ABSTRACT

Title of Dissertation: Discovering and Securing Shared Resources on the Internet

Robert Sherwood, Doctor of Philosophy, 2008

Dissertation directed by: Associate Professor Samrat Bhattacharjee and
Assistant Professor Neil Spring
Department of Computer Science

The Internet is a collection of shared resources. Internet users share bandwidth and processing resources both in the network at routers and on the network's edge at servers. However, the Internet's architecture does not prevent nodes from consuming disproportionate resources. In practice, resource exhaustion does occur due to inefficiently scaling systems, selfish resource consumption, and malicious attack. The current Internet architecture has limited support for both securing and identifying shared Internet resources.

This dissertation has two main contributions. First, I demonstrate the existence of end-host protocols that protect the availability of shared Internet resources. I consider resource sharing with respect to cooperative, selfish, and malicious user models, and for each case design a protocol that protects resource availability without modifying

the existing Internet infrastructure. Second, I design and validate measurement techniques for discovering shared Internet resources including links and routers. Specifically, I improve the completeness and accuracy of resource maps by combining embedding probes, disjunctive logic programming, and information from the record route IP option. We validate and quantify the improvement of our maps by comparison to publicly available research networks.

Discovering and Securing Shared Resources on the Internet

by

Robert Sherwood

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:

Associate Professor Samrat Bhattacharjee and
Assistant Professor Neil Spring, Co-Chairs/Co-Advisors
Associate Professor Pete Keleher
Associate Professor Francois Guimbretiere
Professor Mark Shayman

ACKNOWLEDGEMENTS

I acknowledge and thank my co-advisors, Bobby Bhattacharjee and Neil Spring, for their support and extreme patience. While both formidibable mentors individually, their combined efforts are more than the sum of the parts, much to my significant and grateful benefit.

From my measurement work, there are many people I would like to thank. I owe Mark Huang and the rest of the PlanetLab staff for their help and patience in the early phases of debugging Sidecar and its tools. I would also like to thank Vivek Pai and the rest of the CoDeeN project for allowing us access to their slice, and Fritz McCall and the UMIACS staff for their timely assistance. Also, thanks to the systems administrators at University of British Columbia and Vrije University of Amsterdam for their help in mapping unexplained routing behavior back to their manufacturer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

The Internet is a collection of shared resources. Users share bandwidth on links, processing on routers, and storage and computation at end-points. However, the Internet's architecture does not prevent nodes from consuming disproportionate resources. In practice, resource exhaustion does occur due to inefficiently scaling systems, selfish resource consumption, and malicious attack. Resource exhaustion results poor performance and negative user experience. Given the Internet's increasing importance in our daily lives, what can be done to ensure that these resources remain available?

Completely redesigning the protocols and network to improve resource usage is likely to prove impractical in terms of cost and coordination. The cost of individual routers range from tens to hundreds of thousands of dollars [150], and my work estimates (Chapter 7) that there are approximately one hundred thousand routers deployed on the Internet. Thus, any solution that could not leverage the existing routing hardware would cost billions of dollars to implement. A compounding problem is coordination. The Internet is world-wide production system, and any deployment of a redesigned network or protocols must globally coordinated and downtime kept to a minimum. Given the expense in modifying network hardware and the difficulty in coordinating global protocol update, can resource sharing be improved without replacing

existing protocols or modifying the network hardware?

Answering these resource sharing questions is complicated by the absence of a complete and accurate router-level map of the Internet. Without this map, researcher's understanding of the distribution and usage patterns of network resources is limited, i.e., they do not know where the routers and links are or how they are shared. Obtaining this map is difficult. Existing Internet protocols were not explicitly designed for measurement Further, Internet Service Providers (ISPs) each know their local networks maps, but are unwilling to publish them for fear of loss of competitive advantage. Researchers use active measurements and observations to infer the network map, but the Internet's size, hardware diversity, and ISP policy make these maps incomplete and inaccurate. However, active measurements are limited by administrators who confuse measurement probes for attacks and generate "abuse reports" and even legal threats that curtail the scope of experiments. What can be done to improve the completeness and accuracy of the map of the Internet without causing the ire of network administrators?

My thesis is that is is possible to *secure and discover shared Internet resources without global protocol redeployment or architectural support*. In support of this thesis, my dissertation has two main contributions:

1. I design and evaluate protocols that secure the Internet's shared resources without requiring network support or global protocol redeployment. Because the notion of security varies with respect to the attack model, I demonstrate security in three distinct user assumptions: cooperative, selfish, and malicious users (as defined below).

2. I design and evaluate new techniques for increasing the accuracy and completeness of Internet topology discovery. These techniques leverage existing protocol

and hardware features, and thus can be implemented on today's Internet.

1.1 Securing Resource Availability

Techniques for ensuring the availability of shared resources vary based on the assumed behavior of users in the system. In this dissertation, I consider three user models—cooperative, selfish, and malicious—where each model has an orthogonal set of problem and solutions. For each model, I describe the user behavior, the set of problems associated with that behavior, an example of resource exhaustion, and summarize the dissertation chapter that addresses the problem.

- **Cooperative** users faithfully follow prescribed protocols. Despite following the "correct" behavior, cooperative users can cause exhaustion when resources are not shared *efficiently*. Flash crowds, where many users suddenly access the same website, i.e., the Slashdot effect, are an example of inefficiently shared resources exhausted from cooperative users.

  Chapter 3 describes Slurpie, a bulk data transfer protocol that mitigates the effects of flash crowds on web (HTTP) and file transfer (FTP) servers while reducing the average download time for users. Similar to BitTorrent [29], Slurpie users download random subsections ("blocks") of the file from the primary server, and trade among other users to obtain a complete copy of the file. Slurpie works in parallel with existing data transfer protocols, e.g., HTTP and FTP, and thus does not require content providers to change or other global protocol redeployment.

- **Selfish** users deviate arbitrarily from prescribed protocols if they can increase their personal benefit. Selfish users causes exhaustion when resources are not shared *fairly* at the expense of cooperative users. Peers in a peer-to-peer file

sharing protocol, like BitTorrent, that cheat the protocol and share less then prescribed are an example of selfish behavior.

Chapter 4 presents, NICE, a reputation-based trust and abstract resource trading system. In NICE, users *interact*, and exchange signed certificates ("cookies") that testify to the quality of the interaction. Over time, users can transitively learn to avoid selfish users that try to cheat them out of their resources. This system works on top of existing protocols and hardware, so does not require global redeployment.

- **Malicious** users deviate arbitrarily from prescribed protocols expressly to exhaust shared resources. Malicious users exhaust resources that are not shared *securely* to the detriment of other users in the system. Armies of compromised machines (BotNets) mounting a distributed denial-of-service attack, e.g., saturating a link such that it is unusable, is an example of malicious behavior.

