Enterprise

In every one of our traces, we found patterns of typical behavior describing the network setup and the dependencies between applications, and also patterns of atypical behavior indicating configuration problems or malicious activity. We have been running eXpose inside Microsoft on the link connecting a thousand node research LAN with the rest of the corporate network (Figure 4-1d) for a few months. Here, we present rules learnt by eXpose on a 3hr trace.

Load Balancing: Perhaps the most significant pattern in the Enterprise trace was due to the proxies used for external web access. There are about thirty different proxies, and eXpose found this rule cluster of generics:

$$\begin{aligned} Proxy1.80 : *.* &\Rightarrow Proxy2.80 : *.*;\ Proxy3.80 : *.*;\ \ldots \\ Proxy2.80 : *.* &\Rightarrow Proxy1.80 : *.*;\ Proxy3.80 : *.*;\ \ldots \\ \ldots &\ \ \text{and so on.} \end{aligned}$$

This rule-cluster was a clique; the generic corresponding to each proxy was linked by a rule to every one of the generics corresponding to the other proxies. These rules show that whenever a client talks to one of the proxy servers, the client is very likely to talk to some other proxy server. The explanation for this rule is simple – a client browsing the web fetches multiple HTTP objects in quick succession, and the load-balancing aspect of the proxy cluster spreads the client's HTTP requests across multiple proxy servers.

We note two things. First, this rule could not be discovered without abstraction. No one client talks to all the proxies, nor does any one client browse for very long. Second, without any knowledge of the network setup in the Enterprise, eXpose reveals the existence of load balancing web proxies and further shows that the load balancer works well. Such information helps in understanding the network setup and in troubleshooting performance problems.

Within the Enterprise, many services are load-balanced. We found similar load-balancing patterns for DNS servers on port 53, WINS (windows name lookup) servers on port 137 and domain-controller servers (certificates) on port 138.

Application Dependencies: We believe that one of the main contributions of eXpose is the ability to automatically find dependencies for all the important applications. eXpose learnt rules for web activity, email viewing, video lecture broadcasts, printers and dependencies at all the prominent servers.

eXpose discovered rules indicating that name lookup is a key ingredient of activities ranging from distributed file-systems to email and web access.

$$*.* : Server.port \Rightarrow *.* : DNS.53$$
$$*.* : Server.port \Rightarrow *.* : WINS.137$$

The first rule shows that a client accessing any server does a DNS lookup for names, while the latter shows the client doing a WINS (windows name server) lookup. Names in the Enterprise are distributed across DNS and WINS tables. Such rules were ubiquitous for every one of the servers mentioned in the rest of the trace, so we omit further mention.

Web: For web-browsing, we found that a majority of the proxies requested authentication credentials before access.

$$Proxy1.500 : *.* \Rightarrow Proxy1.80 : *.*$$

This generic rule shows that when fetching HTTP data (at port 80) from a proxy, clients exchange Kerberos credentials (at port 500).

E-Mail: eXpose found dependencies for e-mail access.

$$Client.* : Mail.135 \Rightarrow Client.* : DC.88$$
$$Client.* : Mail.135 \Rightarrow Client.* : Mail.X$$
$$Client.* : Mail.X \Rightarrow Client.* : PFS_1.X, Client.* : PFS_2.X$$
$$Client.* : Mail.X \Rightarrow Client.* : Proxy.80$$

The rules show that whenever a client talks to a mail server, he talks to a domain-controller (DC); the DC servers on port 88 issue Kerberos credentials after authenticating the user. The second rule shows that the actual mail server is on a "custom port", so the clients first look up the end-point mapper on port 135 at the server to learn which port the actual exchange server process is located at. Enterprise-wide public mail is stored elsewhere on *public folder servers*. Many clients browse through public mail simultaneously with their personal mail. Finally, we found that most mail contains HTTP content, or has links to content, so the users connect to the proxies when reading mail.

A couple of points are worth noting here. The e-mail dependencies are the first instance of multi-host dependencies and show the ease with which eXpose extends across hosts and applications. eXpose discovered these rules without being told that email was an important activity and without knowing which ports/servers are involved in email. Clients within the Enterprise are distributed across multiple mail servers, so we found multiple instances of the above rules, one for each mail server. Finally, eXpose found the port used by most of the exchange servers and our admin was particularly interested in the exceptions.

$$*.* : Mail_1.135 \Rightarrow *.* : Mail_1.49155$$

Most mail servers were running out of a default configuration file that created exchange server processes on port 49155. The few exceptions were probably due to legacy servers that had out-of-date configurations. Ability to detect such fine-grained configuration helps to understand and debug problems.

File-Servers: For Windows SMB (think NFS for Windows), eXpose discovers that

$$SMBServer.445 : *.* \Rightarrow SMBServer.139 : *.*.$$

The rule indicates that clients ask which server has the file by querying the server (at port

139) which in turn responds with a file-handle that the clients use to fetch the file (at port 445). Of course, the file-handle may point to another server.

$$OtherSMBServer.445 : *.* \Rightarrow SMBServer.139 : *.*$$

Such SMB redirection is common practice so that logical names share a hierarchy, while the files are distributed across servers. eXpose uncovered this tangle of re-directions.

Video Lecture Broadcasts: eXpose discovered multiple cliques corresponding to video lecture broadcasts.

$$\begin{aligned} Video.rtsp : Client_1.* &\Leftrightarrow Video.rtsp : Client_2.* \\ Video.rtsp : Client_1.* &\Leftrightarrow Video.rtsp : Client_3.* \\ Video.rtsp : Client_2.* &\Leftrightarrow Video.rtsp : Client_3.* \\ &\ldots \text{ and so on.} \end{aligned}$$

It turns out that the Video server live-streamed talks and other events within the enterprise over Real Time Streaming Protocol (rtsp). Each clique corresponded to the audience of a particular talk. Rules linking a pair of clients who were tuned in for a long overlapping period had higher scores than those involving clients who tuned out of the broadcast quickly.

Mailing List Servers: eXpose discovered multiple instances of cliques involving email servers.

$$\begin{aligned} ListServ.* : Client_1.* &\Leftrightarrow ListServ.* : Client_2.* \\ ListServ.* : Client_1.* &\Leftrightarrow ListServ.* : Client_3.* \\ ListServ.* : Client_2.* &\Leftrightarrow ListServ.* : Client_3.* \\ &\ldots \text{ and so on.} \end{aligned}$$

It turns out that each of these mail servers was responsible for a particular mailing list. Whenever a mail would be sent to the list, the mail server forwards the mail onto all the participants of that list. eXpose discovered when each list was active and the participants for each list.

DHCP: Clients that were searching for a DHCP server by broadcasting on the network caused the pattern,

$$NetworkBroadcast.137 : *.* \Rightarrow DHCPServer.137 : *.*.$$

Printers: eXpose found a clique involving print spoolers:

$$\begin{aligned} IP_1.161 : PrintServ.* &\Leftrightarrow IP_2.161 : PrintServ.* \\ IP_1.161 : PrintServ.* &\Leftrightarrow IP_3.161 : PrintServ.* \\ IP_2.161 : PrintServ.* &\Leftrightarrow IP_3.161 : PrintServ.* \\ &\ldots \text{ and so on.} \end{aligned}$$

It turns out that each of the IP's corresponded to the network interfaces of the printers throughout the Enterprise. The print-server appears to periodically poll the SNMP (161) port of these printers for usage and other information.

Workload Clusters: eXpose found cliques for file-servers.

$$\begin{aligned} Pool_1.* : FileServ.445 &\Leftrightarrow Pool_2.* : FileServ.445 \\ Pool_1.* : FileServ.445 &\Leftrightarrow Pool_3.* : FileServ.445 \\ Pool_2.* : FileServ.445 &\Leftrightarrow Pool_3.* : FileServ.445 \\ &\ldots \text{ and so on.} \end{aligned}$$

File-servers showing up in the above rules are centralized data stores for various groups. They were accessed by pools of clients who were crunching the data in parallel.

Presence Server: eXpose found many such rules,

$$Presence.5601 : Client_1.* \Leftrightarrow Presence.5601 : Client_2.*$$

It turns out that a presence server is part of Windows Office Communicator. Whenever a client logs in, logs out or goes idle, his machine sends an update to the presence server which forwards the information onto the user's friends. So, each rule above links users who are in each others *friend list.*

Oddities: eXpose found that a particular user Bob's machine does:

$$\begin{aligned} Bob.* : Home_1.* &\Leftrightarrow Bob.* : Home_2 \\ Bob.* : Home_1.* &\Leftrightarrow Bob.* : Home_3 \\ Bob.* : Home_2.* &\Leftrightarrow Bob.* : Home_3 \\ &\ldots \text{ and so on.} \end{aligned}$$

It turns out that Bob's machine was talking to many IPs belonging to DSL users and cable-modem pools (Verizon DSL Customers, Road Runner Customers). eXpose found this pattern because the accesses to these home machines were periodic and synchronized. Either Bob is running an experiment that probes many home machines, or he is part of a zombie bot-net and the communication here is keep-alive messages between bots in the botnet.

4.4.7 Case: Patterns in the LabEnterprise

We have deployed eXpose within the CSAIL lab at MIT such that eXpose sees traffic to and from the major internal servers, including web, email, Kerberos, AFS, NFS servers, managed user work-stations and a Debian/Fedora mirror (see Figure 4-1c). Here we report rules learnt by eXpose on a 400 minute trace.

syslogd clique: All the managed workstations within the lab and most of the internal servers ran the same version of Debian with similar configuration. In particular, these machines export their system log files to a server via port 514. Cron jobs in the config ran at specific times causing synchronized updates to the individual system logs and synchronized updates to the syslogd server.

$$\begin{aligned} IP_1.514 : Syslogd.514 &\Leftrightarrow IP_2.514 : Syslogd.514 \\ IP_1.514 : Syslogd.514 &\Leftrightarrow IP_3.514 : Syslogd.514 \\ IP_2.514 : Syslogd.514 &\Leftrightarrow IP_3.514 : Syslogd.514 \\ &\ldots \text{ and so on.} \end{aligned}$$

As eXpose could list all the machines uploading logs, the admin could chase down machines that were supposed to be managed but were not uploading syslog files synchronously.

AFS accesses: The following pattern repeated across many clients:

$$\begin{aligned} Client.7001 : AFSRoot.7003 &\Rightarrow Client.7001 : * \\ Client.7001 : AFS_1.7000 &\Rightarrow Client.7001 : AFS_2.7000 \\ Client.7001 : AFS_1.7000 &\Rightarrow AFS_1.7000 : AFSRoot.7002 \end{aligned}$$

These rules show that a client talks to the root server (at port 7003) to find which of the many volume servers have the files he needs and then talks to the appropriate volume server. The lab has a single root server but many tens of volume servers with files distributed across the volume servers. A user's content is often spread across more than one server, causing simultaneous accesses from his cache manager (at port 7001) to the AFS servers (at port 7000). Finally, creating new files, or browsing into different parts of the AFS tree initiate connections to a permissions server (at port 7002).

Besides identifying the underlying structure of AFS traffic, eXpose's rules let us understand where each user's content was located and how the accesses were spread across the various servers. The lab admins were happy to see that the accesses to volume servers matched up with the hard-disk sizes on the servers (larger servers got more accesses).

File-Server as the Backend: The fact that many users have all their data on AFS leads to many neat patterns:

$$\begin{aligned} Remote_1 : WebServ.80 &\Rightarrow WebServ.7000 : AFS.7001 \\ Client : LoginServ.22 &\Rightarrow LoginServ.7000 : AFS.7001 \\ Compute_1 : AFS.7001 &\Leftrightarrow Compute_2 : AFS.7001 \\ Compute_1 : AFS.7001 &\Leftrightarrow Compute_3 : AFS.7001 \\ Compute_2 : AFS.7001 &\Leftrightarrow Compute_3 : AFS.7001 \end{aligned}$$

The first shows that accesses to home pages on the lab's web-server cause the web-server to fetch content from the AFS servers. While the home-pages are stored on the web-server, it is likely that users link to content (papers, cgi-scripts) present elsewhere in their AFS share. Also, ssh connections into the lab's login server caused the login server to mount directories from AFS. Finally, compute clusters synchronously accessed many volume servers, most likely because a data-intensive job was parallelized across the cluster.

E-Mail Flow: eXpose discovered how email flows through the lab.

$$\begin{aligned} Incoming.25 : *.* &\Rightarrow Webmail.2003 : Incoming.X \\ *.* : Webmail.143 &\Rightarrow *.* : Webmail.993 \\ *.* : Webmail.110 &\Rightarrow *.* : Webmail.995 \end{aligned}$$

The first rule shows that whenever a connection is made on port 25 (SMTP) to a mail server in the lab, the mail server connects with a different server. It turns out, the lab has one Incoming SMTP server that receives mail from outside but does not store the mail. Instead, mail is forwarded on to a webmail server via LMTP (enterprise version of SMTP, port 2003). The webmail server in turn provides an interface for users to read their mail. The next two rules show the webmail server responding to both IMAP (port 143) and POP (port 110) connections only to force clients to use the corresponding secure versions IMAPS (port 993) and POPS (port 995).

Outgoing E-mail: eXpose tracks patterns in outgoing mail.

$$OutMail.* : dns0.mtu.ru.53 \Leftrightarrow OutMail.* : dns1.mtu.ru.53 \text{ (many)}$$
$$OutMail.* : Mail_1.25 \Leftrightarrow OutMail.* : Mail_2.25 \text{ (clique)}$$

The first rule above shows that whenever the outgoing server does a DNS (port 53) lookup at one remote name server, it does a name lookup on another redundant name server in the same domain. It turns out that the lab's mail server implementation simultaneously looks up multiple DNS servers in the hope that at least one of them would respond. Further, the second rule shows the outgoing mail server simultaneously connecting to multiple mail servers in domains like *messagelabs.com*. Apparently, messagelabs (and many others) are out-sourced email providers who receive email for companies(e.g., Akamai). Several lab mailing lists have employees of these companies, and hence a mail to such lists makes the outgoing server deliver mail simultaneously to multiple SMTP servers at the outsourced email provider.

Nagios Monitor: The Lab admins use a Nagios machine to monitor the health of the lab's key servers.

$$Nagios.7001 : AFS_1.7000 \Rightarrow Nagios.7001 : AFS_2.7000 \text{ (AFS clique)}$$
$$Nagios.* : Mail_1.25 \Rightarrow Nagios.* : Mail_2.25 \text{ (mail clique)}$$
$$Nagios.* : AD.139 \Rightarrow Nagios.* : AD.389 \text{ (active directory)}$$

These rules show that the Nagios machine periodically connects to the servers of each type as a *client* and probes their health. The first rule is part of a full clique, the Nagios machine connected as an AFS client with every one of the AFS servers in the lab. Similarly, the second rule is part of a clique wherein the Nagios box sent mail out through each of the SMTP servers in the lab. The third rule shows Nagios checking the windows server for both its name (netbios, port 139) and LDAP services (directory lookup, port 389).

Discovered Bugs: eXpose discovered many instances of configuration problems and malicious users that the administrators were able to act upon. Here are two such examples:

(1) IDENT Dependency: eXpose found many instances when a client's connection to a server is followed by the server initiating a connection at port 113 on the client. For example,

$$Client.* : MailServer.25 \Leftrightarrow Client.113 : MailServer.*$$

It turns out that port 113 is IDENT traffic.

> Despite the fact that IDENT was never very useful, even today ...UNIX servers, most commonly IRC Chat, but some eMail servers as well, still have this IDENT protocol built into them. Any time someone attempts to establish a connection with them, that connection attempt is completely put on hold while the remote server attempts to use IDENT to connect back to the user's port 113 for identification [21].

Shown this rule, an admin changed configuration to disable IDENT.

(2) Legacy Configuration: A legacy DHCP server was active and responding to requests on a subnet. Even more, the legacy server was accessing the lab's authoritative name server that was supposed to be used only by the front-end DNS servers, which pull the master name table and respond to clients. Shown this rule, an admin disabled the legacy server.

$$NetworkBroadcast.68 : Legacy.67 \Rightarrow Legacy.* : MasterNS.53.$$

False Positives: We were unable to corroborate 45 out of the 346 rule groups that eXpose discovered on the labEnterprise trace, for a false positive rate of 13%. Some of the false positives were due to dependencies that appeared true but we could not explain such as pairs of flows that always happened together in 5% of time windows. Others were due to high volume servers, such as a debian mirror hosted within the lab. There are so many connections to the debian mirror server from so many different IPs that invariably we find many connections overlapping with each other. We do know that many operating systems have software that automatically touches the distribution mirrors at fixed times of day to check for updates, yet it is hard to say for sure why a pair of IPs access the debian mirror synchronously. A natural improvement to eXpose is to automatically scale the score threshold per server, i.e., high volume servers that are at a risk of false positives are reported only if they are in rules that have much higher scores.
False Negatives: Backend dependencies at the web-server hosting personal web-pages were too fine-grained for eXpose. We discovered the bulk dependency, i.e., that most web-page requests cause the web-server to fetch content from AFS. But, we were unable to isolate the more complex dependencies of individual web-pages carrying dynamic content. We believe that this is because of too few connections to such dynamic content and readily admit that eXpose is likely to miss specific dependencies while looking for the broad ones. But, one can *white-list* servers that eXpose should pay closer attention to by adding that server's flows to the list of flows that eXpose builds rules for.

4.4.8 Case: Patterns on the Lab's Access Link

mySQL worm: We found an instance of the mySQL worm; the host pD9E9D641.dip.t-dialin.net performed a port scan throughout the network on port 3306 which is the default mySQL port on Unix.

> The mySQL Bot scans this port looking for mySQL servers with weak passwords and if it is successful in logging in as root ... uses an exploit to install the bot on the system. [25].

Unlike other worms such as the SQL Sapphire Worm [36], this worm causes little traffic and may not show up in tools that detect heavy-hitters. Yet, eXpose detects it because the remote host scanned many lab hosts simultaneously.

Cloudmark: The mail server of a group in the lab was involved in an interesting pattern. Whenever the server received mail (at port 25), the server would connect to one of a bunch of servers owned by cloudmark.com at port 2703:

$$MailServer.25 : *.* \Rightarrow MailServer.* : CloudMark.2703.$$

Apparently, cloudmark is a filtering service for spam, phishing and virus-bearing mails to which this group subscribes.

NTP synchs: The lab runs a major Network Time Protocol (NTP) server. To keep system clocks up-to-date, clients periodically probe an NTP server, and adjust clocks based on the server's response and the round trip time estimate. Client implementations vary widely though. Most clients query the server infrequently and the probe/response pairs are just one UDP packet. But eXpose found

$$Client_1 : NTP.123 \Leftrightarrow Client_2 : NTP.123,$$

indicating that pairs of clients were accessing the server synchronously and often (one probe in every couple of seconds). Most likely, these machines have the same poorly written NTP client.

TOR: The lab contributes servers to the TOR [72] anonymity network. The ability to identify temporally correlated pairs strips one-hop anonymity, we can identify the next-hop for flows routed through the lab's TOR server. For example, eXpose finds rules,

$$IP_1.9001 : TOR.* \Leftrightarrow TOR.* : IP_2.9001$$

showing that traffic flows from IP_1 to IP_2 or vice versa. TOR's anonymity is not broken though. eXpose-like traffic analysis has to be done at every TOR server on a flow's path to identify the participants. But, this does highlight a weakness–TOR seems to not use cover traffic and since there isn't a lot of traffic to begin with, it is easy to correlate flows across one hop.

FTP Session Dependencies: The lab provides a mirror for both debian and fedora linux distributions. Clients around the world download OS images and packages. eXpose found that

$$IP_1.* : Mirror.* \Rightarrow Mirror.21 : IP_1.*\,.$$

It turns out that an ftp control connection (on port 21) to exchange commands precedes ftp data connections that do the actual data transfer. Further, data connections are started either actively, i.e., started by the server on port 20, or passively, i.e., started by clients at ephemeral ports. The rules show that most data connections, in practice, are passive perhaps to let clients behind NATs access data.

Discovered Bugs: Again eXpose discovered exploits and configuration problems. **(1) Legacy Addresses in Mailing Lists:** We found that our university's outgoing mail server was simultaneously accessing a couple of mail servers in the lab.

$$UnivMail.* : OldMail_1.25 \Leftrightarrow UnivMail.* : OldMail_2.25$$

This was consistent with other mailing list patterns we had seen, the university server was delivering mail to a list that had users at those machines in the lab. Unfortunately, these older mail servers were no longer operational and the email addresses had been invalid for a long time. When shown this rule, the university's admin responded that the mailing lists would be cleaned.

(2) Selective SSH Scans: eXpose identified several hosts in South Asia that were selectively scanning the lab's main routers.

$$*.* : Router_1.22 \Leftrightarrow *.* : Router_2 : 22 \text{ (three router clique).}$$

During the scan a host would simultaneously initiate ssh connections and try login/password pairs on all the main routers. eXpose also found co-operative scanning, wherein multiple hosts would scan a router at the same time.

$$Attack_1.* : Router.22 \Leftrightarrow Attack_2.* : Router.22$$

Given the patterns, the lab's admin blacklisted the scanning IPs.
(3) Web Robots: eXpose discovered rules for web crawlers.

$$Robot.* : Web_1.80 \Leftrightarrow Robot.* : Web_2.80$$

These rules indicate that a crawler bounces between multiple web-servers perhaps as it follows links on the lab's web content. Most of the robots belonged to well-known search sites, but one of them was a machine in south-east asia that had neither a name record nor was pingable after the fact. These robots neither make too many connections nor pull down a lot of data and hence are indistinguishable from normal web traffic. eXpose found them by their characteristic access pattern–synchronous accesses to multiple web-servers while chasing down links. The lab's admin flagged the *unknown IP* to be inspected more carefully by the intrusion detection box.
False-Positives: Our lab hosts Planetlab machines. Some dependencies, such as access to the CODEEN CDN are discernible. Yet, it is almost impossible to figure out from the packet trace which currently active slice caused which packets. So, we did not attempt to corroborate rules involving Planetlab.

4.4.9 Case Study: Rules for HotSpot Traces

Note that both the Sigcomm'04 and OSDI'06 traces are anonymized, so we corroborated rules based on the port and protocol numbers of flows. Figure 4-7 graphically shows all the patterns that eXpose learned from the Sigcomm'04 trace.
Peer-to-peer Traffic: Most of the high-density clusters were due to peer-to-peer traffic. In Figure 4-7, the two large clusters were due to two wireless hosts using BitTorrent, whereas the two smaller clusters were due to Gnutella users. Each of these hosts connected to many tens of unique peers. eXpose found that whenever a peer communicates on one of the ports in the 6881-6890 range, the peer is likely to communicate on another port in the same range. Presumably, this is due to multiple flows between the host and BitTorrent peer. Gnutella's clusters are similar, except in a different and smaller port range; most flows here are on ports 6346-6349.
Suspicious activity on port 1081: One of the wireless hosts communicated synchronously with four different machines on port 1081. Popular wisdom [32] says that the port is used by the WinHole–"A trojanized version of Wingate proxy server". The traffic volume of each of these flows is fairly small, yet eXpose discovered the pattern from what appear to be synchronous heart-beat messages between this victim and other machines:

$$Victim.* : Other_1.1081 \Leftrightarrow Victim.* : Other_2.1081$$

DHCP Servers: Almost all the clients in the OSDI trace were involved in a simple pattern; the client sends out a broadcast message to port 68 and gets a response from either IP_1 or

IP_2.

$$*.67 : 255.255.255.255.68 \Leftrightarrow IP_1.68 : *.67$$

It appears that both IP_1 and IP_2 carry a DHCP Server at the well known port 68. The DHCP daemon responds to requests for new IP addresses sent by DHCP clients from port 67. DHCP traffic is quite infrequent and involves few bytes, yet the synchronous accesses in time lead to this rule. Further, note that eXpose discovered the rule with no knowledge of what to expect in the trace.

Apple iTunes: eXpose found hosts talking on port 5353.

$$H_1.5353 : H_2.5353 \Leftrightarrow H_1.5353 : H_3.5353; H_1.5353 : H_4.5353$$

It turns out that the Apple iTunes application advertises to other Apple machines on the subnet if configured to share its music. Some users appear to have forgotten to disable this feature causing their laptops to advertise music at the conference. eXpose discovers this rule by the temporally correlated advertisements and had no knowledge of iTunes before hand.

Link-level Multicast Name Resolution: We found what appears to be a new form of looking up names. We saw earlier that windows hosts query both the local DNS server and the local WINS server to lookup names. In addition, eXpose observed these rules:

$$\begin{aligned} Host.* : Multicast.5355 &\Leftrightarrow Host.* : DNS.53 \\ Host.137 : WINS.137 &\Leftrightarrow Host.* : DNS.53 \\ Host.137 : WINS.137 &\Leftrightarrow Host.* : Multicast.5355. \end{aligned}$$

A few hosts were sending out packets to a multicast address on port 5355 along with the other name lookups. It turns out that this is link-level multicast based name resolution—a new feature in Vista [23] that is designed specifically for ad-hoc networks. Of course, this is the first time we ever heard of this protocol.

Day-Long Traceroutes: We found two hosts sending what appear to be day-long traceroutes. eXpose discovered these rules.

$$\begin{aligned} Host.0 : IP_1.0 : 1 &\Leftrightarrow Host.0 : IP_2.0 : 1;\ Host.0 : IP_3.0 : 1 \ldots \\ Host.0 : IP_2.0 : 1 &\Leftrightarrow Host.0 : IP_3.0 : 1;\ Host.0 : IP_4.0 : 1 \ldots \end{aligned}$$

The rules show that $Host$ was receiving ICMP (protocol 1) messages from a bunch of IPs all within the same one-second period and repeated many times throughout the day. Our best guess is that somebody in the conference was doing some measurements, maybe to check if a certain path changes during the day. eXpose found this low-volume event and also the fact that the path did not change.

IM: eXpose found these rules:

$$Host.* : MSNServ.1863 \Leftrightarrow Host.* : AOLServ.5190.$$

It turns out that ports 1863 and 5190 are well known ports for MSN and AOL Instant Messaging Servers respectively. It appears as if a couple of users were using aggregated instant messaging (IM) clients like GAIM [19] that can connect to multiple IM networks. There is little traffic in these flows, the IM client appears to refresh the servers periodically and synchronously leading to the rule.

4.5 Discussion

Accuracy Concerns and Human-in-the-loop: Perhaps the greatest limitation of eXpose is the need for a human in the loop who looks at the discovered clusters and explains them, i.e., tags them as either normal behavior, misconfigurations or attacks. Not only does this require knowledgabe administrators, but we also spent a fair amount of time investigating anomalous rules. On the flip side, these investigations revealed mis-configurations and were quite useful, but how can we improve the process? From applying eXpose on a fair number of traces, we noticed much commonality in discovered rules. We saw eXpose extract similar rules from different locations (e.g., the IMAP-IMAPS, POP-POPS) rules. Even more, a vast majority of the rules extracted on different days but at the same location are similar. This suggests that we should build a database of known rules at each location and share information across locations. Rules extracted on each new packet trace, perhaps in a streaming fashion, can be matched with known rules in the database. This lets the administrators focus on the *novel* patterns and gain further confidence in patterns that are seen repeatedly.

How much data does eXpose need? All results in this chapter were computed on traces that were either no longer than 3hours or not larger than 2GB. It is fair to wonder how much data a technique such as eXpose requires. The answer depends on the location at which the trace is collected, short durations suffice on high-traffic links that see traffic from many different clients to many different servers. Other locations may need longer traces. Note however, that running eXpose on two traces T_1, T_2 takes about as much time as running eXpose on the combined trace $T_1 \cdots T_2$, hence the admin gets to choose whether to run on larger traces, or merge the rules discovered on smaller traces (with appropriate post-processing).

Tracking changing dependencies: A couple of thoughts on tracking changes in dependencies. A simple solution would be to run eXpose on fixed-width traces, say every 3hours, and time-weight the discovered rules, i.e., rules discovered recently have more weight than rules in the past.

Operational Use of eXpose: We believe that eXpose's primary deployment would be such that it notices all traffic on the ingress link, or all traffic in and out of servers, by either directly plugging into the right switch, or by setting up appropriate VLANs. This practice would work in most educational institutions and wireless access spots, but doesn't quite scale to enterprises with thousands of servers–there is no one switch to plug eXpose into, nor is setting up such a large VLAN feasible. For such scenarios, we note that eXpose could be deployed at *network bottlenecks*, for example the few routers that connect a building, or a regional office with the rest of the enterprise. Rule mining with a farm of machines, when no one machine sees all the traffic, is an interesting open problem.

4.6 Related Work

A few tools aggregate traffic volumes and visualize the resulting aggregates. FlowScan [136] takes as input NetFlow data and breaks down the traffic volume according to the application (e.g., HTTP, FTP), the IP prefix, or the AS identifier. CoralReef [1] and IPMON [3] produce similar traffic breakdowns based on packet traces. Autofocus [79] also breaks the traffic

volumes into different categories but adapts the breakdown boundaries to zoom-in or out on interesting subnets and port ranges. eXpose extends these tools along a new dimension by identifying temporally correlated clusters of flows.

Other related work attempts to find traffic that matches a pre-defined communication pattern. Venkataraman et.al. [172] and Staniford et.al. [164] present streaming algorithms to identify SuperSpreaders, i.e., machines infected by a worm or virus that in turn infect many other hosts. Another line of work [185, 186] detects *stepping stones* whereby an attacker compromises a machine and uses it to launch a very different attack on other machines. Blinc [104] uses hints from multiple levels to tag each flow with the application that created it. More generally, intrusion detection systems like Bro [133] use a database of signatures for malicious activity and find traffic matching these signatures. Rather than identifying traffic that matches a given pattern, eXpose automatically extracts the underlying patterns in a trace.

Perhaps the closest to eXpose is work that finds detailed dependencies for individual applications. Kannan et. al. [103] analyze network traces to identify the structure of a particular application session (e.g., FTP, HTTP). For individual applications between one source and one destination, they build state machines detailing how the session progresses. When pointed at a server, Sherlock [54] finds dependencies for clients accessing that server even when such dependencies involve multiple other servers or protocols.

Fundamentally, eXpose is different as it finds dependencies without guidance. Without pre-focusing on any given application or server, eXpose looks for all statistically significant clusters of flows. This lets eXpose find patterns that spread across multiple hosts, protocols and applications, and even those that an admin did not know or expect such as configuration errors. Both Sherlock [54] and Kannan et. al. [103] can bring out detailed dependencies that eXpose's broader search might not highlight. But, to obtain output similar to eXpose, one would have to repeatedly apply these techniques to learn dependencies for one server or one application at a time. This is unlikely to scale and also misses out on dependencies at servers that the admin may forget to point towards.

4.7 Conclusion

We advance the state-of-the-art in traffic analysis by presenting a general mechanism to identify temporally correlated flows in a packet trace. While just looking at temporally correlated flows is unlikely to capture the myriad kinds of structures in a packet trace, we show that this is a powerful primitive that is able to capture many useful patterns. Our tool eXpose uniquely defines the concept of generic rules, focuses only on the statistically significant flow pairs and presents an algorithm that scales to large traces. Results from deploying eXpose within MIT and Microsoft Research show that eXpose uncovers many configuration errors and lets operators get a quick read of what is going on in their network without having to understand logs from the various servers.

Chapter 5

Localizing Faults and Performance Problems

In this chapter, we shift focus to using the inferred dependencies to diagnose performance problems. Using a network-based service can be a frustrating experience, marked by appearances of familiar hourglass or beach-ball icons, with little reliable indication of where the problem lies, and even less on how it might be mitigated. User-perceptible service degradations are common even inside the network of a single enterprise where traffic does not need to cross the public Internet. Consider Figure 5-1, which shows the distribution of time required for clients to fetch the home page from a major web-server in a large enterprise network including tens of thousands of network elements and over 400,000 hosts. The distribution comes from a data set of 18 thousand samples from 23 instrumented clients over a period of 24 days. The second mode of the distribution represents user-perceptible lags of 3 to 10+ seconds, and 13% of the requests experience this unacceptable performance. This problem persists because current network and service monitoring tools are blind to the complex set of dependencies across systems and networks in the enterprise, needed for root cause analysis.

5.1 Overview

Well-known management systems focus on individual components. They either monitor individual routers [184], individual servers [5] or deep-parse flows in network traces [28] and flood the administrator with alerts when anything goes wrong with any one of these components. For example, such tools raise an alert whenever a router drops a few packets, a server's memory utilization going past 90%, or when periodically generated traffic statistics cross a threshold. Yet, when an end-user experiences poor performance, these tools are unable to identify which of the involved components is responsible for the problem. In contrast, this chapter advocates an end-to-end approach. Rather than focusing on individual components, we build an end-to-end model to explain the reasons behind user's performance problems.

Such an end-to-end approach faces some challenges. First, both performance and hard faults can stem from problems anywhere in the IT infrastructure. Even simple requests like fetching a web-page involve multiple services: DNS servers, authentication servers, web-servers, and the backend SQL databases that hold the web-page data. Problems at any of these services, or in the routers and links along the network paths to these services, can affect

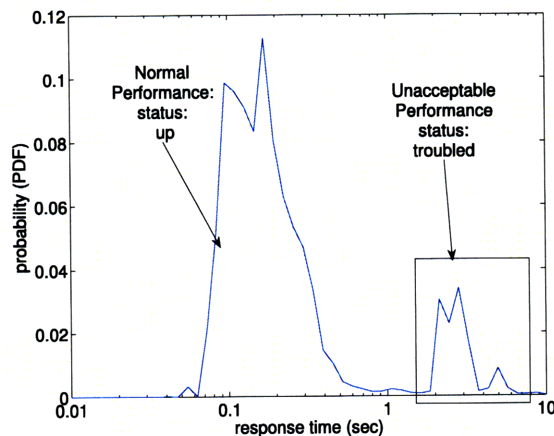


Figure 5-1 – The response time for clients within a large enterprise network when fetching the home page of a major web-server also within the enterprise. The times are clearly bi-modal, with 13% of the requests taking 10x longer than normal and resulting in user perceptible lags. We define the first mode in response time as indicating the service is *up* and the second mode as indicating the service is *troubled*.

the success or failure of the request. Yet, these dependencies among components in IT systems are rarely recorded, and worse, the dependencies evolve as systems grow and new applications get deployed. As a result, we must be able to discover the set of components involved in the processing of user requests. Second, even when the dependencies are known, we would still have to figure out which of the dependent components are likely responsible for a performance problem. For example, if a router on the path dropped a few packets and the server's cpu is at 70%, which of these components is responsible for a user's delayed response?

We are not the first to advocate discovering dependencies. Infact several startups, notably Relicore [33], Collation [11] and nLayers [29] perform "application dependency mapping" [17]. Not much is publicly known about their specific approaches or their applicability. Some of these tools tend to work only with a few applications, or applications that have pre-defined fingerpints; others are geared towards figuring out which applications exist [18] and still others require customization to mine appropriate Windows Registry entries and app-specific config files [12]. Regardless, our contribution here goes beyond dependency discovery. To the best of our knowledge, we are the first to couple dependency discovery with a probabilistic model that captures real-world dependencies and a fault localization technique that uses this model to find real problems in working networks.

Recall that the first challenge, that of discovering dependencies, is a particular use case of eXpose (Chapter 4). While eXpose finds statistically significant rules, i.e., rules that have a high *score*, building the probabilistic model in this chapter requires conditional probabilities. We note, however, that eXpose already computes conditional probabilities during its rule mining. Hence, this chapter focuses on the latter challenge, that of using the inferred dependencies to identify performance problems.

Our key contributions include a probabilistic model that captures the relationship between a end user's action, such as fetching a web page, and the set of components that this action depends on. Failover and load-balancing techniques commonly used in enterprise networks make determining the troublesome component even harder, since the set of components involved may change from request to request. We will show how to accommodate

such complicated dependencies. We also present a fault localization algorithm that computes the list of likely suspects given observations from multiple vantage points. Finally, given the large size of enterprise networks, we will show how to meet the above challenges in a manner that remains tractable even with hundreds of thousands of components.

We present our solutions as implemented in the Sherlock system. Sherlock aims to give IT administrators the tools they need to localize performance problems and hard failures that affect an end-user. Sherlock (1) detects the existence of faults and performance problems by monitoring the response time of services; (2) learns dependencies to identify the components that could be responsible; and (3) localizes the problem to the most likely component.

Sherlock meets these goals in the following ways: First, software agents running on each host analyze the packets that the host sends and receives to determine the set of services on which the host depends. Sherlock automatically assembles an *Inference Graph* that captures the dependencies between all components of the IT infrastructure by combining together these individual views of dependency. [1] Sherlock then augments the Inference Graph with information about the routers and links used to carry packets between hosts, and so encodes in a single model all the components that could affect a request. Second, our Inference Graph model includes primitives that capture the behavior of load-balancers and failover mechanisms. Operators identify where these mechanisms are used manually or via heuristics, but localization is then automatic. Third, we develop an algorithm that efficiently localizes faults in enterprise-scale networks using the Inference Graph and the measurements of service response times made by agents.

Our Fault Model departs from tradition. Conventional management systems treat each service, which we define as an *(IPaddr, port)* pair, as being either up or down. Such a simplistic model hides the kinds of *performance failures* shown in Figure 5-1. Hence, we model service availability as a 3-state value: a service is *up* when its response time is normal; it is *down* when requests result in either an error status code or no response at all; and it is *troubled* when responses fall significantly outside of normal response times. Our definition of troubled status includes the particularly challenging cases where only a subset of service requests are performing poorly.

We deliberately targeted Sherlock at localizing significant problems that affect the users of the IT infrastructure, hence our focus on performance as well as hard faults and our use of response time as an indicator for performance faults. Current systems overwhelm operators with meaningless alerts (the current management system in our organization generates 15,000 alerts a day, and they are almost universally ignored as so few prove significant). In contrast, Sherlock does not report problems that do not directly affect users. For example, Sherlock will not even detect that a server has a high CPU utilization unless requests are delayed as a result.

To the best of our knowledge, Sherlock is the first system that localizes performance failures across network and services in a timely manner without requiring modifications to existing applications and network components. The contributions of this chapter include our formulation of the Inference Graph and our algorithms for computing it for an entire IT infrastructure based on observations of the packets that hosts send and receive. Unlike

[1] We note that the algorithm to learn service dependencies is similar to that in Chapter 4 and refer the reader to [54] for further details.

previous work, our Inference Graph is both multi-level (in order to represent the multiple levels of dependencies found in IT infrastructure) and 3-state (so we can determine whether components are up, down, or experiencing a performance fault and troubled). This chapter also contributes extensions to prior work that optimize fault localization and adapt it for our three-state and multi-level Inference Graph. We document Sherlock's ability to localize performance problems by describing results from deploying Sherlock in both a testbed and a large and complex enterprise network.