  Chapter 2 describes an amplification denial-of-service attack on TCP. In the attack, a malicious TCP receiver sends acknowledgments *optimistically* for segments that it has not received. As a result, the unwitting TCP sender is fooled into flooding the network. Run against many TCP senders (victims) in parallel, the attack becomes a distributed denial-of-service attack, capable of amplifying the attacker's bandwidth thousands or millions of times. I then describe an solution, randomly skipped segments, that solves the problem without requiring network support or global redeployment of TCP.

1.2 Improving Network Maps

To better understand shared resource usage, my dissertation improves the state-of-the-art network maps using three complementary techniques: TCP Sidecar, Passenger, and DisCarte. Each of these techniques leverages existing protocol and hardware features, and thus do not require additional network support or protocol redeployment.

- TCP Sidecar (Chapter 5) embeds measurement probes into third-party TCP streams, allowing probes to traverse NATs and firewalls without alerting intrusion detection systems (IDS). TCP Sidecar improves the completeness of network mapping by allowing probes into more of the network than firewalls, NATs, or local security policy might otherwise allow. Sidecar probes work without additional network, sender, or receiver support.

- Passenger (Chapter 6) makes use of the often ignored Record Router (RR) IP option to expose mid-measurement route changes and discover IP aliases for unresponsive routers. The information gleaned from RR also exposes vendor-specific router implementation behavior, making it possible to identify router manufacturer. However, the same implementation varies complicated data analysis. Using RR improves improves the accuracy by removing error caused by mid-measurement route changes and the completeness of Internet maps by discovering more IP aliases. The RR IP option is an existing part of the IP specification, and thus this technique works with existing Internet hardware.

- DisCarte (Chapter 7) is a constraint solving system that combines three topology discovery data sources—traceroute, record router, and observed network engineering practices—into a single, unified self-consistent topology. DisCarte formulates the unification process as a constraint solving problem using disjunctive

logic programming. Because disjunctive logic programming is exponential in the number of inputs, DisCarte uses a divide-and-conquer scheme to first solve overlapping subsets of the topology and then merge them back together. DisCarte improves the accuracy of network topology discovery by producing the map with least violations of observed engineering practices. Because DisCarte works completely offline, it requires no additional network or protocol support.

Chapter 2

Shared Resources with Malicious Users: OptAck

2.1 Introduction

Savage et al. [127] present three techniques by which a misbehaving TCP receiver can manipulate the sender into providing better service at the cost of fairness to other nodes. With one such technique, optimistic acknowledgment ("opt-ack"), the receiver deceives the sender by sending acknowledgments (ACKs) for data segments before they have actually been received. In effect, the connection's round trip time is reduced and the total throughput increased. Savage et al. observe that a misbehaving receiver could use opt-ack to conceal data losses, thus improving end-to-end performance at the cost of data integrity. They further suggest that opt-ack could potentially be used for denial of service, but do not investigate this further.

Here, we consider a receiver whose *sole interest* is exploiting opt-ack to mount a distributed denial of service (DoS) attack against not just individual machines, but *entire networks*. Specifically, we:

1. Demonstrate a previously unrealized and significant danger from the opt-ack attack (one attacker, many victims) through analysis (Section 2.3) and both simulated and real world experiments.

2. Survey prevention techniques and present a novel, efficient, and *incrementally deployable* solution (Section 2.4.2) based on skipped segments, whereas previous solutions ignored practical deployment concerns.

3. Argue that the distributed opt-ack attack (many attackers, many victims) has potential to bring about sustained congestion collapse across large sections of the Internet, thus necessitating immediate action.

2.1.1 An Attack Based on Positive Feedback

Two significant components of transport protocols are the flow and congestion control algorithms. These algorithms, by necessity, rely on remote feedback to determine the rate at which packets should be sent. This feedback can come directly from the network [50, 89] or, more typically, from end hosts in the form of positive or negative acknowledgments. These algorithms implicitly assume that the remote entity generates correct feedback. This is typically a safe assumption because incorrect feedback rapidly deteriorates end-to-end performance [71]. *However, an attacker who does not care about data integrity could violate this assumption to induce the sender into injecting many packets into the network.* While not all of these packets may arrive at the receiver, they do serve to congest the sender's network and saturate the path from the sender to the receiver.

In this chapter, we always assume that the attacker targets multiple victims, in order to maximize the damage that the attack can cause Because acknowledgment packets are relatively small (40 bytes), it is trivial for an attacker to target hundreds and even thousands of victims in parallel. In effect, not only are each victims' access links saturated, but, due to over-provisioning, higher bandwidth links in the upstream ISPs begin to suffer congestion collapse in aggregate as well. In Section 2.2.4, we argue that

sufficiently many attackers can overwhelm backbone links in the core of the Internet, causing wide-area sustained congestion collapse.

2.1.2 Road map

The rest of the chapter is structured as follows. Section 2.2 describes attack pseudo-code, implementation challenges, variants, and the distributed opt-ack attack. Section 2.3 discusses various bounds on the attacker's bandwidth amplification. In Section 2.4, we consider and evaluate possible solutions, propose one based on skipped segments, and describe its implementation. In Section 2.5, we present performance numbers of attacked machines with and without the proposed fix, in real world and simulated topologies. Next, we discuss related work in Section 2.6. We conclude with implications of the opt-ack attack and future work in Section 2.7. Section 2.8 describes the key observations required in a practical implementation of the opt-ack attack.

2.2 Attack Analysis

In this section we describe pseudo-code for the attack, a summary of implementation challenges, attack variants, and the details of the distributed version of the opt-ack attack. In Section 2.8, we present the observations we made in implementing the attack and techniques for mitigating practical concerns.

2.2.1 The Opt-Ack Attack

Algorithm 1 shows how a single attacker can target many victims at once. Typically, the attacker would employ a compromised machine (a "zombie" [144]) rather than

launch the attack directly.[1] Consider a set of victims, $v_1 \ldots v_n$, that serve files of various sizes. The attack connects to each victim, then sends an application level request, e.g., an HTTP GET. The attacker then starts to acknowledge data segments *regardless of whether they arrived or not* (Figure 2.1). This causes the victim to saturate its local links by responding faster and faster to the attackers opt-acks. To sustain the attack, the attacker repeatedly asks for the same files or iterates through a number of files.