5.2 Related Work

Today's enterprises use sophisticated commercial tools, such as EMC's SMARTS [184], HP Openview [30], IBM Tivoli [38], or Microsoft Operations Manager [5]. In practice, these systems do not help find the causes of performance problems as they treat servers and routers as independent boxes and produce a stream of SNMP counters, syslog messages, and alerts for each box. Fundamentally, box-centric methods are poor predictors of the end-to-end performance that users ultimately care about. For example, it's not clear what CPU load on a server means that users will be unhappy and so it is hard to set thresholds such that the box-centric technique raises alerts only when users are impacted. Many of these management companies recently acquired "application mapping" startups (namely nLayers, Relicore and Corollant) to advance beyond from box-centric approaches, but its not evident how well they have been able to tie application mapping and fault localization. In a specific example, over a 10-day period, the management tool used by Microsoft IT generated two thousand alerts for 160 servers that *might* be sick. Another 18K alerts were divided among 194 different alert types coming from 877 different servers, each of which could *potentially* affect user performance (e.g., 6 alerts for a server CPU utilization over 90%; 8 for low memory causing a service to stop). Investigating so many alerts is simply impractical, especially when a large fraction have no observable effect on users. Sherlock complements existing tools by detecting and localizing the problems that affect users.

Significant recent research focuses on detailed debugging of service problems in distributed systems. Many of these systems also extract the dependencies between components, but are different in character from Sherlock. Magpie [57], FUSE [75] and Pinpoint [70], instrument middleware on every host to track requests as they flow through the system. They then diagnose faults by correlating components with failed requests. Project5 [44] and WAP5 [143] record packet traces at each host and use message correlation algorithms to resolve which incoming packet triggered which outgoing packet. These projects target the debugging and profiling of *individual* applications, so determining exactly which message is caused by another message is critically important. In contrast, Sherlock combines measurements of the *many* applications running on an IT infrastructure to localize problems.

There is a large body of prior work tackling fault localization at the network layer, especially for large ISPs. In particular, BADABING [160] and Tulip [120] measure per-path characteristics, such as loss rate and latency, to identify problems that impact user-perceptible performance. These methods (and many commercial products as well) use active probing to pinpoint faulty IP links. Sherlock instead uses a passive correlation approach to localize failed network components.

Machine learning methods have been widely discussed for fault management. Pearl [135] describes a graph model for Bayesian networks. Sherlock uses similar graph models to build Inference Graphs. Rish et. al. [145] combines active probing and dependency graph modeling for fault diagnosis in a network of routers and end hosts, but they do not describe how the graph model might be automatically constructed. Unlike Sherlock, their method does not model failover servers or load balancers, which are common in enterprise networks. Shrink [101] and SCORE [114] make seminal contributions in modeling the network as a two-level graph and using it to find the most likely root causes of faults in wide-area networks. In SCORE, dependencies are encoded as a set and fault-localization becomes minimal set cover. Shrink introduces novel algorithmic concepts in inference of most likely root causes, using probabilities to describe the strengths of dependencies. In Sherlock, we leverage these concepts, while extending them to deal with multi-level dependencies and with more complex operators that capture load-balancing and failover mechanisms. We compare the accuracy of our algorithm with Shrink and SCORE in Section 5.6.

5.3 The Inference Graph Model

We first describe our new model, called the Inference Graph, for representing the complex dependencies in an enterprise network. The Inference Graph forms the core of our Sherlock system. We then present our algorithm, called Ferret, that uses the model to probabilistically infer the faulty or malfunctioning components given real-world observations. We explain the details of how Sherlock constructs the Inference Graph, computes the required probabilities, and performs fault localization later in Section 5.4.

5.3.1 The Inference Graph

The Inference Graph is a labeled, directed graph that provides a unified view of the dependencies in an enterprise network, spanning services and network components. Figure 5-2 depicts a portion of the Inference Graph when a user accesses a network file share. Edges point from a *parent* node to a *child* node, denoting that the child node is dependent on all of the parent nodes. For example, accessing a file in the distributed share depends on contacting the Kerberos server for authentication, which in turn depends on the Kerberos server itself, as well as the routers and switches on the path from the user's machine to the Kerberos server. The graph encodes a joint probability distribution function for *how* the state of a child node is governed by the states of its parents. For example, one simple function is *or*, indicating that whenever any one of the parents is faulty, the child will be faulty. We introduce functions to capture unequal contributions from parents and also functions that model real-world dependencies such as failover and selection.

Formally, nodes in this graph are of three types. First, nodes that have no parents, shown at the top of the figure, are called *root-cause* nodes and correspond to independently failing entities—e.g., individual servers and links. The granularity of root-cause nodes in Sherlock is a computer (a machine with an IP address), a service (IP address, port), a router, or an IP link, although the model is extensible to root causes at a finer granularity. Second, nodes that have no children, shown at the bottom of the figure, are called *observation nodes* and

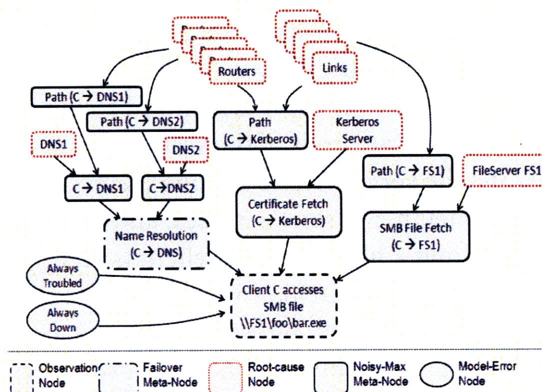


Figure 5-2 – Snippet of a partial Inference Graph that expresses the dependencies involved in accessing a file share. Dotted boxes represent physical components and software, dashed boxes denote external observations and ovals stand-in for un-modeled or external factors.

correspond to directly measurable events—e.g., a user's response time to the web server. There is a separate observation node for every client that accesses a network service. Finally, *meta-nodes* act as glue between the root-cause nodes and the observation nodes. In this chapter we present three types of meta-nodes, *noisy-max*, *selector* and *failover*. These nodes model the dependencies between root causes and observations; the latter two are needed to model load-balancers and failover redundancy respectively (described in detail in Section 5.3.1). The value of such a representation is to figure out the most likely explanation, i.e., the combination of root causes that is most likely faulty, given observations of multiple user activities.

The state of each node in the Inference Graph is expressed as a three-tuple: $(P_{up}, P_{troubled}, P_{down})$. P_{up} denotes the probability that the node is working normally. P_{down} is the probability that the node has experienced a fail-stop failure, such as when a server has crashed or a link is broken. Finally, $P_{troubled}$ is the probability that a node is troubled, which corresponds to the boxed area in Figure 5-1, where services, physical servers or links continue to function but users perceive poor performance. The sum of $P_{up} + P_{troubled} + P_{down} = 1$. We note that the state of root-cause nodes is apriori independent of other nodes in the Inference Graph, while the state of observation nodes can be uniquely determined from the state of root-causes.

An edge from node A to node B in the Inference Graph encodes the dependency that node A has to be in the *up* state for node B to be *up*. Not all dependencies are equal in strength. For example, a client cannot retrieve a file from a file server if the path to that file server is down. However, the client might still be able to retrieve the file even when the DNS server is down, if the file server's name to IP address mapping is found in the client's local DNS cache. Furthermore, the client may need to authenticate more (or less) often than he needs to resolve the server's name. To capture varying strengths in dependencies, edges in a Inference Graph are labeled with a *dependency probability*. A larger dependency probability indicates stronger dependency.

Finally, every Inference Graph has two special root-cause nodes – *always troubled (AT)* and *always down (AD)* – to model external factors not part of our model that might cause a user-perceived failure. The state of AT is set to (0, 1, 0) and that of AD is set to (0, 0, 1). We add an edge from these nodes to all the observation nodes (see §5.4.1).

To illustrate these concepts we revisit Figure 5-2, which shows a portion of the Infer-

ence Graph that models a user fetching a file from a network file server. The user activity of "fetching a file" is encoded as an observation node (dashed box) in the figure because Sherlock can measure the response time for this action. Fetching a file requires the user to perform three actions: (i) authenticate itself to the system, (ii) resolve the DNS name of the file server and (iii) access the file server. These actions themselves depend on other actions to succeed. Therefore, we model them as meta-nodes, and add edges from each of them to the observation node of "fetching a file." Since the client is configured with both a primary and secondary DNS server (DNS1 and DNS2), we introduce a failover meta-node. Finally, note that this snippet shows a single client and a single observation. When other clients access the same servers or use the same routers/links as those shown here, their observation nodes will be connected to the same root cause nodes as those shown to create the complete Inference Graph.

Propagation of State with Meta-Nodes

A crucial aspect of a probabilistic model is *how* the state of parent nodes governs the state of a child node. For example, suppose a child has two parents, A and B; the state of parent A is a three tuple (.8, .2, 0), which means its probability of being *up* is 0.8, *troubled* is 0.2 and *down* is 0, and the state of parent B is (.5, .2, .3). What, then, is the state of the child? While edge labels encode the strength of dependency, the nature of the dependency is encoded in the meta-node. Formally, the meta-node describes the state of the child node given the state of its parent nodes.

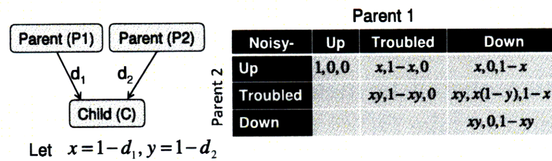
Noisy-Max Meta-Nodes are the simplest and most frequently used meta-node in Sherlock. *Max* implies that if any of the parents are in the *down* state, then the child is *down*. If no parent is *down* and any parent is *troubled*, then the child is *troubled*. If all parents are *up*, then the child is *up*. *Noisy* implies that unless a parent's dependency probability is 1.0, there is some chance the child will be *up* even if the parent is *down* or *troubled*. Formally, if the weight of a parent's edge is d, then with probability $(1 - d)$ the child is not affected by that parent.

Figure 5-3 presents a truth table for noisy-max when a child has two parents. Each entry in the truth table is the state of the child (i.e., its probability of being *up*, *troubled* and *down*) when $parent_1$ and $parent_2$ have states as per the column and row label respectively. As an example, the second row, third column, entry of the truth table shows the probability of the child being *troubled*, given that $parent_1$ is *down* and $parent_2$ is *troubled*:

$$\text{P(Child=Troubled | Parent1=Down, Parent2=Troubled)} = (1 - d_1) * d_2.$$

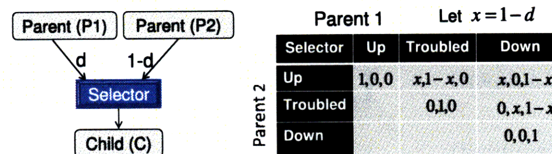
To explain, the child will be *down* unless $parent_1$'s state is masked by noise (prob $1 - d_1$). Further, if both parents are masked by noise, the child will be *up*. Hence the child is in *troubled* state only when $parent_1$ is drowned out by noise and $parent_2$ is not.

Selector Meta-Nodes are used to model load balancing scenarios. For example, a Network Load Balancer (NLB) in front of two servers hashes the client's requests and distributes requests evenly to the two servers. An NLB cannot be modeled using a noisy-max meta-node. Since half the requests go to each server, the client would depend on each server with a probability of 0.5. Using a noisy-max meta-node will assign the client a 25% chance of being



Noisy-	Up	Troubled	Down
Up	1,0,0	$x,1-x,0$	$x,0,1-x$
Troubled		$xy,1-xy,0$	$xy,x(1-y),1-x$
Down			$xy,0,1-xy$

Figure 5-3 – Truth Table for the noisy-max meta-node when a child has two parents. The values in the lower triangle are omitted for clarity. Each entry in the table denotes the $(P_{up}, P_{troubled}, P_{down})$ state of the child when the parent's states are as per the row and column entries.



Selector	Up	Troubled	Down
Up	1,0,0	$x,1-x,0$	$x,0,1-x$
Troubled		0,1,0	$0,x,1-x$
Down			0,0,1

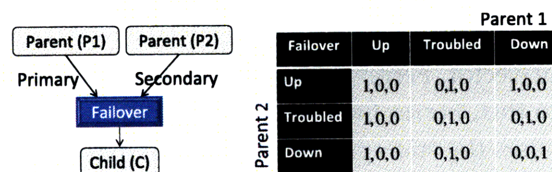
Figure 5-4 – Truth Table for the selector meta-node. A child node selects parent1 with probability d and parent2 with probability 1-d. The values in the lower triangle are omitted for clarity.

up even when both the servers are *down*, which is obviously incorrect. We use the selector meta-node to model NLB Servers and Equal Cost Multipath (ECMP) routing. ECMP is a commonly-used technique in enterprise networks where routers send packets to a destination along several paths. The path is selected based on a hash of the source and destination addresses in the packet. We use a selector meta-node when we can determine the set of ECMP paths available, but not which path a host's packets will use.

The truth table for the selector meta-node is shown in Figure 5-4, and it expresses the fact that the child is making a selection. For example, while the child may choose each of the parents with probability 50%, the selector meta-node forces the child to have a zero probability of being *up* when both its parents are *down* (first number in the Down,Down entry).

Failover Meta-Nodes capture the failover mechanism commonly used in enterprise servers. Failover is a redundancy technique where clients access primary production servers and failover to backup servers when the primary server is inaccessible. In our network, DNS, WINS, Authentication and DHCP servers all employ failover. Failover cannot be modeled by either the noisy-max or selector meta-nodes, since the probability of accessing the backup server depends on the failure of the primary server.

The truth table for the failover meta-node, for the simple case of failover between two servers, is shown in Figure 5-5. As long as the primary server is *up* or *troubled*, the child is not affected by the state of the secondary server. When the primary server is in the *down* state, the child is still *up* if the secondary server is *up*.



Failover	Up	Troubled	Down
Up	1,0,0	0,1,0	1,0,0
Troubled	1,0,0	0,1,0	0,1,0
Down	1,0,0	0,1,0	0,0,1

Figure 5-5 – Truth Table for the failover meta-node encodes the dependence that the child primarily contacts parent1, and fails over to parent2 when parent1 does not respond.

Time to Propagate State

A common concern with probabilistic meta-nodes is that computing the probability density for a child with n parents can take $O(3^n)$ time for a three-state model in the general case.[2] However, the majority of the nodes in our Inference Graph with more than one parent are noisy-max meta-nodes. For these nodes, we have developed the following equations that reduce the computation to $O(n)$ time.

$$\text{P(child up)} = \prod_j \left((1 - d_j) * (p_j^{trouble} + p_j^{down}) + p_j^{up}\right) \quad (5.1)$$

$$1 - \text{P(child down)} = \prod_j \left(1 - p_j^{down} + (1 - d_j) * p_j^{down}\right) \quad (5.2)$$

$$\text{P(child troubled)} = 1 - (\text{P(child up)} + \text{P(child down)}) \quad (5.3)$$

where p_j is the j'th parent, $(p_j^{up}, p_j^{trouble}, p_j^{down})$ is its probability distribution, and d_j is its dependency probability. The first equation implies that a child is *up* only when it does not depend on any parents that are not *up*. The second equation implies that a child is *down* unless every one of its parents are either not *down* or the child does not depend on them when they are *down*.

We show, in Appendix §G., that for all the meta-nodes used by Sherlock, propagation of state can be done in linear time. We note, however, that our Inference Graph model does allow arbitrary meta-nodes, and in particular meta-nodes wherein propagating state may have an exponential computational cost- $O(3^n)$, for a node with n parents. Such meta-nodes are feasible in practice only if these meta-nodes have no more than a couple of parents in the constructed graph.

5.3.2 Fault Localization on the Inference Graph

We now present our algorithm, Ferret, that uses the Inference Graph to localize the cause of user's poor performance. We define an **assignment-vector** to be an assignment of state to every root-cause node in the Inference Graph where each root-cause node has probability 1 of being either up, troubled, or down. The vector might specify, for example, that $link_1$ is troubled, $server_2$ is down and all the other root-cause nodes are up. The problem of localizing a fault is then equivalent to finding the assignment-vector that best explains the observations measured by the clients.

Ferret takes as input the Inference Graph and the measurements (e.g., response times) associated with the observation nodes. Ferret outputs a ranked list of assignment vectors ordered by a confidence value that represents how well they explain the observations. For example, Ferret could output that $server_1$ is troubled (and other root-cause nodes are up) with a confidence of 90%, $link_2$ is down (and other root-cause nodes are up) with 5% confidence, and so on.

For each assignment-vector, Ferret computes a score for how well that vector explains the observations. To do this, Ferret first sets the root causes to the states specified in the assignment-vector and uses the state-propagation techniques described in Section 5.3.1 to

[2]The naive approach to compute the probability of the child's state requires computing all 3^n entries in the truth-table and summing the appropriate entries.

propagate probabilities downwards until they reach the observation nodes. Then, for each observation node, Ferret computes a score based on how well the state of the observation node, i.e., the probabilities derived from state propagation, agree with the statistical evidence derived from the external measurements at this node. Section 5.4 provides the details of how we compute this score.

How can we search through all possible assignment vectors to determine the vector with the highest score? Clearly, the assignment vector that has the highest score for an observation is the most likely explanation. But, there are 3^r vectors given r root-causes, and applying the procedure described above to evaluate the score for each assignment vector would be infeasible. Existing solutions to this problem in machine learning literature, such as loopy belief propagation [129], do not scale to the Inference Graph sizes encountered in enterprise networks. Approximate localization algorithms used in prior work, such as Shrink [101] and SCORE [114], are more efficient. However, they are based on two-level, two-state graph models, and hence do not work on the Inference Graph, which is multi-level, multi-state and includes meta-nodes to model various artifacts of an enterprise network. The results in Section 5.6 clarify how Ferret compares with these algorithms.

Ferret uses an approximate localization algorithm that builds on an observation that was also made by Shrink [101].

Observation 5.3.1 *It is very likely that at any point in time only a few root-cause nodes are troubled or down.*

In large enterprises, there are problems all the time, but they are usually not ubiquitous.[3] We exploit this observation by not evaluating all 3^r assignment vectors. Instead, Ferret evaluates assignments that have no more than k root-cause nodes that are either troubled or down. Thus, Ferret first evaluates $2 * r$ vectors in which exactly one root-cause is troubled or down, next $2 * 2 * \binom{r}{2}$ vectors where exactly two root-causes are troubled or down, and so on. Given k, it is easy to see that Ferret evaluates at most $(2 * r)^k$ assignment vectors. Further, it is easy to prove that the approximation error of Ferret, that is, the probability that Ferret does not arrive at the correct solution (the same solution attained using the brute-force, exponential approach) decreases exponentially with k and becomes vanishingly small for $k = 4$ onwards [101]. Pseudo-code for the Ferret algorithm is shown in Algorithm 1.
Ferret uses another practical observation to speed up its computation.

Observation 5.3.2 *Since a root-cause is assigned to be* **up** *in most assignment vectors, the evaluation of an assignment vector only requires re-evaluating states at the descendants of root-cause nodes that are not* **up**.

Therefore, Ferret pre-processes the Inference Graph by assigning all root-causes to be up and propagating this state through to the observation nodes. To evaluate an assignment vector, Ferret re-computes only the nodes that are descendants of root-cause nodes marked troubled or down in the assignment vector. After computing the score for an assignment vector, Ferret simply rolls back to the pre-processed state with all root-causes in the *up* state. As there are never more than k root-cause nodes that change state out of the hundreds of

[3]We note that there are important cases where this observation might not hold, such as rapid malware infection and propagation.

Algorithm 3 Ferret{Observations O, Inference Graph G, Int X}

```
Candidates ← (up|trouble|down) assignments to root causes with at most k abnormal
    at any time
List_X ← {}                                          ▷ List of top X Assignment-Vectors
for R_a ∈ Candidates do                              ▷ For each Assignment-Vector
    Assign States to all Root-Causes in G as per R_a.
    Score(R_a) ← 1                                   ▷ Initialize Score
    for Node n ∈ G do                                ▷ Breadth-first traversal of G
        Compute P(n) given P(parents of n)           ▷ Propagate
    end for
    for Node n ∈ G_O do                              ▷ Scoring Observation Nodes
        s ← P(Evidence at n | state of n)            ▷ How well does R_a explain observation at n?
        Score(R_a) ← Score(R_a) * s                  ▷ Total Score
    end for
    Include R_a in List_X if Score(R_a) is in top X assignment vectors
end for
return List_X
```

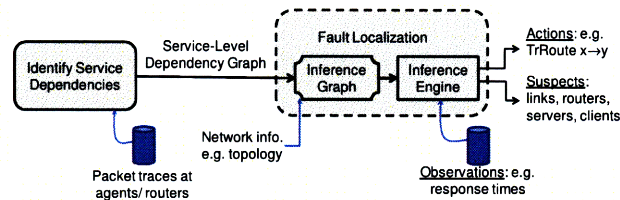


Figure 5-6 – Sherlock Solution Overview

root-cause nodes in our Inference Graphs, this reduces Ferret's time to localize by roughly two orders of magnitude without sacrificing accuracy.

We use the Ferret algorithm exactly as described above for results in this chapter. However, the inference algorithm can be easily extended to leverage whatever domain knowledge is available. For example, if prior probabilities on the failure rates of components are known (e.g., links in enterprise networks may have a much higher chance of being congested than down [132]), then Ferret can sort the assignment vectors by their prior probability and evaluate in order of decreasing likelihood to speed up inference.

5.4 The Sherlock System

Now that we have explained the Inference Graph model and the Ferret fault localization algorithm, we describe the Sherlock system that actually constructs the Inference Graph for an enterprise network and uses it to localize faults. Sherlock consists of a centralized *Inference Engine* and distributed *Sherlock Agents*. Sherlock requires no changes to routers, applications, or middleware used in the enterprise. It uses a three-step process to localize faults in the enterprise network, illustrated in Figure 5-6.

First, Sherlock computes a *service-level dependency graph (SLDG)* that describes the ser-

vices on which each user activity depends. Each Sherlock agent is responsible for monitoring the packets sent and received by one or more hosts. The agent may run on the host itself, or it may obtain packet traces via sniffing a nearby link or router. From these traces, the agent computes the dependencies between the services with which its host communicates and the response time distributions for each user activity. This information is then relayed to the inference engine as described in Section 5.5, where the engine aggregates the dependencies between services computed by each agent to form the SLDG. The SLDG is relatively stable, changing only when new hosts or applications are added to the network, and we expect it will be recomputed daily or weekly. For further details of SLDG construction, we direct the reader to [54].

Second, the inference engine combines the SLDG with the network topology to compute a unified Inference Graph over all services in which the operator is interested and across all Sherlock agents (Section 5.4.1). This step can be repeated as often as needed to capture changes in the network.

Third, the inference engine runs Ferret over the response time observations reported by the agents and the Inference Graph to identify the root-cause node(s) responsible for any observed problems. This step runs whenever agents see large response times.

5.4.1 Constructing the Inference Graph

Here we describe how the Inference Engine constructs a unified Inference Graph.

For each specified activity A (such as fetching web pages, fetching files), the inference engine creates an observation node for every client that reports response time observations for that activity. The engine then examines the service dependency information for these clients to identify the set of services $\mathcal{D}_{\mathcal{A}}$ that the clients are dependent on when performing activity A (such as name resolving through DNS, fetching credentials etc.). The engine links up the observation nodes with each of the service nodes with a noisy-max meta-node. The engine then recurses, by connecting each of the services in $\mathcal{D}_{\mathcal{A}}$ with other services that they in turn depend on (such as a web-server fetching data from a backend database). Once all the service meta-nodes have been created, the inference engine creates a root-cause node to represent each client and server and makes the root-cause node a parent of every meta-node that involve this client/server.

The inference engine then adds network topology information to the Inference Graph by using traceroute results reported by the agents. For each path between hosts in the Inference Graph, it adds a noisy-max meta node to represent the path and root-cause nodes to represent every router and link on the path. It then adds each of these root-causes as parents of the path meta-node. The inference engine connects each service meta-node to the path meta-node that traffic for that service traverses.

Optionally, the operators can tell the inference engine where load balancing or redundancy techniques are used in their network, and the engine will update the Inference Graphs, drawing on the appropriate specialized meta-node. The Inference Engine also applies heuristics specific to an environment, driven by naming conventions and configuration scraping, to figure out where to use specialized meta-nodes. For example, in our network the load-balanced web servers are named *sitename** (e.g., msw01, msw02). Our script looks for this pattern and replaces the default meta-nodes with selector meta-nodes. Similarly, a Sher-

lock agent examines its host's DNS configuration (using ipconfig) to identify where to place a failover meta-node to model the primary/secondary relationship between its name resolvers.

Finally, the inference engine assigns probabilities to the edges in the Inference Graph. The service-level dependency probabilities are directly copied onto corresponding edges in the Inference Graph. Recall that the special nodes *always troubled* and *always down* are connected to all observation nodes. The probability weight of such edges is chosen to approximate the probability that a failure is caused by a component that hasn't been modeled; we pick this to be a $\frac{1}{1000}$ chance. Further, the edges between a router and a path meta-node use a probability of 0.9999, which implies that there is a 1-in-10,000 chance that our network topology or traceroutes are incorrect and the router is not actually on the path. In our experience, Sherlock's results are not sensitive to the precise setting of these parameters (Section 5.6.1).

5.4.2 Fault Localization Using Ferret

As described in Section 5.3.2, Ferret uses a scoring function to compute how well an assignment vector being evaluated matches external evidence. A scoring function takes as input the probability distribution of the observation node and the external evidence for this node and returns a value between zero and one. A higher value indicates a better match. The score for an assignment vector is the product of scores for individual observations.

The scoring function for the case when an observation node returns an error or receives no response is simple; the score is equal to the probability of the observation node being down. For example, if the assignment vector correctly predicts that the observation node has a high probability of being down, its score will be high.

The scoring function for the case when an observation node returns a response time is computed as follows. The Sherlock agent tracks the history of response times and fits two Gaussian distributions to the historical data, namely $Gaussian_{up}$ and $Gaussian_{troubled}$. For example, the distribution in Figure 5-1 would be modeled by $Gaussian_{up}$ with a mean response time of 200 ms and $Gaussian_{troubled}$ with a mean response time of 2 s. If the observation node returns a response time t, the score of an assignment vector that predicts the observation node state to be $(p_{up}, p_{troubled}, p_{down})$ is computed as

$$p_{up} * Prob(t|Gaussian_{up}) + p_{troubled} * Prob(t|Gaussian_{troubled}) \tag{5.4}$$

It is easy to see that a better match yields better score; for example, if the response time t is well explained by the $Gaussian_{up}$ and the assignment vector correctly predicts that the observation node has a high probability of being up, the assignment vector will have a high score.

Together, we have:

$$Score\,(obs, prob) = \begin{cases} p_{up} * Prob(t|Gaussian_{up}) + p_{troubled} * Prob(t|Gaussian_{troubled}) & \\ & \text{observation } obs \text{ has response time } t \\ p_{down} & \text{observation } obs \text{ shows an error} \end{cases} \tag{5.5}$$

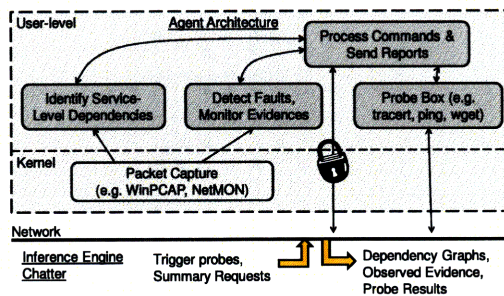


Figure 5-7 – The components of the Sherlock Agent, with arrows showing the flow of information. Block arrows show the interactions with the inference engine, which are described in the text.

When Ferret produces a ranked list of assignment vectors for a set of observations, it uses a statistical test to determine if the prediction is sufficiently meaningful to deserve attention. For a set of observations, Ferret computes the score that these observations would arise even if all root causes were up—this is the score of the null hypothesis. An operator can specify a threshold such that only predictions that beat the null hypothesis by more than the threshold are reported. Or, Sherlock can tune itself. Over time, the inference engine obtains the distribution of $Score(\text{best prediction}) - Score(\text{null hypothesis})$. If the score difference between the prediction and the null hypothesis exceeds the median of the above distribution by more than one standard deviation, the prediction is considered significant.

5.5 Implementation

We have implemented the Sherlock Agent, shown in Figure 5-7, as a user-level service (daemon) in Windows XP. The agent observes ongoing traffic from its host machine, watches for faults, and continuously updates a local version of the service-level dependency graph. The agent uses a WinPcap [43]-based sniffer to capture packets. The sniffer was augmented in several ways to efficiently sniff high volumes of data–even at an offered load of 800 Mbps, the sniffer misses less than 1% of packets. Agents learn the network topology by periodically running traceroutes to the hosts that appear in the local version of the service-level dependency graph. Sherlock could incorporate layer-2 topology as well, i.e., switches, wireless access points etc., if it were available. The Agent uses an RPC-style mechanism to communicate with the inference engine. Both the agent and the inference engine use role-based authentication to validate message exchanges.

The choice of a centralized inference engine makes it easier to aggregate information, but raises scalability concerns about CPU and bandwidth limitations. Back-of-the-envelope calculations show that both requirements are feasible for large enterprise networks. A Sherlock Agent sends 100B observation reports once every 300s. The inference engine polls each agent for its service-level dependency graph once every 3600s, and for most hosts in the network this graph is less than 40 KB. Even for an extremely large enterprise network with 10^5 Sherlock Agents, this results in an aggregate bandwidth of about 10 Mbps. We also note that Sherlock Agents need not be deployed at every machine in the network, for example, deploying at a couple of machines in each switched segment would suffice.

The computational complexity of fault localization scales as a small polynomial with

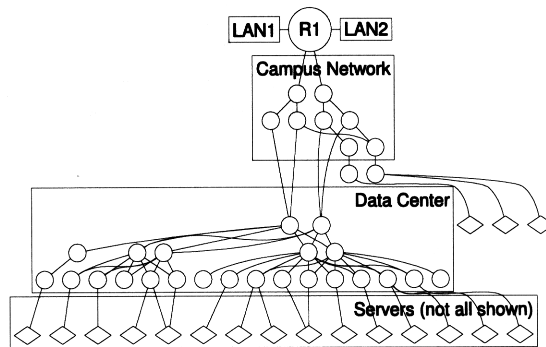


Figure 5-8 – Topology of the production network on which Sherlock was evaluated. Circles indicate routers; diamonds indicate servers; clients are connected to the two LANs shown at the top. Multiple paths exist between hosts and ECMP is used.

graph size, so we believe it is feasible even in large networks. Specifically, computational complexity is proportional to the number of assignment vectors that Ferret evaluates $\times$ the graph depth. Graph depth depends on the complexity of network applications, but is less than 10 for all the applications we have studied.

5.6 Evaluation

We evaluated our techniques by deploying the Sherlock system in a portion of a large enterprise network [4] shown in Figure 5-8. We monitored 40 servers, 34 routers, 54 IP links and 2 LANs for 3 weeks. Out of approximately 1,500 clients connected to the 2 LANs, we deployed Sherlock agents on 23 of them. In addition to observing ongoing traffic, these agents periodically send requests to the web- and file-servers, mimicking user behavior by browsing webpages, launching searches, and fetching files. We also installed packet sniffers at router R1 and at five other routers in the datacenter, enabling us to conduct experiments as if Agents were present on all clients and servers connected to these routers. These servers include the enterprise's main internal web portal, sales website, a major file server, and servers that provide name-resolution and authentication services. Traffic from the clients to the data center was spread across four disjoint paths using Equal Cost Multi-Path routing (ECMP).

In addition to the field deployment, we use both a testbed and simulations to evaluate our techniques in controlled environments (Section 5.6.1). The testbed and simulations enable us to study Ferret's sensitivity to errors in the Inference Graphs and compare its effectiveness with prior fault localization techniques, including Shrink [101] and SCORE [114].

5.6.1 Micro-Evaluation

We begin with a simple but illustrative example where we inject faults in our testbed (Figure 5-9). The testbed has three web servers, one in each of the two LANs and one in the data center. It also has an SQL backend server and supporting DNS and authentication servers (AD). $WebServer_1$ only serves local content and $WebServer_2$ serves content stored in

[4] Specifically, Microsoft's Corporate Network

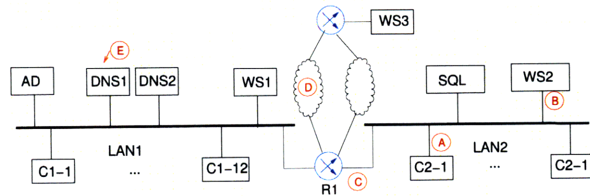


Figure 5-9 – Physical topology of the testbed. Hosts are squares, routers are circles. Example failure points indicated with circled letters. Hosts labeled C* are clients and WS* are webservers.

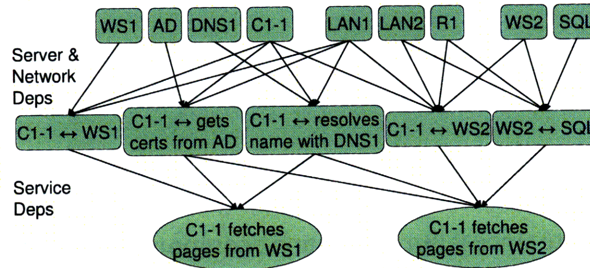


Figure 5-10 – Inference graph for client C_{1-1} accessing *WebServer*$_1$ (WS_1) and *WebServer*$_2$ (WS_2). For clarity, we elide the probability on edges, the specialized (failover) meta-node for DNS_1 and DNS_2, and the activities of other clients.

the SQL database. Note that the testbed shares routers and links with the production enterprise network, so there is substantial real background traffic. We use packet droppers and rate shapers along with CPU and disk load generators to create scenarios where any desired subset of clients, servers, routers, and links in the testbed appear as failed or overloaded. Specifically, an *overloaded link* drops 5% of packets at random and an *overloaded server* has high CPU and disk utilization.

Figure 5-10 shows the inference graph constructed by Sherlock, with some details omitted for clarity. The arrows at the bottom-level are the service-level dependencies inferred by our dependency discovery algorithm. For example, to fetch a web page from *WebServer*$_2$, client C_{1-1} has to communicate with DNS_1 for name resolution and AD for certificates. *WebServer*$_2$, in turn, retrieves the content from the SQL database. Sherlock builds the complete inference graph from the service-level dependencies as described in Section 5.4.1.

Unlike traditional threshold-based fault detection algorithms, Ferret localizes faults by correlating observations from multiple vantage points. To give a concrete example, if *WebServer*$_1$ is overloaded, traditional approaches would rely on instrumentation at the server to raise an alert once the CPU or disk utilization passes a certain threshold. In contrast, Ferret relies on the clients' observations of *WebServer*$_1$'s performance. Since clients do not experience problems accessing *WebServer*$_2$, Ferret can exclude LAN_1, LAN_2 and router R_1 from the potentially faulty candidates, which leaves *WebServer*$_1$ as the only candidate to blame. Ferret formalizes this reasoning into a probabilistic correlation algorithm (described in Section 5.3.2) and produces a list of suspects ranked by their likelihood of being the root cause. In the above case, the top root cause suspect was *WebServer*$_1$ with a likelihood of 99.9%, Router R_1 and a bunch of other components were next with a likelihood of 9.0*10^{-9}%. Ferret successfully identifies the right root cause while the likelihood of the second best candidate is negligibly small.

Ferret can also deal with multiple simultaneous failures. To illustrate this, we created a scenario where both *WebServer*$_1$ and one of the clients C_{1-3} were overloaded at the same

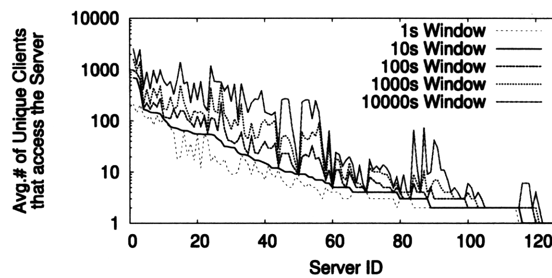


Figure 5-11 – Average number of unique clients accessing the 128 most popular servers in a 10-second time window. The top 20 servers have more than 70 unique clients in every 10 s window.

time. In this case, the top two candidates identified by Ferret are $WebServer_1 \cap C_{1-3}$ with a likelihood of 97.8% and $WebServer_1$ with a likelihood of 1.6%. $WebServer_1$ appears by itself as the second best candidate since failure of that one component explains most of the poor performance seen by clients, and the problems C_{1-3} reports with other services might be noise.

Ferret's fault localization capability is also affected by the number of vantage points. For example, in the testbed where $WebServer_2$ only serves content in the SQL database, Ferret cannot distinguish between congestion in $WebServer_2$ and congestion in the database. Observations from clients whose activities depend on the database but not $WebServer_2$ or vice-versa would resolve the ambiguity.

Ferret's ability to correctly localize failures depends on having observations from roughly the same time period that exercise all paths in the Inference Graph. To estimate the number of observations available, we measured the average number of unique clients that access a server during time windows of various sizes. We do this for the 128 most popular servers in our organization using time window lengths varying from 1 second to 10^5 seconds (roughly 3 hours). The data for Figure 5-11 were collected over a 24-hour period during a normal business day. It shows that there are many unique clients that access the same server in the same time window. For instance, in a time window of 10 seconds, at least 70 unique clients access every one of the top 20 servers. Given that there are only 4 unique paths to the data center and 4-6 DNS/WINS servers, we believe that accesses to the top 20 servers alone provide enough observations to localize faults occurring at most locations in the network. Accesses to less popular services leverage this information, and need only provide enough observations to localize faults in unshared components.

5.6.2 Evaluation of Field Deployment

We now report results from deploying Sherlock in a large enterprise's production network. We construct the Inference Graph using the algorithm described in Section 5.4.1. The resulting graph contains 2,565 nodes and 358 components that can fail independently.

Figure 5-12 shows the results of running the Sherlock system over a 5-day period. Each Y-axis value represents one component, e.g., a server, a client, a link, or a router, in the inference graph and the X-axis is time. A dot indicates a component is in the troubled or down state at a given time. During the 5 days, Ferret found 1,029 instances of performance problems. In each instance, Ferret returned a list of components ranked by their likelihood of being the root cause. This figure illustrates how Sherlock helps network managers by

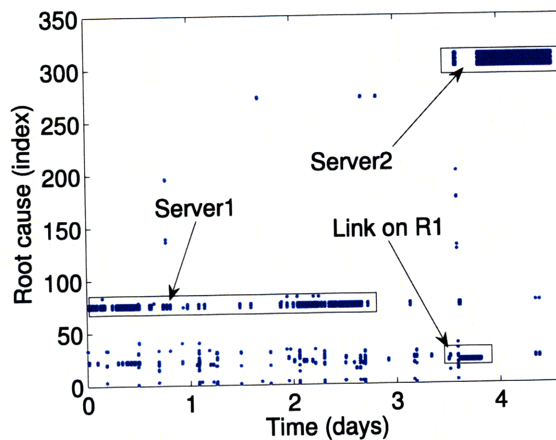


Figure 5-12 – Root causes of performance problems identified by Ferret over a 5-day period. Each Y-axis value represents a separate component in the inference graph and a dot indicates the component is troubled or down at that time.