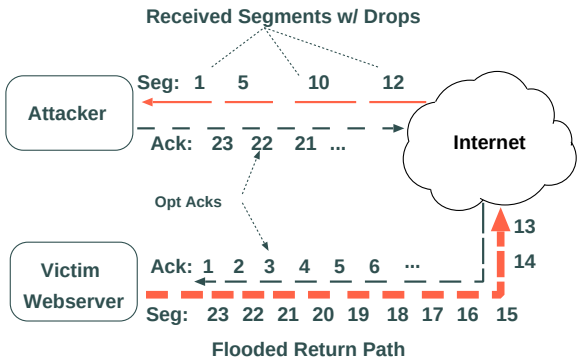


Figure 2.1: Opt-Ack Attack: Single Victim w/ Packet Loss (One of many victims)

The crux of the attack is that the attacker must produce a seemingly valid sequence of ACKs. For an ACK to be considered valid, it must not arrive before the victim has sent the corresponding packet. Thus, the attacker must estimate which packets are sent and when, based only on the stream of ACKs the attacker has already sent. At first this might seem a difficult challenge, but the victim's behavior on receiving an ACK is exactly prescribed by the TCP congestion control algorithm! The attack takes three parameters: a list of n victims, the maximum segment size (mss), and the window

[1]This attack can also be mounted if the attacker is able to spoof TCP connections, either by being on the path between the victim and the spoofed address, or from guessing the initial sequence number, but we do not further consider it.

scaling ($wscale$) factor. In the algorithm, the attacker keeps track of each victim's estimated window (w_i) and sequence number to acknowledge (ack$_i$). The upper bound of w_i, $maxwindow$, is 65535 by default, but can be changed by the window scaling option (see Section 2.3). Note that the attacker can manipulate each victim's retransmission time out (RTO), because the RTO is a function of the round trip time, which is calculated by the ACK arrival rate. So, in other words, the attack can completely manipulate the victims in terms of how fast to send, how much to send, and when to time out.

There is a near arbitrary number of potential victims, given the pervasiveness of large files on the Internet. Any machine that is capable of streaming TCP data is a potential victim, including HTTP servers, FTP servers, content distribution networks (CDN), P2P file sharing peers (KaZaa[79], Gnutella[60]), NNTP servers, or even machines with the once common character generator ('chargen') service.

The attack stream is difficult to distinguish from legitimate traffic. To an external observer that is sufficiently close to the victim, such as a network intrusion detection system (IDS), this stream is in theory indistinguishable from a completely valid high speed connection.[2] While it is common for IDSs to send out alerts if a large stream of packets enters the local network, the stream of ACKs from the attacker is comparatively small (see Section 2.3 for exact numbers). It is the stream of data *leaving* the network that is the problem.

Additionally, an attacker can further obscure the attack signature by sending acknowledgments to more victims less often, with the total amount of traffic generated staying constant. In other words, by generating less traffic per host and staying under

[2]Presumably, a monitoring system deployed closer to the attacker could detect the asynchrony between ACKs and data segments, but it is not practical to store per-flow state deep in the network.

the detection threshold, but increasing the total number of hosts *it is not locally obvious to the victims that they are participating in an DDoS attack*. As a result, short of a globally coordination, potentially through a distributed intrusion detection system, it is difficult for victims to locally determine if a given stream is malicious.

While Algorithm 1 works in theory, there are still challenges for the adversary to keep ACKs synchronized with the segments the victims actually send. We address these issues in the next section.

2.2.2 Implementation Challenges

The main challenge in implementing the attack is to accurately predict which segments the victim is sending and ensure that the corresponding ACKs arrive at the correct time. In Figure 2.1, the attacker injects ACKs into the network before the corresponding segments have even reached the attacker, so remaining synchronized with the victim can be non-trivial. Maintaining this synchronization of sequence numbers is crucial to the attack. If the attacker falls behind, i.e., it starts to acknowledge segments slower than they are sent, then the victim slows down, may time out, and the effect of the attack is reduced. Similarly, if the attacker gets ahead of the victim in the sequence space, i.e., the victim received ACKs for segments that are not yet sent, the victim ignores these ACKs and the stream stops making progress. We refer to this condition as *overrunning* the victim. Overruns can occur in three different ways: ACKs arriving too quickly, lost ACKs, and delays at the server. However, if an attacker does overrun the server, it is possible for the attacker to detect this condition and recover (Section 2.8).

In accordance with RFC793 [121], Section 3.4, when the sender receives ACKs that are not in the window, it should not generate a RST, but instead an empty packet

with the correct sequence number. One of the tenets of the Internet design philosophy is the robustness principle: "be conservative in what you send, and liberal in what you accept," and it is this principle that opt-ack exploits.

There are many ways that an overrun condition may result, most common being the sending application stalls its output because it was preempted by another process. In general, there are a myriad of factors that affect the sender's actual output rate, including: the victim's load, application delay, the victim's send buffer size, and the victim's hardware buffer. However, these factors are mitigated when the number of victims is large. By sending ACKs to more victims, each individual victim receives ACKs less often. This provides more time for the victim to flush its buffers, place the sending application back into the run queue, etc.

It is worth noting that the implementation we developed is only a demonstration of the potential severity of opt-ack. It is by no means an optimal attack. There are a number of points where a more thorough attacker might be able to mount a more efficient attack. However, as we note in Section 2.5, the implementation is sufficiently devastating as to motivate immediate action.

In Section 2.8, we discuss further strategies to mitigate and recover from overrunning the victim.

2.2.3 Lazy Opt-Ack

Lazy opt-ack is a variant of the standard opt-ack attack. Recall that the main difficulty in our implementation is in remaining synchronized with the sender's sequence number. The synchronization issue can be totally avoided if the attacker ACKs any segment that it actually receives, independent of missing segments. This lazy variant is malicious in that the attacker is effectively concealing any packet loss, thereby creating a

flow that does not decrease its sending rate when faced with congestion (i.e., a non-responsive flow). Since the attacker is using the actual RTT to the victim, it generates less traffic than the attack described in Algorithm 1. However, it is well known [51] that in a congested network, a non-responsive flow can cause compliant flows to back off, creating a DoS. Note that the lazy variant is different from the standard attack in that it is impossible for the attacker to overrun the victim. This observation is precisely what makes many existing solutions insufficient. The skipped segments solution we provide in Section 2.4.2 protects against both the lazy and standard attacks.

2.2.4 Distributed Opt-Ack Attack

In this section, we consider the distributed case where *multiple attackers* run the opt-ack attack in parallel, trivially, and with devastating effect. The only coordination required is that each attacker chooses a different set of victims. Because a single attacker can solicit an overwhelming number of packets (as we will see in Section 2.3) *a relatively small group of attackers can cause the Internet to suffer widespread and sustained congestion collapse*.

First, because opt-ack targets any TCP server, there are *millions* of potential victims on the Internet. Considering P2P file distribution networks alone, Kazaa and Gnutella have over 2 million[87, 85, 63] and 1.4 millions [90] users respectively that each host large multimedia files. While P2P nodes are typically low bandwidth home users, the popular content distributor Akamai runs over 14,000 [8] highly provisioned, geographically distributed servers.

It is not immediately clear how much traffic is necessary to adversely affect the wide-area Internet. One data point is the traffic generated from the Slammer/Sapphire worm. In [103], Moore et al. used sampling techniques to estimate the peak global

worm traffic at approximately 80 million packets per second. At 404 bytes/packet, the worm generated approximately 31GB/s of global Internet traffic. Subsequent email exchanges by Internet operators [106] noted that many access links were at full capacity, and completely unusable. However, as noted in Table 2.1, it is theoretically possible for *a single attacker on a modem* to generate more than enough traffic to exceed this threshold using large $wscale$ values. If using large $wscale$ values were infeasible (for example, if packets containing the $wscale$ option were firewalled), then *five attackers* on T3 connections with more typical TCP options, i.e., $mss = 536$ and $wscale = 0$, would be sufficient to match the Slammer worm's traffic. If each attacker targeted sufficient number of victims, such that the load on no one victim was notably high, it would be difficult to locally distinguish malicious and valid data streams. So, unlike Slammer, there would be no clear local rule to apply to thwart the attack.

The traffic from the Slammer worm was not sufficient to push the core of the Internet into congestion collapse. Because of the inherent difficulty in modeling wide scale Internet phenomena, it is not clear how to estimate the number of opt-ack attackers required to induce such a collapse. However, a single attacker on a modem or a small number of other attackers can induce traffic loads equivalent to the Slammer worm. Recent studies[10] show that there exists networks of compromised machines("botnets") with over 200,000 nodes. Since each of these nodes represents a possible attacker, a large distributed opt-ack attack could easily be catastrophic.

2.3 Amplification Factor

While it is not surprising that a victim can be induced to send large amounts of data into the network, the actual opt-ack amplification factor is truly alarming. For example,

Attacker Speed ($\mathcal{T}_{max}$)	$mss = 536$ $wscale = 0$	$mss = 88$ $wscale = 0$	$mss = 536$ $wscale = 14$	$mss = 88$ $wscale = 14$
Multiplier $\beta = 1$	1336 B/s	1958 B/s	20.9 MB/s	30.6 MB/s
Modem $\beta = 7000$	8.9 MB/s	13.1 MB/s	142.7 GB/s	209.2 GB/s
DSL $\beta = 16000$	20.4 MB/s	29.9 MB/s	326.1 GB/s	478.1 GB/s
T1 $\beta = 193000$	245.8 MB/s	360.5 MB/s	3.84 TB/s	5.63 TB/s
T3 $\beta = 5625000$	7.0 GB/s	10.3 GB/s	112.0 TB/s	164.1 TB/s

Table 2.1: Maximum theoretical flooding for various attacker speeds and options. MB/s refers to 2^{20} bytes/second, etc.

an attacker on a 56Kbps modem can cause victims to push 71.2Mb/s of traffic into the network with standard TCP options. In the worst case, i.e., mss=88 and $wscale$=14, the same attacker can cause up to 1.6Tb/s of traffic to be generated. See Table 2.1 for other examples. While estimating these bounds is fairly simple (Section 2.3.1), our analysis includes the more sophisticated issues (Section 2.3.2) of maximum number of victims due to application time out, minimum victim bandwidths, and the time to grow the force of the attack.

2.3.1 Congestion Control Bounds

The upper bound on the traffic induced across all victims from a single attacker is a function of four items: the number of victims (n), and for each individual victim i, the rate at which ACKs arrive at each victim (α_i), the maximum segment size (mss_i), and the size of the victim's congestion window (w_i). Note that the attacker can use a single ACK to acknowledge an entire congestion window of packets. The number of packets from a single victim in the network at any one time is $\lfloor\ w_i/mss_i\ \rfloor$. If we assume a

standard TCP/IP 40 byte header with no options and that the link layer is Ethernet (14 byte header), then the packet size is $54 + mss_i$ bytes. The rate of attack traffic $\mathcal{T}$ in bytes/second is simply the sum across each victim of the product of the ACK arrival rate(α_i), the number of packets($\lfloor w_i/mss_i \rfloor$), and the size of each packet ($54 + mss_i$), or:

$$\mathcal{T} = \sum_{i=1}^{n} \alpha_i \times \left\lfloor \frac{w_i}{mss_i} \right\rfloor \times (54 + mss_i) \tag{2.1}$$

To find the theoretic maximum possible flooding rate, $\mathcal{T}_{max}$, we have to consider the bandwidth the attacker *dedicates* to each victim (i.e., the thin dark line in Figure 2.1 from attacker to victim), which we denote β_i for the ith victim. If we assume β_i is measured in bytes/second, each ACK is 40 bytes, and again assume the link layer is Ethernet (14 byte header), then we find that $\beta_i = 54\alpha_i$ at maximum bandwidth. We use $\beta = \sum_{i=1}^{n} \beta_i$ to denote the attacker's total attack bandwidth to all victims, and for simplicity assume that each victim has the same mss and w_i, i.e., $\forall i; mss_i = mss$ and $\forall i; w_i =$ maxwindow. Thus, substituting β, mss, and maxwindow into (2.1) and rearranging produces:

$$\mathcal{T}_{max} = \left\lfloor \beta \times \text{maxwindow} \times \left(\frac{1}{mss} + \frac{1}{54} \right) \right\rfloor \tag{2.2}$$

As noted before, the maximum congestion window ($maxwindow$) is typically 65535. For a wide area connection, a typical value for mss would be 536.[3] Substituting these values into (2.2) produces $\mathcal{T}_{max} = 1336\,\beta$. Thus, using typical values, an attacker has an amplification factor of 1336. In real world terms, that means an attacker on a 56 Kilo-*bits*/s modem ($\beta = 7000$ B/s) can in theory generate 9,351,145 B/s or approximately 8.9MB/s of flooding summed across all victims. This value is more than the capacity of a T3 line, and close to the theoretical limit of a 100Mb Ethernet connection.

[3] Another typical value is mss=1460, but the effect on $\mathcal{T}_{max}$ is minimal

See Table 2.1 for the amplification factor, and more examples.

For non-standard values of mss and $maxwindow$, the amplification factor of the opt-ack attack is significantly magnified. Recall from RFC 793 [121] that the mss is a 16 bit value set via TCP option by the receiver (the attacker) in the SYN packet. Looking at (2.2), decreasing mss makes packets smaller, but increases the number of packets sent, for a net increase in $\mathcal{T}_{max}$. While it is already well known [119] that transferring large files with a low mss value can create denial of service conditions, the damage is significantly amplified when coupled with the opt-ack attack. As noted by Reed, the minimum mss is highly system dependent with values varying from 1 to 128 for popular OSes. For example, Windows 2000 and Linux 2.4 have a minimum mss of 88, whereas Windows NT4's is 1. Reed also noted that at extremely low values of mss, the server can become CPU-bound because of high context switching from fielding too many interrupts.

In addition, RFC 1323 [73] defines the $wscale$ TCP option to increase $maxwindow$. The attacker can use the $wscale$ option to scale the congestion window by a factor of 2^{14}, increasing $maxwindow$ to 65535×2^{14} or approximately 10^9 bytes. As shown in Table 2.1, the effect of window scaling on $\mathcal{T}_{max}$ is dramatic. Specifically, a malicious connection with mss=88 and $wscale$=14 can reach a theoretical amplification factor of 32,085,228 or over *32 million*. With this level of amplification, it is possible for an attacker on a modem targeting many victims to induce more traffic than the Slammer worm (Section 2.2.4).