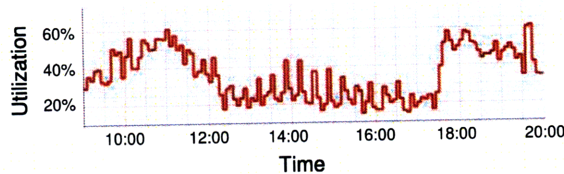


Figure 5-13 – 5-minute averages of link utilization reported by SNMP. Oscillations around 14:00 correspond to observed performance issue.

highlighting the components that cause user-perceived faults.

By Ferret's computations, 87% of the problems were caused by only 16 components (out of the 358 components that can fail independently). We were able to corroborate the 3 most notable problems marked in the figure with external evidence. The *Server*$_1$ incident was caused by a front-end web server with intermittent but recurring performance issues. In the *Server*$_2$ incident, another web server was having problems accessing its SQL backend. The third incident was due to recurring congestion on a link between R_1 and the rest of the enterprise network. In Figure 5-12, when Ferret is unable to determine a single root cause due to lack of information, it will provide a list of most likely suspects. For example in the *Server*$_1$ incident, there are 4 dots which represent the web server, the last links to and from the web server, and the router to which the web server is directly connected.

In a fourth incident, some clients were experiencing intermittent poor performance when accessing a web server in the data center while other clients did not report any problem. Ferret identified a suspect link on the path to the data center that was shared by only those clients that experienced poor performance. Figure 5-13 shows the MRTG [130] data describing the bandwidth utilization of the congested link. Ferret's conclusion on when the link was troubled matches the spikes in link utilization between 12:15 and 17:30. However, an SNMP-based solution would have trouble detecting this performance incident. First, the spikes in the link utilization are always less than 40% of the link capacity. This is common with SNMP counters, since those values are 5-minute averages of the actual utilization and may not reflect instantaneous high link utilization. Second, the 60% utilization at 11:00

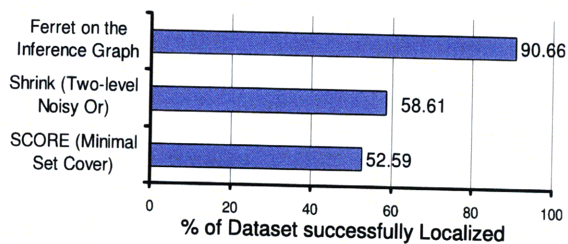


Figure 5-14 – Multi-level probabilistic model allows Ferret to correctly identify 30% more faults than approaches based on two-level probabilistic models (Shrink) or deterministic models (SCORE).

and 18:00 did not lead to any user-perceived problems, so there is no threshold setting that catches the problem while avoiding false alarms. Finally, due to scalability issues, administrators are unable to collect relevant SNMP information from all the links that might run into congestion.

5.6.3 Comparing Sherlock with Prior Approaches

Sherlock differs from prior fault localization approaches in its use of a multi-level inference graph instead of a two-level bipartite graph, and its use of probabilistic dependencies. Comparing Sherlock with prior approaches lets us evaluate the impact of these design decisions.

To perform the comparison, we need a large set of observations for which the actual root causes of the problems are known. Because it is infeasible to create such a set of observations using a testbed, we conduct experiments with a simulated dataset. We first created a topology and its corresponding inference graph that exactly matches that of the production network. Then we randomly set the state of each root cause to be troubled or down and perform a probabilistic walk through the inference graph to determine the state of all the observation nodes. Repeating this process 1,000 times produced 1,000 sets of observations for which we have the actual root causes. We then compare different techniques on their ability to identify the correct root cause given the 1,000 sets of observations.

Figure 5-14 shows that by using multi-level inference graphs, Ferret is able to correctly identify up to 32% more faults than Shrink, which uses two-level bipartite graphs. Figure 5-2 and Figure 5-10 show that multi-level dependencies do exist in real systems, and representing this type of dependency using bipartite graphs does lose important information. SCORE [114] uses a deterministic dependency model in which a dependency either exists or not and has a hard time modeling weak dependencies. For example, since names are often cached, name resolution (DNS) is a weak dependency. If the SCORE model includes these weak dependencies, i.e., considers them on par with other dependencies, it would result in many false positives; yet excluding these dependencies results in false-negatives.

5.6.4 Time to Localize Faults

We now study how long it takes Ferret to localize faults in large enterprise networks. In the following simulations, we start with the same topology (and hence the corresponding Inference Graph) as our field deployment. We create larger scale versions of the topology by scaling up the numbers of clients and servers evenly and use our measurements in Figure 5-11 to determine how many unique clients access a server in a time window. The experiments

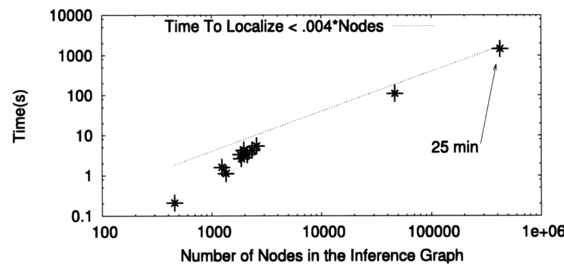


Figure 5-15 – The time taken by Ferret to localize a fault grows linearly with the number of nodes in the Inference Graph.

were run on an AMD Athlon 1.8GHz machine with 1.5GB of RAM. Figure 5-15 shows that the time it takes to localize injected faults grows almost linearly with the number of nodes in the Inference Graph. The running time of Ferret is always less than 4 ms times the number of nodes in the Inference Graph. With an Inference Graph of 500,000 nodes that contains 2,300 clients and 70 servers, it takes Ferret about 24 minutes to localize an injected fault. Note that Ferret is easily parallelizable (see pseudo-code in Algorithm 3) and implementing it on a cluster would linearly reduce the running time.

5.6.5 Impact of Errors in Inference Graph

Sometimes, errors are unavoidable when constructing inference graphs. For example, service-level dependency graphs might miss real dependencies (false negatives) or infer ones that don't exist (false positives). Traceroutes might also report wrong intermediate routers, for e.g., if they coincide with an ongoing OSPF convergence. To understand how sensitive Ferret is to errors in inference graphs, we compare the results of Ferret on correct inference graphs with those on perturbed inference graphs.

We deliberately introduce four types of perturbations into the inference graphs: First, for each observation node in the inference graph, we randomly add a new parent. Second, for each observation node, we randomly swap one of its parents with a different node. Third, for each edge in the inference graph, we randomly change its weight. Fourth, for each network-level path, we randomly add an extra hop or permute its intermediate hops. Note that the first three types of perturbations correspond to errors in service-level dependency graphs, specifically missing dependencies, incorrect dependencies and bad estimate of dependency probability respectively. Further, the fourth type corresponds to errors in traceroutes. We note that all our perturbations are probabilistic–given a probability p each node, or edge as the case may be, is perturbed with probability p.

We use the same inference graph as the one in the field deployment and perturb it in the ways described above. Figure 5-16 shows how Ferret behaves in the presence of each type of perturbation. Each point in the figure represents the average of 1,000 experiments. Note that Ferret is reasonably robust to all four types of errors. Even when half the paths/nodes/weights are perturbed, Ferret correctly localizes faults in 74.3% of the cases. Perturbing the edge weights seems to have the least impact while permuting the paths seems to be most harmful.

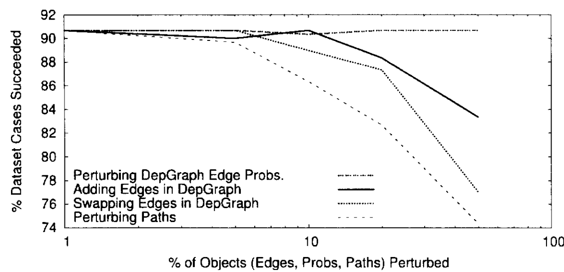


Figure 5-16 – Impact of errors in inference graph on Ferret's ability to localizing faults.

5.6.6 Modeling Redundancy Techniques

Specialized meta-nodes have important roles modeling load-balancing and redundancy, such as ECMP, NLB, and failover. Without these nodes, the fault localization algorithm may come up with unreasonable explanations for observations reported by clients. To evaluate the impact of specialized meta-nodes, we again used the same inference graph as the one in the field deployment. We created 24 failure scenarios where the root cause of each of the failures is a component connected to a specialized meta-node (e.g., a primary DNS server or an ECMP path). We then used Ferret to localize these failures both on inference graphs using specialized meta-nodes and on inference graphs using noisy-max meta-nodes instead of specialized meta-nodes.

In 14 cases where the root cause wouldn't affect observations, for example, if the failure was at a secondary server or a backup path, there is no difference between the two approaches–Ferret does not raise any alarm. In the remaining 10 cases where a primary server or path failed and impacted user's observations, Ferret correctly identified the root cause in all 10 of the cases when using specialized meta-nodes. In contrast, when not using specialized meta-nodes Ferret missed the true root cause in 4 cases.

5.6.7 Summary of Results

The key points of our evaluations are:

- In a field deployment we show that the Sherlock system is effective at identifying performance problems and narrowing down the root-cause to a small number of suspects. Over a five day period, Sherlock identified over 1,029 performance problems in the network, and narrowed down more than 87% of the blames to just 16 root causes out of the 350 potential ones. We also validated the three most significant outages with external evidence. Further, we show that Sherlock can localize faults that are overlooked by using existing approaches.
- Our simulations show that Sherlock is robust to noise in the Inference Graph and its multi-level probabilistic model helps localize faults more accurately than prior approaches that either use a two-level probabilistic model or two-level deterministic model.

5.7 Discussion

Comparing with Traditional Machine Learning: We note two departures from traditional machine learning. First, Sherlock bypasses the training phase. Traditional techniques such as Bayesian structure prediction [87] first learn the most likely structure, i.e., map of dependencies, and then use that structure to make predictions. Instead Sherlock directly uses the raw data to learn dependencies based on co-occurrence in time. Second, our fault localization algorithms are much simpler than traditional inference such as loopy belief propagation. Our reasons for this choice are (a) scalability concerns in running these algorithms on our large inference graphs and (b) need for not just the best explanation but a ranked list of explanations, say the top K, to help an admin focus his debugging effort.

Dependencies at the Granularity of an IP Address: To save money and datacenter space, some enterprises consolidate multiple servers onto a single piece of hardware via virtual machines (VMs). Since our techniques learn dependencies from packet traces, if multiple servers run inside VMs located at one IP address, we would mistakenly combine the individual dependencies of these servers. Luckily though, most datacenters assign each virtual server its own IP address for ease of configuration. The host machine, in this case, resembles a virtual Ethernet switch or IP router that multiplexes the VMs to the single physical network interface. Hence, the algorithms described here can learn the dependencies of each virtual server. More generally though, any dependency learner that merely looks at network packets suffers from a mis-match between the granularity at which it learns dependencies (IP address) and the granularity at which dependencies happen (a server). To resolve this mismatch, we believe requires a complementary host-based dependency learner that observes how network traffic at a machine de-multiplexes to individual processes.

Correlated Failures Due to Shared Risk: There are several instances of shared risk that are not easy to learn. Continuing the above example, the multiple VMs each of which has their own IP address, share an underlying physical host, and hence share risk. Software updates are another source of shared risk; for example, if a buggy patch were automatically installed on all the hosts, it would induce a correlated failure among the hosts. Other causes of shared risk include sharing a building, a power-source, a natural disaster or even an optical fiber that might get cut. There exist few direct ways to learn shared risk, primarily because they accrue from widely disparate sources. Traditional machine learning approaches can learn shared risk groups from (a large amount of) correlated failure data; for example, if root-causes $r_1, r_2, \ldots, r_n$ fail together often, they are likely to share a previously unknown root-cause. We believe that an important side-effect of the operational use of Sherlock would be the ability to guess at shared risk groups. Note that shared risk groups which fail more often will be learnt faster than those that fail rarely.

Using Shared Risk Groups: Regardless of how shared risk groups are discovered, we want to model them appropriately for fault localization. Our approach, as shown in Figure 5-17 is to create a new root-cause node for each shared-risk group, splice each pre-existing root-cause node (corresponding to a link, router, server etc.), say R_2, into two entities–one that corresponds to the un-shared component of the root-cause (here R_2^*), and another that is a descendant of the un-shared component and all the shared groups that contain this root-cause (here S_1, S_2). Note that modeling shared risk has little impact on the computational

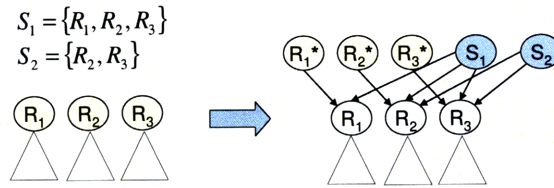
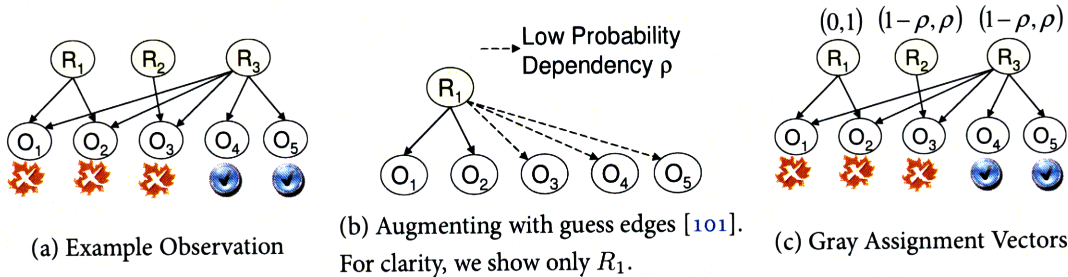


Figure 5-17 – To Model Shared Risk, Sherlock creates a root-cause for each shared risk and expands each pre-existing root-cause (say R_2) into an amalgam of (1) a root-cause corresponding to the unshared risk (R_2^*) and (2) all the shared groups that impact this root-cause (here S_1, S_2). Recall that Sherlock uses the *noisy-max* meta-node by default, i.e., R_2 inherits the state of its most trouble-some parent from among R_2^*, S_1, and S_2.



(a) Example Observation

(b) Augmenting with guess edges [101]. For clarity, we show only R_1.

(c) Gray Assignment Vectors

Figure 5-18 – An example observation where the only possible explanation is that $\{R_1, R_2\}$ have simultaneously failed. Such a setup is fragile to missing or incorrect dependencies. Shrink makes localization robust to errors by augmenting the graph with guess edges–low probability dependencies where none have been inferred as shown in (b). Since, the Shrink approach doesn't extend to multi-level Inference Graph, Sherlock instead uses gray-scale assignment vectors, i.e., every assignment vector allows root-cause nodes to have a small probability of being troubled or down.

complexity of Ferret, the number of root-causes increase linearly with the number of shared risk groups and the depth of the Inference Graph increases by atmost one.

Gray Assignment Vectors help deal with errors in dependency inference and also speed-up fault localization. Consider the example Inference Graph and observation in Figure 5-18(a). Given the observation, the only explanation, i.e., an assignment of state to the root-cause nodes, that has a non-zero probability is that nodes $\{R_1, R_2\}$ have failed (and R_3 is up). This setup is quite fragile. For example, we don't know how the remaining explanations compare among each other; for example here though R_1's failure explains more of the observation than R_2's failure, both have probability zero given the observation. Even worse, errors in inferring dependencies significantly impact localization—false negatives bloat up explanations and false positives lead to an over-constrained system with no feasible explanations. For example, if edge $R_1 \rightarrow O_3$ is a false negative, i.e., R_1 does impact O_3 but the dependency was missed, then the simplest explanation for the observation is that R_1 failed. Yet, the false negative causes the setup to overcompensate and include an additional root-cause R_2 in the explanation. In another example, suppose an $R_2 \rightarrow O_4$ false positive edge happens to be present, i.e., R_2 does not impact O_4 but a dependency between the two happened to be inferred, then there is no feasible explanation for the observation in Figure 5-18(a).

How can we make fault localization robust to errors in dependency inference? Our prior work, Shrink [101], shows that augmenting the Inference Graph with guess edges—low probability dependencies, where none have been inferred as shown in Figure 5-18(b)

helps. To see how, note that after augmenting with guess edges, three explanations have non-zero probability for the observation–$\{R_1, R_2\}$ failing together has probability 1, $\{R_1\}$ failing has probability ρ and $\{R_2\}$ failing has probability ρ^2 for some $\rho << 1$. Observe that the secondary explanations are ranked by the slack, i.e., the number of *errors* that need to be assumed for the explanation to be feasible.

Unfortunately, the Shrink approach of augmenting with guess edges does not extend to the multi-level Inference Graph that Sherlock builds (see Figure 5-2). Instead, Sherlock uses gray-scale assignment vectors. Recall (Section 5.3.2) that our fault localization algorithm identifies the assignment vector, i.e., a state assignment of *troubled* to a few root-causes while all other root-causes are *up*, that best explains the observation. Rather than having every root-cause be completely bad or good, Ferret allows all the *up* root-causes to be *troubled* with a small probability ρ, and computes the best among such "gray" assignment vectors (see Figure 5-18c). Note that just as with guess edges, alternate explanations are ranked by the number of gray assumptions necessary to make the explanation feasible. Further, unlike guess edges, gray assignment vectors make no assumptions about the topology of the Inference Graph. An important side-effect is that Ferret can limit to evaluating assignment vectors that have exactly one root-cause troubled or down. Since, even when multiple root-causes being simultaneously troubled were the best explanation, each of the individual root-causes would rank highly among the potential explanations.

Faults that change dependencies when they occur are awkward to deal with. For example, a fiber-cut on a link would cause a routing protocol like OSPF to move traffic onto alternate paths. If users observe poor response times because this alternate path is congested, Sherlock wouldn't be able to answer correctly until it updates the dependencies. This is a fundamental weakness with systems, like Sherlock, that learn dependencies at slow time scales. Fortunately, changing dependencies seem to be rare. In our several week deployment over all the paths and servers that we monitored, we saw paths change once and only one prominent web-server changed locations in the data-center.

Time Sensitivity: Sherlock localizes performance problems by correlating observations at different clients. In practice, however, these observations occur at different times and some may be quite stale, i.e., may report system state that is different from the current. To account for such disparity, Sherlock can be extended to weigh observations proportional to how far into the past an observation was made (or reported).

Logical Dependencies and Heisenbugs: We readily admit that Sherlock does not help with logical dependencies and heisenbugs, for example, suppose one end-user's banking transaction fails while others succeed, maybe because of a bug in the banking application. Such faults are certainly important and, we believe, require detailed modeling of individual applications. Instead, Sherlock focuses on performance problems that occur quite frequently and happen at key parts of the infrastructure that are shared by multiple users. Results from our deployment (Section §5.6) show the prevalence of such failures.

Incompletely Shared Dependencies: We note that Sherlock benefits from an incomplete sharing of dependencies among the key services; for example, all important servers tend to be located in one or a few data-centers and all traffic to and from these servers shares the few paths to the data-center. Yet services and end-users rarely have identical dependencies. Crucially, this means that observations for one service add information when correlated

with other observations. It is easy to see that the extreme where each service has a dedicated infrastructure would be a different challenge; identifying dependencies would be easier as there is fewer noise in packet traces; the Inference Graph will consist of many small partitions, one for each service; but there may be too few observations to localize performance problems. On the other hand, if services have identical dependencies, observations of these services are redundant. We observe from our deployments that the infrastructure at most edge networks (enterprises and educational institutions) exhibits such incomplete sharing.

5.8 Conclusions

In this chapter we describe Sherlock, a system that helps IT administrators localize performance problems across the network and servers in a timely manner without requiring modifications to existing applications and network components.