2.3.2 Application Timeouts and Growing the Congestion Window

Fortunately, there is a significant difference between the theoretical and practical effects of the opt-ack attack. First, there is a limit to the number of victims an attacker

can target at once. From Algorithm 1, the attacker needs to connect to each victim, and retrieve the initial sequence number (ISN) for each connection *before* sending the application data request (Line 8), e.g., an http get or ftp file request. Note that it is very difficult for an attacker to learn new ISNs while attacking other hosts, because its incoming links are saturated. Thus, the attacker must connect to the entire victim set, learn their ISNs, and then launch the attack. However, if the attacker targets too many nodes, the loop in Algorithm 1 at line 3 will take too long, and the first victim will timeout at the application level before receiving its data request. If victims timeout before the request is sent, then the connection is dropped and the attack foiled. Note that the minimum time for the attacker to complete a TCP connection is the time to send the SYN packet and the time to send the ACK packet ($2 \times (40 + 14) = 108$ bytes). This assumes that the attacker efficiently interleaves SYNs and ACKs to multiple victims, such that each victim's time to respond with the SYN-ACK is not a limiting factor. Then, using the minimum connection time and the application timeout (ATO) value, we can calculate maximum possible number of victims as follows:

$$\text{max victims} = \text{ATO} \times \frac{\beta}{108} \tag{2.3}$$

Realistic ATO values are highly application dependent, and even within applications, the timeout value is tuned to the specific environment and workload. For example, a survey among an arbitrarily chosen set of popular websites showed ATO values for http ranged from 135 seconds (www.google.com) down to 15 seconds (www.cnn.com). As a further data point, the popular web server package Apache has a default timeout of 300 seconds. However, even with an ATO of 15 seconds, an attacker on a home DSL line (128Kbps up-link, $\beta = 16000$) can attack 2307 victims in parallel.

Another limitation is that it is not possible to create more flooding than the sum of the victims' up-links capacities. A direct implication of (2.3) is that each victim

must have a minimum amount of bandwidth in order for an attack to reach the rates described in Table 2.1. To calculate the minimum bandwidth of each victim, we divide (2.2) by (2.3) and produce:

$$\text{min victim bandwidth} = \frac{\text{maxwindow}}{\text{ATO}} \times \left(\frac{108}{mss} + 2 \right) \tag{2.4}$$

In other words, the same attacker on a home DSL line also needs each of the 2307 victims to have bandwidth in excess of 220MB/s in order to achieve 478.1GB/s in flooding as described in Table 2.1 (ATO=15, $mss = 88$, $wscale = 14$). Obviously, this is not immediately practical. However, if any victim is below the minimum bandwidth from (2.4), the result is simply that the sender saturates its outgoing link, which is sufficiently devastating to the victim's local network.

The last bound on the amplification is the time to grow the congestion window. The attacker must send sufficient number of ACKs to each victim in order to increase the congestion window to from its initial value (one mss) to its maximum value($maxwindow$). By Algorithm 1, as the number of victims increase, the time between ACKs sent to an individual node diminishes. Thus, we can calculate the minimum time required for the attack to reach maximum effect:

$$\text{min time} = \frac{54 \times \text{maxwindow} \times n}{\text{mss} \times \beta} \tag{2.5}$$

The values in Table 2.1 are upper bounds on $\mathcal{T}$ and may in fact never be achieved in practice. Other factors such as the victims' TCP send buffer size, outgoing bandwidth, and processing capacity affect the rate at which traffic is produced (as discussed in Section 2.8). In Section 2.5, we show that our implementation achieves nearly 100% of $\mathcal{T}_{max}$ in simulation.

2.4 Defending against Opt-Ack

In this section, we present a simple framework for evaluating different defense mechanisms against the opt-ack attack, and evaluate potential solutions within that framework. Finally, we present one particular solution, randomly skipping segments, that efficiently and effectively defends against opt-ack. We also describe an implementation of randomly skipped segments in detail.

2.4.1 Solutions Overview

Any mechanism that defends against opt-ack should minimally possess the following qualities:

1. **Easy to Deploy** Due to the severity of the attack, any solution should be practically and quickly deployable in the global infrastructure. Minimally, the solution should allow incremental deployment, i.e., unmodified clients should be able to communicate with modified servers.

2. **Efficient** Compliant (i.e., non-attacking) TCP streams should suffer minimal penalty under the proposed solution. Also, low power embedded network devices do not have spare computational cycles or storage space. Because the problem is endemic to all implementations, the solution needs to be efficient on all devices that implement TCP.

3. **Robust** Any fix needs to defend against all variants (Section 2.2.3) of the opt-ack attack.

4. **Easy to Implement** This is a more pragmatic goal, leading from the observation that TCP and IP are pervasive, and run on an diverse range of devices.

Any change in the TCP specification would affect hundreds (or thousands) of different implementations. As such, a simpler solution is more likely to be implemented.

In the rest of this section, we describe a number of possible defenses against optack, and present a summary of solutions in Table 2.2.

Secure Nonces

One possible solution is to require that the client prove receipt of a segment by repeating an unguessable nonce. Assume each outgoing segment contains a random nonce which the corresponding ACK would have to return in order to be valid. Savage [127] et al. improve on this solution with *cumulative* nonces. In their system, the response nonce is a function of all of the packets being acknowledged, i.e., a cumulative response, ensuring that the client actually received the packets it claims to acknowledge.

Unfortunately, cumulative nonces are not practically deployable. They requires both the client and server to be modified, preventing incremental deployment. If deployment was attempted, updated servers would be required to maintain backward compatibility with non-nonce enabled clients, until all client software was updated. As a result, updated servers would have to chose between being vulnerable to attack or compatibility with unmodified clients. Additionally, nonces require additional processing and storage for the sender. Calling a secure pseudo-random generator once per packet could prove expensive for devices with limited power and CPU resources, violating our efficiency goal.

To aid deployment, one could consider implementing nonces in existing, unmodified clients via the TCP timestamp option. The send could replace high order bits of the timestamp with a random challenge, and any non-malicious client which imple-

mented TCP timestamps would respond correctly with the challenge. If a client did not implement timestamps, the server could restrict throughput to something small, e.g, 4Kb/s. While this improves on the deployment of nonces, this solution still has problems. First, it loses the critical cumulative ACK property of Savage's solution. That is, an acknowledgment for a set of packets does not necessarily imply that all packets in the set were received, which opens itself to the lazy opt-ack attack. Second, as we discuss in Section 2.4.1 below, bandwidth caps are not effective.

Require ACK Alignment

One aspect of TCP that makes the opt-ack attack possible is the predictability of the ACK sequence. Furthermore, because communication is a stream, the client can in theory acknowledge the bytes anywhere in the sequence, not just along packet boundaries. Clark [39] cites the ability to retransmit one large packet when a number of smaller packets are lost as a main benefit of this. In practice, with large buffers and client-side window scaling, most implementations send only packet-aligned acknowledgments. We could use this insight to require clients to acknowledge only along packet boundaries, and then add a small, unpredictable amount of noise to the packet size. For example, with equal probability, the server could send a packet of size mss or of size $(mss - 1)$. In this way, a client that actually received the packet would get information that a opt-ack attacker does not have: the actual packet size. Only a client that actually receives all of the packets could continue to correctly ACK them probabilistically over time. The noise could be generated pseudo-randomly as a function of the sequence, so storing per-outstanding-packet state at the server could be avoided.