In realizing Sherlock, we make three important technical contributions: (1) We introduce a multi-level probabilistic inference model that captures the large sets of relationships between heterogeneous network components in enterprise networks. (2) We devise techniques to automate the construction of the inference graph by using packet traces, traceroute measurements, and network configuration files. (3) We describe an algorithm that uses an Inference Graph to localize the root cause of the network or service problem.

We evaluate our algorithms and mechanisms via testbeds, simulations and field deployment in a large enterprise network. Our key findings are: (1) service dependencies are complicated and continuously evolving over time thus justifying a need for automatic approaches to discovering them. (2) Our fault localization algorithm shows promise in that it narrows down the root cause of the performance problem to a small number of suspects thereby helping IT administrators track down frequent user complaints, and finally, (3) comparisons to other state-of-art techniques show that our fault localization algorithm is robust to noise and it localizes performance problems more quickly and accurately.

Chapter 6

Concluding Remarks

This thesis presents two classes of techniques to harden networked systems.

First, we enable systems to confront unpredictable inputs such as overloads, failures and attacks by designing adaptive solutions that quickly re-optimize how resources are used. TeXCP re-balances ISP backbones by moving traffic onto alternate paths when links fail or become congested. Such responsive engineering of traffic simplifies the ISP operator's job and lets an ISP defer investing in capacity upgrades as the same user demands can now be carried at lower peak utilizations. Kill-Bots manages server resources better during a denial-of-service attack. Kill-Bots distinguishes attackers from legitimate users through CAPTCHA-based authentication, admits just as many new connections as the server can serve, and eventually detects and blocks attackers by observing their reaction to CAPTCHAs. Such adaptive authentication coupled by admission control lets a server handle orders of magnitude higher attack rates and also improves performance during overloads caused by legitimate users, i.e., flash crowds.

Second, we present techniques to learn the intricate dependencies in functioning edge networks (enterprises, educational institutions, etc.) without any instrumentation of the deployed servers and with little pre-knowledge about the applications, clients or servers in the network. eXpose allows an operator to build mental models of whats going on in his network and expose misconfigurations. Further, we show that combining dependencies with observations from multiple vantage points into a probabilistic Inference Graph model simplifies troubleshooting. From among the many possible causes of a performance problem, Sherlock presents a small probabilistic list of the most likely causes of failures that an operator could examine.

Much work remains to be done. Operational experience with tools like eXpose through longer deployments in production networks, would refine our rule mining techniques, for example, how to prioritize rules that are worth human attention versus rules that are similar to others that an operator has already tagged as benign, and at the same time present us with more use cases, for example, can rule mining be used to check for policy violations. An example policy could be that clients can talk to the database server only after fetching credentials from the kerberos server, and an operator would care about being automatically told about clients violating the policy. Significant new venues of complexity in networked systems remain fairly unexplored. For example, enterprise datacenters house hundreds of thousands of machines, perform internal computation continously (e.g., building a global

search index and keeping it fresh) and handle enormous volumes of incoming traffic. What are the common modes of failure inside a data-center? Would TeXCP-like traffic engineering, or Kill-Bots-like attack prevention techniques or Sherlock-like fault localization techniques apply to the data-center?

A. Analysing Kill-Bots

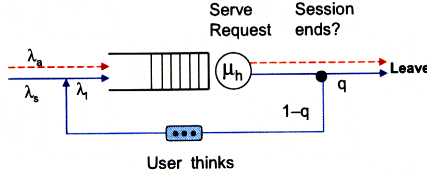


Figure -1 – Model of a server that does not use authentication.

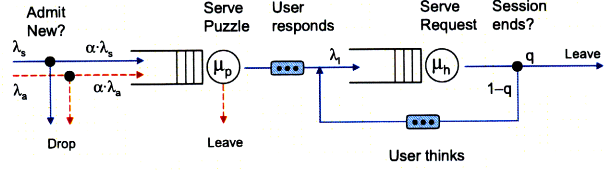


Figure -2 – A server that implements some authentication mechanism.

(a) Server with No Authentication (Base Server): We first analyze the performance of a Web server that does not use any authentication or admission control. Figure -1 shows a model for such a server. The server serves HTTP requests at an average rate μ_h. Attacking HTTP requests arrive at a rate λ_a. Legitimate users/sessions, on the other hand, arrive at an average rate λ_s, where a given session consists of some random number of HTTP requests. When their request is served, legitimate users either leave the web site with probability $\frac{1}{q}$ or send another request with probability $1 - \frac{1}{q}$ potentially after some thinking time. At the input to the queue in Figure -1, the average rate of legitimate requests, denoted λ_l, equals the sum of λ_s plus the rate from the feedback loop of subsequent requests, where the latter is $1 - \frac{1}{q}$ times the departure rate from the server of legitimate requests, denoted λ_d:

$$\lambda_l = \lambda_s + (1 - \frac{1}{q})\lambda_d \tag{1}$$

As long as the server occupancy is less than 1, i.e., the server keeps up with the incoming requests though possibly with some delay, arrivals should equal departures $\lambda_d = \lambda_l$. Solving equation 1 for λ_l yields:

$$\lambda_l = q\lambda_s. \tag{2}$$

One can view q as the mean number of requests per session.

We make some simplifying assumptions. First, we assume that the system of Figure -1 is in steady state, i.e., that a time interval exists where the parameter values are constant. Second, we assume that the server is work-conserving, i.e., it processes requests if any are present. However, our analysis is based simply on the mean values of arrival rates and service times; hence it assumes very little about the distributions or independence of the interarrival times of legitimate sessions, or of attacker requests or of service times. Under these conditions, the occupancy of the server, ρ, i.e., the fraction of time the server is busy, will be the offered load, whenever this load is less than 1. Otherwise, ρ will be capped at 1. The offered load is the arrival rate of requests $\lambda_a + \lambda_l$ times the mean service time $\frac{1}{\mu_h}$, thus

$$\rho = \min\left(\frac{\lambda_a + q\lambda_s}{\mu_h}, 1\right). \tag{3}$$

The server's goodput is the fraction of the occupancy due to processing legitimate requests:

$$\rho_g^b = \frac{q\lambda_s}{q\lambda_s + \lambda_a}\rho = \min\left(\frac{q\lambda_s}{\mu_h}, \frac{q\lambda_s}{q\lambda_s + \lambda_a}\right). \quad (4)$$

When there are no attackers, the server's goodput is simply its occupancy, which is $\frac{q\lambda_s}{\mu_h}$. However for large attack rates, the server's goodput decreases proportionally to the attack rate. Moreover, for offered loads greater than one, the goodput could degrade further depending on how the server's operating system handles congestion.

(b) Server Provides Authentication and Admission Control: Next, we present a general model of the performance of a server that implements some authentication mechanism Figure -2 illustrates the model. The server divides its time between authenticating new clients and serving the ones already authenticated. A new client is admitted to the authentication phase with a probability α that depends on the occupancy of the server, i.e., how busy the server is. Authentication costs $\frac{1}{\mu_p}$ cpu time for each request, while receiving a request and potentially dropping it costs $\frac{1}{\mu_d}$. (When applied to Kill-Bots, $\frac{1}{\mu_p}$, is the average time to send a graphical test). Other parameters are the same as before. The server spends fractions of time, ρ_p, on authenticating clients, ρ_d, on receiving and dropping clients, and ρ_h, on serving HTTP requests from authenticated clients. Since the server serves HTTP requests only from authenticated clients, the goodput, ρ_g equals ρ_h.

Using the same general assumptions as for Figure -1, we wish to determine the value of the admit probability, α^*, that maximizes the goodput, ρ_h, given the physical constraint that the server can not be busy more than 100% of the time. That is:

$$\max_{0\leq\alpha\leq 1} \quad \rho_h \quad (5)$$

$$\text{such that } \rho_d + \rho_p + \rho_h \leq 1, \quad (6)$$

$$\text{and given constraint 6: } \rho_p = \alpha\frac{\lambda_a + \lambda_s}{\mu_p} \quad (7)$$

$$\rho_h = \alpha\frac{q\lambda_s}{\mu_h} \quad (8)$$

$$\rho_d = (1-\alpha)\frac{\lambda_a + \lambda_s}{\mu_d} \quad (9)$$

Since the goodput ρ_h is increasing in α (Equation 8), we would want to make α as big as possible, subject to the constraint (6). Consider first the simple case where $\rho_p + \rho_h$ is strictly less than 1 even when all clients are admitted to the authentication step, i.e., $\alpha = 1$. Then the optimal choice for α, denoted α^*, is 1, and the maximal goodput ρ_g^* is $\frac{q\lambda_s}{\mu_h}$. For the more interesting case, now suppose that the constraint (6) would be binding at some value of α less than or equal to one. The value of α at which the constraint first becomes binding is the largest feasible value for α, and thus is the value that maximizes the goodput. Substituting Equation 7 and 8 into $\rho_d + \rho_p + \rho_h = 1$ and solving for α yields the maximizing value $\alpha*$.

Summarizing the two cases, the optimal value for the admission probability is:

$$\alpha^* = \min\left(\frac{\mu_p - C(\lambda_a + \lambda_s)}{(Bq+1)\lambda_s + \lambda_a - C(\lambda_a + \lambda_s)}, 1\right), \quad \text{where} \quad B = \frac{\mu_p}{\mu_h}, C = \frac{\mu_p}{\mu_d}. \tag{10}$$

We note that receiving a request and potentially dropping it costs much less than serving a puzzle, i.e, $\mu_d >> \mu_p \Rightarrow C = \frac{\mu_p}{\mu_d} \to 0$. Using this and substituting α^* into (8) yields the maximal goodput:

$$\rho_g^* = \rho_h^* = \min\left(\frac{Bq\lambda_s}{(Bq+1)\lambda_s + \lambda_a}, \frac{q\lambda_s}{\mu_h}\right). \tag{11}$$

Note that since the authentication is performed in the interrupt handler, it preempts serving HTTP requests. The expressions for occupancy Equation 7 and 8 can incorporate the constraint Equation 6 as:

$$\rho_p = \min\left(\alpha\frac{\lambda_a + \lambda_s}{\mu_p}, 1\right), \tag{12}$$

$$\rho_d = \min\left((1-\alpha)\frac{\lambda_a + \lambda_s}{\mu_d}, 1\right), \tag{13}$$

$$\rho_h = \min\left(\alpha\frac{\lambda_s}{q\,\mu_h}, 1 - \rho_p - \rho_d\right). \tag{14}$$

Setting α to 1 in Equations 12, 13, 14 yields the goodput, $\rho_g^a = \rho_h$, obtained when the web server tries to authenticate all new clients regardless of the offered load.

$$\rho_g^a = \min\left(\frac{q\lambda_s}{\mu_h}, \max\left(0, 1 - \frac{\lambda_a + \lambda_s}{\mu_p}\right)\right). \tag{15}$$

B. Interaction between Admission Control and Re-transmissions

There is a subtle interaction between probabilistic dropping and retransmission of requests [95]. Network stacks re-transmit SYNs and data packets whereas users may refresh pages. A dropper that drops a fixed proportion of incoming requests, will be letting in requests that have been retransmitted after being dropped a couple of times.

This interaction is wasteful in many ways. First, if α is the probability of admitting requests, the average request being admitted will have been dropped $\frac{1}{\alpha} - 1$ times already. To see this, note that,

$$Prob(\text{ Admitted after } i \text{ attempts }) = \alpha \cdot (1-\alpha)^{i-1}, \tag{16}$$

and the average number of attempts for an admitted request is

$$E(\text{ Number of Attempts for Admission }) = \frac{1}{\alpha} - (n + \frac{1}{\alpha})(1-\alpha)^n \approx \frac{1}{\alpha} \text{ for large } n. \tag{17}$$

Hence, requests simply take longer, i.e., the average response time is inflated by the extra $(\frac{1}{\alpha} - 1)$ (see Equation 17) round trip times involved in retransmitting each request. Second, if λ is the rate of new request arrivals and the number of retransmissions is n, then the number of arrivals at the server $\breve{\lambda}$ are

$$\breve{\lambda} = \lambda + (1-\alpha)\lambda + (1-\alpha)^2\lambda + \ldots (1-\alpha)^n\lambda \;=\; \lambda \cdot \frac{1-(1-\alpha)^{n+1}}{\alpha}. \tag{18}$$

This is because arrivals at the server include the retransmissions of requests that were dropped in each of the n previous time-slots. It is easy to see the inflation in arrival rate. For example, at $\alpha = .1, n = 3$, the average arrival rate inflation is $\frac{\breve{\lambda}}{\lambda} = 3.439$ (by substituting values in Equation 18). Such inflated arrival rates waste server cycles. Finally, if the server admits an α fraction of arrivals with the goal of admitting $\alpha \cdot \lambda$ requests, it would instead admit $\alpha \cdot \breve{\lambda}$ arrivals which is often much larger (see Equation 18).

Kill-Bots ameliorates this adverse interaction in two ways. First, Kill-Bots eliminates most retransmissions by responding to network-level packets – it sends out FINs after a puzzle and issues RSTs at other times. Second, Kill-Bots persistently drops requests, i.e., when the admission control drops a request, Kill-Bots sends out a *been_dropped* HTTP cookie (along with an empty page). Requests with the *been_dropped* cookie are dropped for a short period of time without invoking admission control. Persistent dropping ensures that retransmissions are transparent to admission control.

C. Stability of The Kill-Bots Controller

How do you analyze the stability of a Kill-Bots server?

Base Server: Analyzing the base server, i.e., without authentication or admission control, is quite straightforward. If $\ell(t)$ is the load at time t, $\lambda(t)$ is the input rate at time t, then

$$\ell(t+1) = \frac{\lambda(t)}{\mu_h} + (1 - \frac{1}{q})\ell(t). \tag{19}$$

Here, we assume that load and requests arrive at discrete times and that $\lambda(t) = \lambda_s(t) + \lambda_a(t)$ is the aggregate request rate. To analyze stability we take the Z transform. Suppose $\lambda(Z)$ and $\ell(Z)$ are the respective z-transforms, then

$$Z\ell(Z) = \frac{\lambda(Z)}{\mu_h} + (1 - \frac{1}{q})\ell(Z), \tag{20}$$

which implies that the transfer function

$$H(Z) = \frac{\ell(Z)}{\lambda(Z)} = \frac{\frac{1}{\mu_h}}{Z - \left(1 - \frac{1}{q}\right)}, \text{ with pole } Z^* = 1 - \frac{1}{q}. \tag{21}$$

Such a system is stable, in the bounded input bounded output (BIBO) sense whenever the magnitude of the pole is less than one. Here, the pole Z^* is guaranteed to be smaller than 1, since q the mean number of requests in each session is a positive number that is atleast 1. This means that the server load is going to be bounded as long as the inputs are bounded.

Server with Authentication: A server that authenticates incoming requests will have load due to legitimate users just as the base server,

$$\ell_g(t+1) = \frac{\lambda_s(t)}{\mu_h} + (1 - \frac{1}{q})\ell_g(t), \tag{22}$$

but its total load includes authenticating both legitimate and attacker requests,

$$\ell(t) = \frac{\lambda_a(t) + \lambda_s(t)}{\mu_p} + \ell_g(t). \tag{23}$$

As before, we take the Z-transforms of Equations 22 and 23,

$$Z\ell_g(Z) = \frac{\lambda_s(Z)}{\mu_h} + (1 - \frac{1}{q})\ell_g(Z), \text{ and} \tag{24}$$

$$\ell(Z) = \frac{\lambda_a(Z) + \lambda_s(Z)}{\mu_p} + \ell_g(Z). \tag{25}$$

Since we have two types of input, we resort to state-space analysis with a bi-variate input vector $\begin{bmatrix} \lambda_a(Z) \\ \lambda_s(Z) \end{bmatrix}$, an internal state variable $\ell_g(Z)$, and an output variable $\ell(Z)$. Eliminating $X(Z)$ from the two equations obtains the transfer function.

$$\ell(Z) = \left(\frac{1}{\mu_h \left(Z - \left(1 - \frac{1}{q}\right)\right)} \begin{bmatrix} 0 \\ 1 \end{bmatrix}^T + \frac{1}{\mu_p} \begin{bmatrix} 1 \\ 1 \end{bmatrix}^T \right) \begin{bmatrix} \lambda_a(Z) \\ \lambda_s(Z) \end{bmatrix}. \tag{26}$$

The pole of this transfer function is $Z^* = 1 - \frac{1}{q}$, and the server is stable, in the bounded

input bounded output sense since q, the mean number of requests is atleast 1, and hence $|Z^*| < 1$.

Server with Authentication and Admission Control: Now, a Kill-Bots server that does both authentication and admission control has,

$$\ell_g(t+1) = \alpha(t)\frac{\lambda_s(t)}{\mu_h} + (1 - \frac{1}{q})\ell_g(t), \tag{27}$$

$$\alpha(t+1) = \alpha(t) f(\beta - \ell(t)), \text{ where } f \text{ is as per Equation 2.6, and} \tag{28}$$

$$\ell(t) = \alpha(t)\frac{\lambda_a(t) + \lambda_s(t)}{\mu_p} + \ell_g(t). \tag{29}$$

This is a non-linear system and can be analysed by state space analysis with input vector $\begin{bmatrix} \lambda_g(t) \\ \lambda_s(t) \end{bmatrix}$, internal state vector $\begin{bmatrix} \ell_g(t) \\ \alpha(t) \end{bmatrix}$, and an output variable $\ell(t)$.

D. Proof of TeXCP Stability due to Explicit Feedback

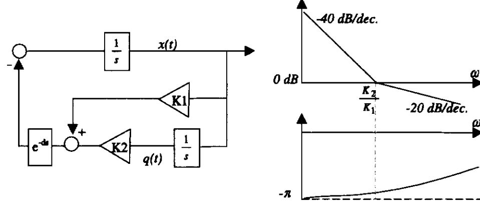


Figure -3 – The feedback loop and the Bode plot of its open loop transfer function.

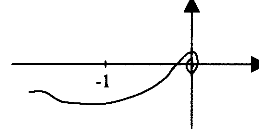


Figure -4 – The Nyquist plot of the open-loop transfer function with a very small delay.

This proof is similar to that in [107], and we include here for sake of completeness, (i.e.,. we claim no credit for it).

Model & Assumptions: We assume a fluid model of traffic, a single bottleneck along each path, the RTT is a constant d, and the IE flows have infinite demands. We also ignore boundary conditions.

Consider a bottleneck of capacity c traversed by N IE flows. Let $r_i(t)$ be the sending rate of IE flow i at time t. The aggregate traffic rate on the link is $\phi(t) = \sum r_i(t)$. The router sends some aggregate feedback every T_p. The feedback reaches the TeXCP agents after a round trip time, d. Assuming the IE flows have enough demands, the change in their aggregate rate per second is equal to the feedback divided by T_p.

$$\frac{d\phi(t)}{dt} = \sum \frac{dr_i(t)}{dt} = \frac{1}{T_p}\left(-\alpha \cdot (\phi(t-d) - c) - \beta \cdot \frac{q(t-d)}{T_p}\right).$$

The whole system can be expressed using the following delay differential equations.

$$\dot{q}(t) = \phi(t) - c \tag{30}$$

$$\dot{\phi}(t) = -\frac{\alpha}{T_p}(\phi(t-d) - c) - \frac{\beta}{T_p^2}q(t-d) \tag{31}$$

Proof Idea: This is a linear feedback system with delay. The stability of such systems may be studied by plotting their open-loop transfer function in a Nyquist plot. We prove that the system satisfies the Nyquist stability criterion. Further, the gain margin is greater than one and the phase margin is positive independently of delay, capacity, and number of IE flows.[1]

Proof Let us change variable to $x(t) = \phi(t) - c$.

$$\dot{q}(t) = x(t)$$

$$\dot{x}(t) = -K_1 x(t-d) - K_2 q(t-d)$$

[1] The gain margin is the magnitude of the transfer function at the frequency $-\pi$. The phase margin is the frequency at which the magnitude of the transfer function becomes 1. They are used to prove robust stability.

$$K_1 = \frac{\alpha}{T_p} \quad \text{and} \quad K_2 = \frac{\beta}{T_p^2},$$

The system can be expressed using a delayed feedback (see Figure -3). The open loop transfer function is:

$$G(s) = \frac{K_1 \cdot s + K_2}{s^2} e^{-ds}$$

For very small $d > 0$, the closed-loop system is stable. The shape of its Nyquist plot, which is given in Figure -4, does not encircle -1.

Next, we prove that the phase margin remains positive independent of the delay. The magnitude and angle of the open-loop transfer function are:

$$|G| = \frac{\sqrt{K_1^2 \cdot w^2 + K_2^2}}{w^2},$$

$$\angle G = -\pi + \arctan \frac{wK_1}{K_2} - w \cdot d.$$

The break frequency of the zero occurs at $w_z = \frac{K_2}{K_1}$.

To simplify the system, we choose α and β such that the break frequency of the zero w_z is the same as the crossover frequency w_c (frequency for which $|G(w_c)| = 1$). Substituting $w_c = w_z = \frac{K_2}{K_1}$ in $|G(w_c)| = 1$ leads to $\beta = \alpha^2\sqrt{2}$.

To maintain stability for any delay, we need to make sure that the phase margin is independent of delay and always remains positive. This means that we need $\angle G(w_c) = -\pi + \frac{\pi}{4} - \frac{\beta}{\alpha}\frac{d}{T_p} > -\pi$. Since $T_p > d$ by design, we need $\frac{\beta}{\alpha} < \frac{\pi}{4}$. Substituting β from the previous paragraph, we find that we need $0 < \alpha < \frac{\pi}{4\sqrt{2}}$. In this case, the gain margin is larger than one and the phase margin is always positive (see the Bode plot in Figure -3). This is true for any constant delay, capacity, and number of IE flows. For another analysis of the same system with fewer assumptions, we refer the reader to a recent publication [55].

E. Proof of TeXCP Load Balancer Convergence

Preliminaries: Before going into the proof, we manipulate the TeXCP equations to obtain an expression of the change in link utilization. The proof uses this expression to show that the max-utilization decreases monotonically and stabilizes at a balanced load. Our variable definitions are in Table 3.2 and our assumptions are in §3.4.

Recall Equation 3.5 in a simplified form:

$$\Delta x_{sp} = \begin{cases} x_{sp}(n)(\overline{u}_s(n) - u_{sp}(n)), & \forall p, u_{sp} > u_s^{min}, \\ \epsilon + x_{sp}(n)(\overline{u}_s(n) - u_{sp}(n)), & p, u_{sp} = u_s^{min}. \end{cases} \tag{32}$$

Note that perturbing by $\epsilon > 0$ increases the value of Δx for atmost one path and Δx would otherwise sum to zero. Hence, in all but the degenerate case of $x_{sp}(n+1) = 0$, the re-normalization in Equation 3.8 only reduces the change in x_{sp} further. It is easy to verify that:

$$x_{sp}(n+1) \leq x_{sp}(n) + \Delta x_{sp}(n). \tag{33}$$

Our proof goes in discrete steps. In each step, each TeXCP agent s applies Equation 32 to adjust the amount of traffic r_{sp} sent along each path. The change in traffic Δr_{sp} is,

$$\Delta r_{sp} = R_s(x_{sp}(n+1) - x_{sp}(n)) \leq R_s \Delta x_{sp}(n). \tag{34}$$

The last part is from using Equation 33.

The change in the utilization of link l, due to this new traffic assignment, is the sum of the changes over all the paths traversing the link.

$$\begin{aligned} u_l(n+1) &\leq u_l(n) + \sum_s \sum_{p \in P_s, p \ni l} \frac{R_s}{C_l} \Delta x_{sp}(n) \\ &\leq u_l(n) + \sum_s \sum_{p \in P_s, p \ni l} \frac{R_s}{C_l} (\epsilon + x_{sp}(n)(\overline{u}_s(n) - u_{sp}(n))). \end{aligned}$$

The first part is from Equation 34 and the second is by substituting with Equation 32. By definition, a TeXCP agent uses the maximum utilization along a path as u_{sp}; hence, u_{sp} is at least as large as the utilization on links along path p, i.e., $\forall l \in p, \;\; u_{sp}(n) \geq u_l(n)$. Replacing u_{sp} by u_l in the last equation, and noting that $\sum_s \sum_{p \in P_s, p \ni l} \frac{R_s x_{sp}(n)}{C_l} = u_l(n)$, we get:

$$u_l(n+1) \leq u_l(n)(1 - u_l(n)) + \sum_s \sum_{p \in P_s, p \ni l} \frac{R_s}{C_l} (\overline{u}_s(n) x_{sp}(n) + \epsilon). \tag{35}$$

Assertion: We will prove that in every step, the maximum link utilization in the network always decreases without oscillations, i.e.,

$$\max_l u_l(n+1) < \max_l u_l(n).$$

Assume link i has the maximum utilization at step n and link j has the maximum utilization

at step $n+1$. **Thus, we need to prove that**

$$u_j(n+1) < u_i(n). \tag{36}$$

Define $\overline{u}^M(n)$ as the maximum average utilization $\max_s \overline{u}_s(n)$ over all TeXCP agents. We identify three cases.

Case 1: $u_j(n) > \overline{u}^M(n)$ **[Highly Utilized in Past]**
In this case, at step n, u_j was larger than the largest $\overline{u}_s$. Every TeXCP agent sending traffic on a path containing link j will decrease its traffic share x_{sp} according to equation 32. Thus, the utilization of link j decreases $u_j(n+1) < u_j(n)$. But, $u_j(n) \leq u_i(n)$ because i is the max-utilization link at step n. Thus, $u_j(n+1) < u_i(n)$ and we are done.

Case 2: $u_j(n) < \overline{u}^M(n)$ **[Low Utilization in Past]**
We want to bound the increase in traffic on link j. Substituting $\overline{u}_s(n) \leq \overline{u}^M(n)\ \forall s$ in Equation 35, and using $\sum_s \sum_{p \in P_s, p \ni j} \frac{R_s x_{sp}(n)}{C_j} = u_j(n)$, we get:

$$u_j(n+1) \leq u_j(n)(1 - u_j(n) + \overline{u}^M(n)) + \sum_s \sum_{p \in P_s, p \ni j} \frac{R_s \epsilon}{C_j} \tag{37}$$

Each TeXCP agent independently picks an ϵ as follows, where N is the total number of IE pairs, P is the maximum number of paths that an IE pair can use, C_{min} is the minimum capacity over all links in the network, and u_s^{min} is the minimum utilization over all paths used by the TeXCP agent s.

$$\epsilon_s = .99 * \frac{C_{min}}{NP} \frac{(\overline{u}_s - u_s^{min})(1 - u_s^{min})}{R_s}. \tag{38}$$

Substituting this value of ϵ, and using $C_{min} \leq C_j$, $u_s^{min} \geq u_j$, $\overline{u}_s(n) \leq \overline{u}^M(n)$, we can bound the total increase due to perturbations:

$$\sum_s \sum_{p \in P_s, p \ni j} \frac{R_s \epsilon}{C_j} < (\overline{u}^M - u_j)(1 - u_j) \sum_s \sum_{p \in P_s, p \ni j} \frac{1}{NP}, \tag{39}$$

$$< (\overline{u}^M - u_j)(1 - u_j). \tag{40}$$

Plugging Equation 40 in Equation 37, and canceling the common terms, gives us

$$u_j(n+1) < \overline{u}^M(n). \tag{41}$$

Thus, the maximum utilization at step $n+1$ is smaller than $\overline{u}^M(n)$. Since, $\overline{u}^M(n)$ is a linear combination of link utilizations at step n, whereas $u_i(n)$ is the maximum utilization at step n, we have

$$\overline{u}^M(n) \leq u_i(n). \tag{42}$$

Merging Equations 41 and 42, we have

$$u_j(n+1) < \overline{u}^M(n) \leq u_i(n). \tag{43}$$

Case 3: $u_j(n) = \overline{u}^M(n)$ **[Intermediate Utilization]**
First, suppose that the traffic on link j reduces at this step, i.e., $u_j(n+1) < u_j(n) = \overline{u}^M(n)$. But, as we argued above, the linear combination over utilizations $\overline{u}^M(n)$ can be no larger than the maximum $u_i(n)$. Hence, if traffic reduces on link j, we have $u_j(n+1) < u_i(n)$ and are done.
No increase in traffic on paths traversing link j: Second, we will show that no TeXCP agent can increase its share of traffic on any path traversing link j. Note that for any TeXCP agent s, with path q traversing link j, $u_{sq}(n) \geq u_j(n) = \overline{u}^M(n) \geq \overline{u}_s(n)$, i.e., the utilization along the path q is larger than the average utilization at s. Hence, Equation 32 shows that the TeXCP agent will increase the share of path q by atmost ϵ and only if q is the minimum utilized path at agent s. But, if q does turn out to be the least utilized path at agent s, then the above inequality $u_{sq}(n) \geq \overline{u}_s(n)$ collapses to $u_{sq} = \overline{u}_s$, since the linear combination of utilizations $\overline{u}_s$ cannot be smaller than the minimum. Substituting this in Equation 38 results in $\epsilon_s = 0$. Hence, no TeXCP agent can add traffic on a path traversing link j.
Traffic on link j **doesn't change:** Third, the only option left is that the traffic on link j stays unchanged, i.e., $u_j(n+1) = u_j(n)$. Now, trivially, if link j was not the maximum utilized link at step n, i.e., $u_j(n) < u_i(n)$, then $u_j(n+1) < u_i(n)$ and we are done. This leads us to the final case, wherein link j was indeed the maximum utilized link at step n, i.e., $u_j(n+1) = u_j(n) = u_i(n)$, and link j's traffic remains unchanged. This brings us to this final sub-case.

Case 3.1: No Decrease in Max. Utilization in the network, $\overline{u}^M = u_j(n+1) = u_j(n) = u_i(n)$
We will show that this means the network has reached the stable state described in Theorem 3.4.2.

We classify the TeXCP agents as follows. Let set Z consist of TeXCP agents s that have $\overline{u}_s = \overline{u}^M$, and set $\overline{Z}$ consists of the remaining agents. Note, that all TeXCP agents that put traffic on link j are in Z. Now, we make two assertions.

First, no agent $s \in Z$ can know a path p that has a smaller utilization than $\overline{u}^M$. If it did, then it can move some fraction of its traffic onto this path using Equation 32 and reduce the maximum utilization—a contradiction of Case 3.1. Further, $\overline{u}_s = \overline{u}^M = \max_l u_l(n)$ implies that all the paths used by agent s traverse maximum utilization links, because a linear combination $\overline{u}_s$ can be equal to the maximum only if all the values are individually equal to the maximum.

Second, no agent $s \in \overline{Z}$ can be sending traffic on a path p that traverses a maximum utilized link. If it did, the path p would have utilization larger than the average at agent s, i.e., $u_{sp} = \overline{u}^M > \overline{u}_s$, and the agent would move traffic away from this path thereby reducing the traffic on the maximum utilized link—a contradiction of Case 3.1. Hence, agents in $\overline{Z}$ do not use any of the paths traversing maximum utilization links.

These assertions lead to an interesting conclusion. Since every path used by an agent in Z traverses a max-utilization link and no agent in $\overline{Z}$ uses a path containing a max-utilization link, the subsets of paths used by the agents in these two sets is disjoint. The agents in these two sets will not interact any more. Further, agents in Z have reached a balanced load and in accordance with Equation 32 will not move traffic any more. Thus, $u_j(n+1)$ will never decrease and this is the final max-utilization of the system.

We now focus on the agents in $\overline{Z}$. Since the link bandwidth consumed by agents in Z stays static after step n, we can ignore agents in Z after adjusting the link capacities accordingly. The new system has reduced capacity links and contains only the agents in set $\overline{Z}$. We can analyze this reduced system as we did the original system. The max-utilization of the reduced system will strictly decrease until we are again in sub-case 3.1. We are guaranteed to reach sub-case 3.1 as it is impossible for the max-utilization, a positive value, to keep decreasing indefinitely. At sub-case 3.1, a non-zero number of TeXCP agents (those in the new set Z_1) will satisfy the stability conditions in Theorem 3.4.2 and can be ignored henceforth. We repeat this argument iteratively until there are no TeXCP agents of type $\overline{Z}$ left.

To summarize, *eventually all TeXCP agents see equal utilization along all of the paths they use and do not know any paths with smaller utilization. In this state, no TeXCP agent moves traffic from one path to another, and the system has stabilized.*

F. Value of Preferring Shorter Paths in TeXCP

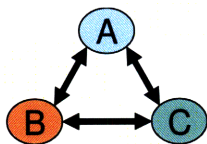


Figure -5 – Example Topology. Note that links are bi-directional.

From \ To	A	B	C
A	0	1	1
B	1	0	1
C	1	1	0

Figure -6 – Example Demand Matrix, each node sends one unit of traffic to every other node.

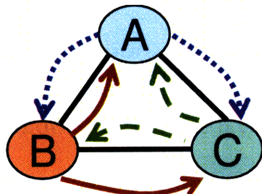


Figure -7 – Optimal Solution (and a Nash Equilibrium). Each node sends traffic directly to its neighbor on the shortest path. Each un-directional link here carries one unit of traffic. Further, no node can reduce the maximum utilization on the paths it uses by unilaterally moving traffic.

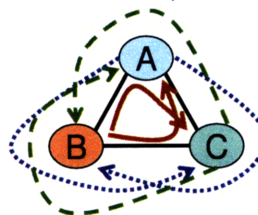


Figure -8 – Another Nash Equilibrium. Each node sends traffic indirectly on the longer path through the other neighbor. Note that in this case every uni-directional link carries two units of traffic. TeXCP agents at nodes, see that both their shortest direct path and the longer indirect path have the same utilization and hence do not move traffic.

We present a simple counter-example, wherein even when TeXCP converges, i.e., TeXCP agents no longer move any traffic, the traffic distribution does not minimize the maximum utilization in the network.

Figure -5 shows a simple topology with three nodes and six unidirectional links. Each node desires to send one unit of traffic to every other node in the network, as shown in Figure -6. Figure -7 shows the load distribution that minimizes the maximum utilization. Each node sends traffic intended for a neighbor directly on the edge connecting that neighbor. Each link, here, carries one unit of traffic. Figure -8 shows an alternate load distribution wherein each node sends traffic intended for a neighbor along a one hop detour through the other neighbor. Note, that each link here carries two units of traffic. Clearly, this is sub-optimal. Yet, the TeXCP agents at each node do not move traffic because their observed utilizations, on the shortest direct paths and the indirect paths, are equal!

Note that this does not happen in practice with TeXCP. Our analysis does not use the fact that TeXCP's feedback prefers shorter paths. Specifically, the feedback issued by a router at the head of each link prefers TeXCP agents that have the link on the shorter path. Hence, even if the network were to start with the load distribution shown in Figure -8, TeXCP would move the network to the distribution shown in Figure -7.

G. Propagating State for Meta-Nodes in Sherlock

Selector Meta-Node:

$$\text{P(child up)} = \sum_j d_j * p_j^{up} \tag{44}$$

$$\text{P(child troubled)} = \sum_j d_j * p_j^{troubled} \tag{45}$$

$$\text{P(child down)} = \sum_j d_j * p_j^{down} \tag{46}$$

Equations 44–46 show how to propagate state for the selector meta-node. The reasoning is simple. Intuitively, the state of the child, i.e., its probability distribution, is simply the weighted sum of its parents' state, where the weight for each parent is the probability with which the child selects this parent.

Failover Meta-Node: Algorithm 4 summarizes how to propagate the state for a failover meta-node. Recall that this meta-node encapsulates failover behavior. For example, given a ranked set of DNS servers (say primary, secondary, tertiary), looking up a name first attempts to connect to the primary server and when that doesn't work, attempts to connect with the secondary and so on. Our algorithm for state propagation follows from a simple recursive observation. Let us rank the parents according to their failover order, $1, \ldots, n$, i.e., with the primary parent ranked 1. Parents $1 \cdots n$ cause the child to be up whenever the primary parent is up, and when the primary parent is down whenever the remaining parents, i.e., ranks $2 \cdots n$ cause the child to be up. The same logic applies for the troubled state, except that failing over from an earlier parent to a later parent may inflate response time. Hence, upon failover, we deduct a small fraction β (defaults to 0.1) from the probability of being up and add it to the probability of being troubled. Finally, a child is down only when the current parent is down and all lower-ranked parents cause it to be down. Algorithm 4 unrolls this recursion by processing parent nodes in a certain order.

Algorithm 4 Failover Meta-Node: Computing Prob. of Child Node Given Parents

---

$\beta = .1$ ▷ Depreciation to account for Failover
Rank all n Parents such that Primary is 1, Secondary is 2 and so on.
Child State $(c^{up}, c^{troubled}, c^{down})$ = State of nth parent. ▷ Initialize to last parent
for Parent p in $n-1, \ldots, 1$ **do** ▷ For other parents, in the appropriate order
  $c^{up} = p^{up} + \left(p^{down} * \max(0, c^{up} - \beta)\right)$
  $c^{troubled} = p^{troubled} + p^{down} * \left(c^{troubled} + \min(c^{up}, \beta)\right)$
  $c^{down} = p^{down} * c^{down}$
end for

Bibliography

[1] CoralReef - Workload Characterization. http://www.caida.org/analysis/workload/.

[2] GT-ITM: Georgia Tech. Internetwork Topology Models. http://cc.gatech.edu/fac/Ellen.Zegura/gt-itm/gt-itm.tar.gz.

[3] IPMON. http://ipmon.sprintlabs.com.

[4] Configuring OSPF. http://www.cisco.com/univercd/cc/td/doc/product/software/ios122/.

[5] MOM–Microsoft Operations Manager. http://www.microsoft.com/mom/.

[6] NS–The Network Simulator. http://www.isi.edu/nsnam/ns.

[7] Cisco Express Forwarding (CEF). Cisco white paper, Cisco Systems., July 2002.

[8] JUNOS 6.3 Internet Software Routing Protocols Configuration Guide. http://www.juniper.net/techpubs/software/junos/junos63/swconfig63-routing/html/.

[9] Abilene. http://abilene.internet2.edu.

[10] Akamai. http://www.akamai.com.

[11] Collation/ IBM. http://www-01.ibm.com/software/tivoli/welcome/collation/.

[12] Private Communication with Microsoft Systems Center.

[13] Understanding Denial-of-Service Attacks. http://www.us-cert.gov/cas/tips/ST04-015.html.

[14] Estonia Computers Blitzed, Possibly by the Russians. http://www.nytimes.com/2007/05/19/world/europe/19russia.html?ref=world,.

[15] Estonia Under Attack. http://www.economist.com/displayStory.cfm?Story_ID=E1_JTGPVJR,.

[16] Cyberattack in Estonia–What it Really Means. http://news.cnet.com/Cyberattack-in-Estonia-what-it-really-means/2008-7349_3-6186751.html,.

[17] The Forrester Wave: Application Mapping For The CMDB. http://www.forrester.com/go?docid=36891, Feb. 2006.

[18] The Forrester Wave: Application Mapping And The CMDB, Q4 2006 Update. http://www.forrester.com/go?docid=40746, Nov. 2006.

[19] GAIM/Pidgin. http://www.pidgin.im/.

[20] Google GMail E-mail Hijack Technique. http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/.

[21] IDENT. http://www.grc.com/port_113.htm.

[22] Jcaptcha. http://jcaptcha.sourceforge.net/.