ACK alignment suffers from many of the same problems as non-cumulative secure nonces. Specifically, it is not secure against the lazy variant of opt-ack, and ACK

alignment could be expensive for low powered devices. In addition, network devices, such as network address translation (NAT) box translating FTP or a firewall, could change the size of or split the packet in flight. Any such change in the packet size would result in false positives, which are unacceptable.

Bandwidth Caps

The obvious solution to an attacker consuming too many resources, as is the case with the opt-ack attack, is to limit resource consumption. Conceivably, this could be done at the server with a per IP address bandwidth cap, but unfortunately this is not sufficient. First, any restriction on bandwidth can simply be over come by increasing the number of victims. Suppose for example, that each victim sets the policy that no client can use more than a fraction $c \in (0, 1]$ of their bandwidth. Then the attacker need simply increase the number of victims by $1/c$ to maintain the same total attack traffic. Further, bandwidth caps interfere with legitimately fast clients, violating our efficiency goal.

Network Support

Since the opt-ack attack stream acts essentially as a non-responsive flow, one possible defense would be to implement fair queuing or "penalty boxes" in the network. As [51] notes, this is not a new problem, and there exists a wealth of research on the subject [27, 52, 49, 35]. A similar solution would be force flows that cause congestion to solve puzzles[153] in order to maintain their rate. However, these solutions are not currently widely deployed and the cost of doing so would seem prohibitive.

Solution	Efficient	Robust	Deployable	Simple	Change TCP Spec.
Cumulative Secure Nonces	yes	yes	no	yes	client & server
Secure Nonces w/ timestamps	yes	no	yes	yes	server only
ACK Alignment	yes	no	yes	yes	server only
Bandwidth Caps	no	no	yes	yes	no
Network Support	yes	yes	no	no	no
Random Pauses	no	no	yes	yes	server only
Skipped Segments	yes	yes	yes	yes	server only

Table 2.2: Summary of Defenses to Opt-Ack Attack

Disallow Out of Window ACKs

A straightforward solution is to change the TCP specification to disallow out of window ACKs. Recall from Section 2.2.2 that our implementation runs the risk overrunning the victim. If a victim sent a reset, terminating the connection, upon receipt of an out of window ACK, the opt-ack attack would be mitigated. However, this is not a viable solution as this opens non-malicious connections to a new DoS attack. A malicious third party could inject a forged out of window ACK into a connection, causing a reset [154]. Because the ACK is out of window, there would be no need to guess the sequence space. Also, compliant receivers can send out of window acknowledgments due to delays or packet reordering. For example, suppose ACKs for packets numbered 2 and 3 are sent but received in reverse order. The ACK for packet 3 would advance the window, and then the ACK for packet 2 would be and out of window ACK, causing a RST.

Random Pauses

As described in Section 2.2.2, the main difficulty in the implementation was to keep the attacker's sequence numbers synchronized with what the server was sending. Thus, one way of thwarting the attacker would be for the server to randomly pause. A client correctly implementing the protocol will reciprocate by pausing with the server and waiting for more data. On the other hand, an attacker will continuously send ACKs for packets not yet sent, exposing the attack. This solution does not prevent against the lazy variant of the opt-ack attack. Also, if the server applied this pausing test too often, performance could suffer significantly. Our final proposed solution expands on the random pausing idea with additional robustness and minimal performance penalty.

2.4.2 Proposed Solution: Randomly Skipped Segments

The main problem with the random pause solution is the efficiency penalty to non-malicious clients. Instead of pausing, we propose the server *skip* sending the current segment, and instead send the rest of the current window. Note that this is equivalent to locally, *intentionally* dropping the packet. A client that actually gets all of the packets, save the skipped one, will start re-ACKing for the lost packet, thereby invoking the fast retransmit algorithm. However, an attacker, because it does not have a global view of the network, cannot tell where along the path a given packet was dropped, so it cannot tell the difference between an intentionally dropped packet and a packet dropped in the network by congestion. Thus, an attacker will ACK the skipped packet, alerting the server to the attack. Note that usually fast retransmission indicates network congestion, so the congestion window is correspondingly halved. However in this case, retransmission was not invoked due to congestion in the network, so the sender should not halve the congestion window/slow start threshold as it typically would. Given that

most modern TCP stacks implement selective acknowledgments (SACK)[98], this solution is significantly more efficient than randomly pausing (see Section 2.5 for performance). *The only penalty applied to a conforming client is a single round trip time in delay*.

To determine how often to apply the skipped packet test, we maintain a counter of ACKs received. Once a threshold number of ACKs are received, the skip test is applied. It is important that the threshold be randomized, as the security of this system requires that the attack not predict which segment was be skipped. However, there is an obvious trade off in where to make the skipped packet threshold. If it is too low, the server will lose efficiency from skipping packets too often. Setting the threshold too high allows the attacker to do more damage before being caught (see Section 2.5 for an exploration of this trade-off). Our solution is to chose the threshold uniformly at random over a configurable range of values.

This simple skipped segment solution meets all of our goals. It is efficient: compliant clients suffer only one round trip time in delay, the computational costs consist of keeping only an extra counter, and the storage costs are trivial (5 bytes per connection, described below). The skipped packet solution is robust against the variations of the attack described in Section 2.2.3, because it inherently checks whether a client actually received the packets. This solution is a local computation, so it needs no additional coordination or infrastructure, i.e., the deployment requirements are met. Best of all, it is transparent to unmodified clients, allowing for incremental deployment.

2.4.3 Skipped Packet Implementation

We implemented the skipped packet solution for the Linux 2.4.24 kernel. The total patch is under 200 lines (including comments, prototypes, and headers), and was de-

veloped and tested in under one week's time by someone previously unfamiliar with the Linux kernel. We add two entries to the per connection state (struct tcp_opt): opt_ack_mode (1 byte) and opt_ack_data (4 bytes). Further, we add 3 global configuration variables: $sysctl\_tcp\_opt\_ack\_enabled$, $sysctl\_tcp\_opt\_ack\_min$, and $sysctl\_tcp\_opt\_ack\_max$. When a new connection is created, opt_ack_mode is initialized to $OPT\_ACK\_MODE\_COUNTDOWN$, and opt_ack_data is set to a number uniformly at random between $sysctl\_tcp\_opt\_ack\_min$ and $sysctl\_tcp\_opt\_ack\_max$, inclusive. With each successful ACK, opt_ack_data is decremented, until it reaches zero. Upon opt_ack_data reaching zero, we set opt_ack_mode to $OPT\_ACK\_MODE\_SKIP$, update the send head pointer to the next block (skipping the segment), and save the sequence number of the segment skipped into opt_ack_data. A compliant client will ACK the beginning of the hole (i.e. the sequence in opt_ack_data), where a malicious attacker will ACK a segment past the hole. If the client ACKs a segment before the hole, we leave the test in place until another ACK arrives. If the client ACKs past the hole, it fails the test: we reset the connection and log a message to the console. In implementation, we use parameters $sysctl\_tcp\_opt\_ack\_min = 100$ and $sysctl\_tcp\_opt\_ack\_max = 200$, as suggested by our evaluations in Section 2.5.3. Last, under Linux, the retransmission code automatically handles resending the skipped segment for clients that correctly ACK the beginning of the hole.

The description of the fix is complete, except for a few additional details. If a timeout occurs in the middle of the skip test, we need to reset the threshold countdown, and go back to mode $OPT\_ACK\_MODE\_COUNTDOWN$. The reasoning is this: if the segment before the hole is lost, and there are no segments after the hole (or they are all lost), then the client will not ACK the beginning of the hole, until after the retransmit. However when a timeout occurs, the retransmit code might resend the skipped segment, negating the test. Resetting the threshold counter and changing the

mode in this obscure case solves this problem.

Also, to insure that the randomly skipped segments solution does not introduce a new DoS attack, we must ignore out of window ACKs during the skipped segments test. Otherwise, it might be possible for a malicious node to convince a server that a benevolent client was performing an opt-ack attack.

2.5 Attack Evaluation

We evaluate the feasibility and effectiveness of the opt-ack attack in a series of simulated, local area, and wide area network experiments. In the first set of simulations, we determine the total amount of traffic induced by the opt-ack attacks. Next, we determine the effect of the attack on other (honest) clients trying to access the victim. We also present results for the amount of traffic (described in Section 2.3) our real world implementation actually achieves across a variety of platforms and across different file sizes. Finally, in Section 2.5.3, we evaluate the efficiency of our skipped segment solution.

2.5.1 Simulation Results

We have implemented the opt-ack attack in the popular packet level simulator ns2 and simulate the amount of traffic induced in various attack configurations. In each experiment, there is a single attacker and multiple victims connected in a star topology. Each victim has a link capacity of 100Mb/s, and all links have 10ms latency (the choice of delay is arbitrary because it does not affect the attack). We vary the number of victims, and the mss and $wscale$ of the connection. The attacker makes a TCP connection to each victim in turn, and only sends acknowledgments once all victims

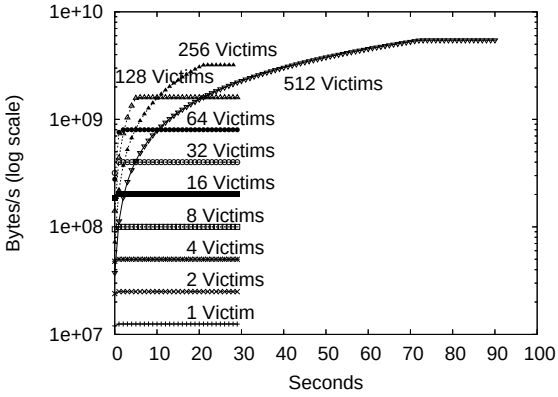


Figure 2.2:]

Maximum Traffic Induced Over Time; Attacker on T1 with mss=1460,$wscale$=4

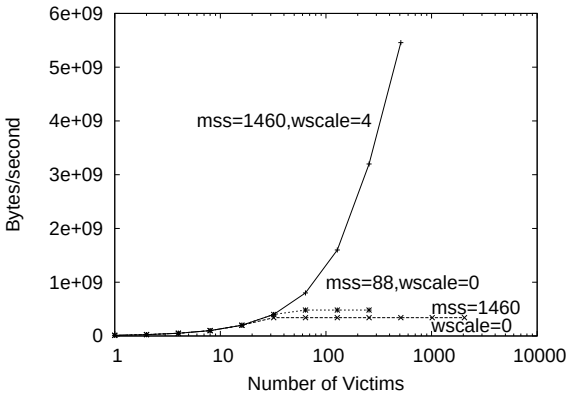


Figure 2.3: Maximum Traffic Induced By Number of Victims; Attacker on T1

have been contacted. Victims are running the "Application/FTP" agent, which uses an infinite stream of data.

In Figure 2.2, we show the sum of the attack traffic generated over time with variable numbers of victims. In this experiment, the attacker is on a T1 (1.544Mbs) and uses connection parameters mss=1460 and $wscale$=4 ($maxwindow$=1048576). When the number of victims is less than 512, the amount of flooding is limited by the sum of the bandwidths of the victims, as predicted by Equation 2.4 in Section 2.3. The

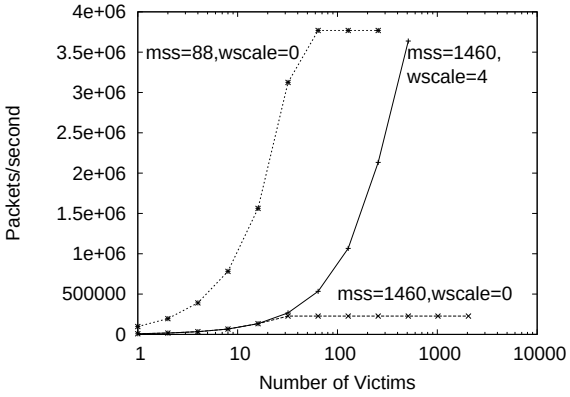


Figure 2.4: Maximum Packets Induced By Number of Victims; Attacker on T1

amount of traffic doubles as the number of victims double until 512 victims. The number of victim's increases, the attack takes longer to achieve full effect as predicted by Equation 2.5. The case with 512 victims took 73 seconds to reach it peak attack rate, while all others did so in under 30 seconds. At 512 victims, the simulation achieves 99.9% of the traffic predicted by Equation 2.2.

As shown in Figure 2.2, once the attack's maximum effect is reached, it can be sustained indefinitely. In Figure 2.3 and Figure 2.4, we show the maximum traffic induced as we vary the number of victims, mss and $wscale$ for bytes/second and packets/second respectively. As predicted by Section 2.3, attackers with a lower mss produce more traffic than one with a higher value. Likewise, an increased $wscale$ has a dramatic increase in the total traffic generated.

Due to CPU and disk space limits, we were not able to simulate more than 512 victims for all parameters, or $wscale$ values above 4, despite the fact that our simulation machine was a dual processor 2.4Ghz Athlon-64 with 16GB ram and 300GB in disk.

2.5.2 Real World Implementation

In order to validate the attack, we implemented it C and experimented on real machines in a number of network settings. We measure the effect of the attack on a single victim and the actual bandwidth generated from a single victim running various popular operating systems.