[23] Link-Level Multicast Name Resolution. http://www.windowsnetworking.com/articles_tutorials/Overview-Link-Local-Multicast-Name-Resolution.html.

[24] mathopd. http://www.mathopd.org.

[25] PortPeeker Capture of mySQL Bot attack. http://www.linklogger.com/mySQLAttack.htm.

[26] Nagios: Host, Service, Network Monitor. http://nagios.org.

[27] Netfilter/IPTables, . http://www.netfilter.org.

[28] NetQoS Superagent 2.0. http://www.networkworld.com/reviews/2002/0527rev.html, .

[29] nLayers/ EMC. http://www.crn.com/storage/188702420.

[30] Hewlett Packard Openview. http://www.openview.hp.com.

[31] Porn Gets Spammers Past Hotmail, Yahoo Barriers. CNet News, May 2004. http://news.com.com/2100-1023_3-5207290.html.

[32] Port 1081. http://isc.incidents.org/port.html?port=1081.

[33] Relicore Clarity/ Symantec. http://www.forrester.com/Research/Document/Excerpt/0,7211,38951,00.html.

[34] Rocketfuel. www.cs.washington.edu/research/networking/rocketfuel.

[35] SAINT – The Security Administrator's Integrated Network Tool. http://www.wwdsi.com/saint.

[36] Analysis of the Sapphire Worm. http://www.caida.org/analysis/security/sapphire/.

[37] The Slashdot Effect. http://www.urbandictionary.com/define.php?term=slashdot+effect.

[38] IBM Tivoli. http://www.ibm.com/software/tivoli.

[39] WebStone–The Benchmark for Web Servers. http://www.mindcraft.com/webstone/.

[40] Wells Fargo Recovering from Computer Crash. http://www.infoworld.com/article/07/08/21/Wells-Fargo-recovering-from-computer-crash_1.html, .

[41] Wells Fargo Computer Network Buckles. http://www.sfgate.com/cgi-bin/article.cgi?file=/chronicle/archive/2000/12/02/BU120769.DTL, .

[42] Wells Fargo Bank System Failure – Invitation to Phishers? http://blog.ironsortkey.com/?p=181, .

[43] WinPCAP. http://www.winpcap.org/.

[44] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP*, 2003.

[45] A. Akella, B. Maggs, S. Seshan, A. Shaikh, , and R. Sitaraman. A Measurement-Based Analysis of Multihoming. In *SIGCOMM*, 2003.

[46] A. Akella, S. Seshan, and A. Shaikh. An Empirical Evaluation of Wide-Area Internet Bottlenecks. In *IMC*, 2003.

[47] D. Andersen. Mayday: Distributed Filtering for Internet services. In *USITS*, 2003.

[48] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[49] E. J. Anderson and T. E. Anderson. On the Stability of Adaptive Routing in the Presence of Congestion Control. In *INFOCOM*, 2003.

[50] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *HotNets*, 2003.

[51] D. Applegate and E. Cohen. Making Intra-Domain Routing Robust to Changing and Uncertain Traffic Demands. In *SIGCOMM*, 2003.

[52] D. Applegate, L. Breslau, and E. Cohen. Coping with Network Failures: Routing Strategies for Optimal Demand Oblivious Restoration. In *SIGMETRICS*, 2004.

[53] D. Awduche, L. Berger, D. Gan, T. Li, V. Srinivasan, and G. Swallow. RSVP-TE: Extensions to RSVP for LSP Tunnels, 2001. IETF RFC 3209.

[54] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[55] H. Balakrishnan, N. Dukkipati, N. McKeown, and C. Tomlin. Stability Analysis of Explicit Congestion Control Protocols. Technical Report SUDAAR #776, Stanford Aero-Astro Dept., 2008.

[56] G. Banga, P. Druschel, and J. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI*, 1999.

[57] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[58] A. Basu and J. G. Reicke. Stability Issues in OSPF Routing. In *SIGCOMM*, 2001.

[59] D. Bertsekas and R. Gallager. *Data Networks*. Englewood Cliffs, 1992.

[60] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. In *Allerton*, 2002.

[61] Bryan Parno and Dan Wendlandt and Elaine Shi and Adrian Perrig and Bruce Maggs and Yih-Chun Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *SIGCOMM*, 2007.

[62] J. E. Burns, T. J. Ott, A. E. Krzesinski, and K. E. Muller. Path Selection and Bandwidth Allocation in MPLS Networks. *Perform. Eval*, 2003.

[63] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and K. van der Merwe. Design and implementation of a routing control platform. In *NSDI*, 2005.

[64] J. Cao, D. Davis, S. Wiel, and B. Yu. Time-varying network tomography : Router link data. Bell labs tech. memo, Bell Labs, Feb. 2000.

[65] N. Carr. Crash: Amazon's S3 Utility Goes Down. http://www.roughtype.com/archives/2008/02/amazons_s3_util.php, Feb. 2008.

[66] CERT. Incident Note IN-2004-01 W32/Novarg.A Virus, 2004.

[67] CERT. Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, 1998. http://www.cert.org/advisories/CA-1998-01.html.

[68] CERT. Advisory CA-2003-20 W32/Blaster worm, 2003.

[69] C. Chambers, J. Dolske, and J. Iyer. TCP/IP Security. http://www.linuxsecurity.com/resource_files/documentation/tcpip-security.html.

[70] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *NSDI'04*, Mar. 2004.

[71] A. Coates, H. Baird, and R. Fateman. Pessimal print: A Reverse Turing Test. In *IAPR*, 1999.

[72] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security*, 2004.

[73] C. Dixon, T. Anderson, and A. Krishnamurthy. Phalanx: Withstanding Multimillion-Node Botnets. In *NSDI*, 2008.

[74] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. John Wiley, 2002.

[75] J. Dunagan, N. Harvey, M. Jones, D. Kostic, M. Theimer, and A. Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*, 2004.

[76] C. Dwork and M. Naor. Pricing via Processing or Combatting Junk Mail. In *Crypto 92, Springer-Verlag Lecture Notes in Computer Science*, 1992.

[77] K. Egevang and P. Francis. The IP Network Address Translator (NAT). In *RFC 1631*, 1994.

[78] A. Elwalid, C. Jin, S. H. Low, and I. Widjaja. MATE: MPLS Adaptive Traffic Engineering. In *INFOCOM*, 2001.

[79] C. Estan, S. Savage, and G. Varghese. Automatically Inferring Patterns of Resource Consumption in Network Traffic. In *SIGCOMM*, 2003.

[80] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and K. van der Merwe. The Case for Separating Routing from Routers. In *SIGCOMM FDNA*, 2004.

[81] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford. Deriving Traffic Demands from Operational IP Networks: Methodology and Experience. *IEEE/ACM Transaction on Networking*, 2001.

[82] W.-C. Feng, E. Kaiser, W.-C. Feng, and A. Lu. The Design and Implementation of Network Puzzles. In *INFOCOM*, 2005.

[83] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616 - Hypertext Transfer Protocol - HTTP/1.1.

[84] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1993.

[85] B. Fortz and M. Thorup. Optimizing OSPF Weights in a Changing World. In *IEEE JSAC*, 2002.

[86] B. Fortz and M. Thorup. Robust Optimization of OSPF/IS-IS Weights. In *INOC*, 2003.

[87] N. Friedman and D. Koller. Being Bayesian about Bayesian Network Structure:A Bayesian Approach to Structure Discovery in Bayesian Networks. In *UAI*, 2000.

[88] D. Funderburg and J. Tirole. *Game Theory*. The MIT Press, 1991.

[89] R. Gallager. A Minimum Delay Routing Algorithm using Distributed Computation. *IEEE Transactions on Computers*, 1977.

[90] T. Gil and M. Poletto. MULTOPS: A Data-Structure for Bandwidth Attack Detection. In *USENIX Security*, 2001.

[91] R. Govindan and V. Paxson. Estimating Router ICMP Generation Times. In *PAM*, Munich, 2002.

[92] J. Guichard, F. le Faucheur, and J. P. Vasseur. *Definitive MPLS Network Designs*. Cisco Press, 2005.

[93] E. Hellweg. When Bot Nets Attack. *MIT Technology Review*, September 2004.

[94] R. Iyer, V. Tewari, and K. Kant. Overload Control Mechanisms for Web Servers. In *Performance and QoS of Next Gen. Networks Workshop*, 2000.

[95] H. Jamjoom and K. G. Shin. Persistent Dropping: An Efficient Control of Traffic. In *SIGCOMM*, 2003.

[96] Jose Bernardo and Adrian F. M. Smith. *Bayesian Theory*. John Wiley, 2000.

[97] A. Juels and J. G. Brainard. Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. In *NDSS*, 1999.

[98] J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *WWW*, 2002.

[99] S. Kandula, D. Katabi, B. Davie, and A. Charny. Walking the Tightrope: Responsive Yet Stable Traffic Engineering. In *SIGCOMM*, 2005.

[100] S. Kandula, D. Katabi, M. Jacob, and A. W. Berger. Botz-4-Sale: Surviving Organized DDoS Attacks That Mimic Flash Crowds. In *NSDI*, 2005.

[101] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A Tool for Failure Diagnosis in IP Networks. In *SIGCOMM MineNet Workshop*, 2005.

[102] S. Kandula, R. Chandra, and D. Katabi. What's Going On? Learning Communication Rules in Edge Networks. In *SIGCOMM*, 2008.

[103] J. Kannan, J. Jung, V. Paxson, and C. E. Koksal. Semi- Automated Discovery of Application Session Structure. In *IMC*, 2006.

[104] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the dark. In *SIGCOMM*, 2005.

[105] S. Karp. Gmail Slow and Still Working : Enough Is Enough! http://publishing2.com/2008/01/04/gmail-slow-and-still-working-enough-is-enough/, Jan. 2008.

[106] D. Katabi and C. Blake. Inferring Congestion Sharing and Path Characteristics from Packet Interarrival Times. Technical Report MIT-LCS-TR-828, MIT, 2001.

[107] D. Katabi, M. Handley, and C. Rohrs. Internet Congestion Control for High Bandwidth-Delay Product Networks. In *SIGCOMM*, 2002.

[108] R. Keralapura, N. Taft, C.-N. Chuah, and G. Iannaccone. Can ISPs take the Heat from Overlay Networks? In *HOTNETS*, 2004.

[109] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, 2002.

[110] A. Khanna and J. Zinky. The Revised ARPANET Routing Metric. In *SIGCOMM*, 1989.

[111] G. Kochanski, D. Lopresti, and C. Shih. A Reverse Turing Test using speech. In *ICSLP*, 2002.

[112] M. Kodialam, T. V. Lakshman, and S. Sengupta. Efficient and Robust Routing of Highly Variable Traffic. In *HOTNETS*, 2004.

[113] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[114] R. Kompella, J. Yates, A. Greenberg, and A. Snoeren. IP Fault Localization Via Risk Modeling. In *NSDI*, May 2005.

[115] M. Krigsman. Amazon S3 Web Services Down. Bad, bad news for Customers. http://blogs.zdnet.com/projectfailures/?p=602, Feb. 2008.

[116] S. Kunniyur and R. Srikant. Analysis and Design of an Adaptive Virtual Queue. In *SIGCOMM*, 2001.

[117] J. Leyden. East European Gangs in Online Protection Racket, 2003. http://www.theregister.co.uk/2003/11/12/east_european_gangs_in_online/.

[118] J. Leyden. The Illicit Trade in Compromised PCs, 2004. http://www.theregister.co.uk/2004/04/30/spam_biz/.

[119] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *CCR*, 2002.

[120] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User Level Internet Path Diagnosis. In *SOSP*, Oct. 2003.

[121] D. Mazieres. Toolkit for User-Level File Systems. In *USENIX*, 2001.

[122] S. McCanne. The Berkeley Packet Filter Man page, May 1991.

[123] A. Medina, N. Taft, K. Salamatian, S. Bhattacharayya, and C. Diot. Traffic Matrix Estimation: Existing Techniques and New Directions. In *SIGCOMM*, 2002.

[124] D. Mitra and K. G. Ramakrishna. A Case Study of Multiservice Multipriority Traffic Engineering Design. In *GLOBECOM*, 1999.

[125] J. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. In *USENIX Tech. Conf.*, 1996.

[126] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *USENIX Security*, 2001.

[127] W. G. Morein et al. Using Graphic Turing Tests to Counter Automated DDoS Attacks. In *ACM CCS*, 2003.

[128] G. Mori and J. Malik. Recognizing Objects in Adversarial Clutter: Breaking a Visual CAPTCHA. In *CVPR*, 2003.

[129] K. P. Murphy, Y. Weiss, and M. I. Jordan. Loopy Belief Propagation for Approximate Inference: An Empirical Study. In *Uncertainity in Artificial Intelligence*, 1999.

[130] T. Oetiker and D. Rand. Multi Router Traffic Grapher. http://people.ee.ethz.ch/~oetiker/webtools/mrtg.

[131] E. Osborne and A. Simha. *Traffic Engineering with MPLS*. Cisco Press, 2002.

[132] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A First Look at Modern Enterprise Traffic. In *IMC*, 2005.

[133] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 1999.

[134] V. Paxson. An Analysis of Using Reflectors for Distributed Denial-of-Service Attacks. *ACM CCR*, 2001.

[135] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[136] D. Plonka. Flowscan: A Network Traffic Flow Reporting and Visualization Tool. In *USENIX System Admin. Conf.*, 2000.

[137] K. Poulsen. FBI Busts Alleged DDoS Mafia, 2004. http://www.securityfocus.com/news/9411.

[138] E. Protalinski. Users Complain New Gmail Version Slow, Crashes Browsers. http://www.neowin.net/news/main/07/11/17/users-complain-new-gmail-version-slow-crashes-browsers, Nov. 2007.

[139] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On Selfish Routing in Internet-Like Environments. In *SIGCOMM*, 2003.

[140] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *SIGCOMM*, 2006.

[141] rextilleon. Gmail So Slow. http://www.blackberryforums.com/general-blackberry-discussion/130313-gmail-so-slow.html, May 2008.

[142] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting The Unexpected in Distributed Systems. In *NSDI*, 2006.

[143] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-box Performance Debugging for Wide-area Systems. In *WWW*, May 2006.

[144] L. Ricciulli, P. Lincoln, and P. Kakkar. TCP SYN Flooding Defense. In *CNDS*, 1999.

[145] I. Rish, M. Brodie, and S. Ma. Efficient Fault Diagnosis Using Probing. In *AAAI Spring Symposium on Information Refinement and Revision for Decision Making*, March 2002.

[146] E. Rosen, A. Viswanathan, and R. Callon. Multi-protocol Label Switching Architecture. RFC 3031.

[147] M. Roughan, A. Greenberg, C. Kalmanek, M. Rumsewicz, J. Yates, and Y. Zhang. Experience in Measuring Backbone Traffic Variability: Models, Metrics, Measurements and Meaning. In *IMW*, 2002.

[148] Y. Rui and Z. Liu. ARTiFACIAL: Automated Reverse Turing Test Using FACIAL Features. In *Multimedia*, 2003.

[149] R. Russell. Linux IP Chains-HOWTO. http://people.netfilter.org/~rusty/ipchains/HOWTO.html.

[150] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. In *ACM CCR*, 1999.

[151] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *SIGCOMM*, 2000.

[152] E. Schonfeld. Amazon Web Services Goes Down, Takes Many Startup Sites With It. http://www.techcrunch.com/2008/02/15/amazon-web-services-goes-down-takes-many-startup-sites-with-it/, Feb. 2008.

[153] E. Schonfeld. Amazon Web Services Gets Another Hiccup. http://www.techcrunch.com/2008/04/07/amazon-web-services-gets-another-hiccup/, Apr. 2008.

[154] seanwuzhere. Slow Gmail! http://www.youtube.com/watch?v=8ST-KfsrT6g, Dec. 2007.

[155] A. Shaikh, J. Rexford, and K. Shin. Load-Sensitive Routing of Long-lived IP Flows. In *SIGCOMM*, 1999.

[156] S. Sinha, S. Kandula, and D. Katabi. Harnessing TCPs Burstiness using Flowlet Switching. In *HOTNETS*, 2004.

[157] S. Skogestad and I. Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons, 1996.

[158] P. Smyth and R. M. Goodman. *Knowledge Discovery in Databases.* MIT Press, 1991.

[159] A. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-Based IP Traceback. In *SIGCOMM*, 2001.

[160] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving Accuracy in End-to-end Packet Loss Measurement. In *SIGCOMM*, 2005.

[161] D. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *INFOCOM*, 2001.

[162] A. Sridharan, S. B. Moon, and C. Diot. On the Correlation between Route Dynamics and Routing Loops. In *IMC*, 2003.

[163] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer Caching Schemes to Address Flash Crowds. In *IPTPS*, 2002.

[164] S. Staniford, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS: A Graph-based Intrusion Detection System for Large Networks. In *National Information Systems Security Conference*, 1996.

[165] S. Staniford, V. Paxson, and N. Weaver. How to own the Internet in Your Spare Time. In *USENIX Security*, 2002.

[166] A. Stavrou, D. Rubenstein, and S. Sahu. A Lightweight Robust P2P System to Handle Flash Crowds. *IEEE JSAC*, 2004.

[167] L. Taylor. Botnets and Botherds. http://sfbay-infragard.org.

[168] R. Teixeira, K. Marzullo, S. Savage, and G. Voelker. In Search of Path Diversity in ISP Networks. In *IMC*, 2003.

[169] J. Tsitsiklis and D. Bertsekas. Distributed Asynchronous Optimal Routing in Data Networks. *IEEE Trans. on Automatic Control*, 1986.

[170] J.-P. Vasseur, M. Pickavet, and P. Demeester. *Network Recovery: Protection and Restoration of Optical SONET-SDH, IP, and MPLS*. Morgan-Kauffmann, 2004.

[171] S. Vempala, R. Kannan, and A. Vetta. On Clusterings Good, Bad and Spectral. In *FOCS*, 2000.

[172] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Fast Detection of Superspreaders. In *NDSS*, 2005.

[173] C. Villamazir. OSPF Optimized Multipath (OSPF-OMP), 1999. Internet Draft.

[174] C. Villamazir. MPLS Optimized Multipath (MPLS-OMP), 1999. Internet Draft.

[175] T. Voigt and P. Gunningberg. Handling Multiple Bottlenecks in Web Servers Using Adaptive Inbound Controls. In *High-Speed Networks*, 2002.

[176] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *EUROCRYPT*, 2003.

[177] S. Vutukury and J. J. Garcia-Luna-Aceves. A Simple Approximation to Minimum-Delay Routing. In *SIGCOMM*, 1999.

[178] M. Walfish. *Defending Networked Resources Against Floods of Unwelcome Requests*. PhD thesis, Massachusetts Institute of Technology, Feb. 2008.

[179] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg. COPE: Traffic Engineering in Dynamic Networks. In *ACM SIGCOMM*, 2006.

[180] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *SOSP*, 2001.

[181] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend against DDoS Attacks. In *IEEE Security & Privacy*, 2003.

[182] H. Yan, D. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: A 4D Network Control Plane. In *NSDI*, 2007.

[183] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting Network Architecture. In *SIGCOMM*, 2005.

[184] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High Speed and Robust Event Correlation. In *IEEE Communications Magazine*, 1996.

[185] K. Yoda and H. Etoh. Finding a Connection Chain for Tracing Intruders. In *ESORICS*, 2000.

[186] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *USENIX Security*, 2000.

[187] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An Information-Theoretic Approach to Traffic Matrix Estimation. In *SIGCOMM*, 2003.

[188] R. Zhang-Shen and N. McKeown. Designing a Predictable Internet Backbone Network. In *HotNets*, 2004.