It should be noted that we did not experiment with multiple attackers or multiple victims due to real world limitations of our test bed. Our experiments with a single attacker and single victim were sufficient to cause overwhelming traffic on our local networks. It would be irresponsible and potentially illegal to have tested the distributed attack on a wide-area test bed (e.g., PlanetLab[114]), and even our simple one attacker-one victim wide-area experiments caused network operators to block our experiments.[4]

Single Victim DoS Effect - Lan and Wan

Experiment	Average (sec)	Dev.	Increase
No Attack	89.11	0.007	1
LAN Attack	1552.03	141.76	17.42
WAN Attack	779.93	139.32	8.75

Table 2.3: Average Times with Deviations for a Non-malicious Client to Download a 100MB File

This experiment measured the effect on a third party client's efficiency in downloading a 100MB file from a single victim during various attack conditions. We re-

[4]Incoming traffic to one author's home DSL IP address was temporarily blocked as a result of these experiments. This did not serve to stop the attack, as the outbound ACKs could still be sent. However, this served as evidence that we should cease the experiment.

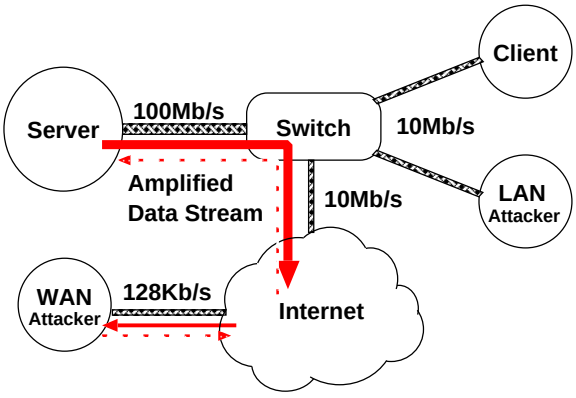


Figure 2.5: Topology for Experiments

peated this experiment with no attacker, with an attacker on the local area network, and with an attacker across the Internet (see Figure 2.5). The local area attacker was a dual processor Pentium III running Linux with a 10Mb Ethernet card, while the WAN attacker was a 100Mhz Pentium running GNU/Linux on an asymmetric 608/128 Kb/s downstream/upstream residential DSL line. The latency on the WAN link varied over time, with a average RTT of 13.5ms.

A typical web server runs on a fast local area network, which connects to a slower wide area network. In order to emulate this bottleneck, and also to safeguard against saturation of our production Internet connection, we connected our test web server to the world via a 10Mb connection on a Cisco Catalyst 3550 switch. Furthermore, both LAN and WAN attackers were configured to use $TargetBandwidth$ of 10^9 bytes/second, and $\beta = 16000$ bytes/s as their local bandwidth setting (see Section 2.8 for description). The intuition is that the LAN and WAN attackers should be equally capable with respect to their available bandwidth, but the WAN attacker must compensate for more end-to-end jitter and delay. Each run used mss=536 and $wscale$=0, i.e., typical values for Internet connections. Each experiment was repeated 10 times and the values averaged. The numbers were measured with a command line web client (similar to *wget*)

specially instrumented to measure bandwidth at 10 ms intervals. We present the results from these experiments in Table 2.3. The "Increase" column refers to the increase in time relative to the "No Attack" baseline.

The effect of the attack is significant. The 100MB file takes on average 17.42 and 8.75 times longer to download under LAN and WAN attack, respectively. We believe that the time difference between the WAN and LAN attacks is due to the increased jitter of the wide area Internet, and the increased standard deviation in the results supports this. This variability makes keeping synchronizing with the victim more difficult due to the buffered ACK problem, as described in Section 2.8. However, more advanced attackers could target more victims (Section 2.2.2) or potentially employ more sophisticated segment prediction to increase the effectiveness of the attack.

We also re-ran the same set of experiments with a set of hubs in place of the switch, effectively removing queuing from the system. The times to download the 100MB file while under attack were reduced to 5 times and 4.5 times the baseline for LAN and WAN attackers, respectively. In other words, having queuing on the bottleneck link significantly *increased* the damage from the attack. We surmise this is because the opt-ack attacker used $mss = 536$ and the non-malicious client, since it was on local Ethernet, used $mss = 1448$. Once the queue was full, the switch could service two of the attack packets before there was room for a legitimate (i.e. destined to the non-malicious client) packet. Effectively, the higher rate of smaller packets caused the switch to drop more non-malicious/legitimate packets. Removing the queue from the system reduced the amount of dropped legitimate packets, therefore increasing non-malicious throughput.

Amplification Factors

To evaluate the potential effectiveness of the distributed opt-ack attack, we measure the amount of traffic that our implementation code can induce in a single victim. In this experiment, we use the LAN attacker, as above, to attack various operating systems including GNU/Linux 2.4.24, Solaris 5.8, Mac OS X 10.2.8, and Windows XP with service pack 1. For this experiment, instead of a web server, each victim ran a program that streamed data from memory. This was done to remove any potential application-level bottlenecks from the experiment. As above, the attacker used parameters $\beta = 16000$, $mss = 536$, and $wscale = 0$. We measured the bandwidth in one second intervals using a custom tool written with the libpcap library. Each experiment in Table 2.4 was run 10 times, averaged, and is shown as an amplification factor of the attacker's used local bandwidth.

We believe that the variation in amount of flooding by OS is due to the lack of sophistication of our attack implementation. The amplification factor for Linux is 251.6 times the used bandwidth, which translates to $251.6/1336$ or approximately 18% of the theoretical maximum traffic, $\mathcal{T}_{max}$. This low number is in part because the implementation sends four ACKs per window (as described in Section 2.8), which alone limits the attack to 25% of $\mathcal{T}_{max}$.

Smaller Files

In the first set of experiments, we assumed the victim served a 100MB file for the attacker to download. While there are files of this size and larger on the web (Windows XP service pack 2 is 272MB and heavily replicated), we repeated the experiment with smaller file sizes. The test bed is exactly as above (Figure 2.5) with the LAN attacker and queuing. Again, the non-malicious client downloaded a 100MB file from the

OS	Avg. KB/s	Dev.	Amplification
Linux 2.4.24	3931.93	1102.38	251.6
Mac OSX	806.2	258.1	51.6
Solaris 5.8	3150.6	1301.1	201.6
Windows XP	640.62	378.85	41.0

Table 2.4: Average bytes/s of Induced Flooding, Standard Deviation, and Amplification Factor of Attacker's Bandwidth

File Size	Time(s)	Dev.	Factor Increase
No Attack	89.11	0.007	1
100MB File	1552.03	141.76	17.42
10 MB File	281.00	9.81	3.15
1 MB File	152.87	21.48	1.75
512 KB File	106.63	9.03	1.20

Table 2.5: Average Times for Client to Download a 100MB File, With Attacker Downloading Various-Sized Files

victim. In this experiment, we vary the size of the file the attacker downloads. The results are presented in Table 2.5. As expected, smaller files are less useful for the attacker. However, even 10MB files cause the client to slow down by a factor of 3.15, so smaller files can still cause some damage.

Clearly, these results depend upon the attack implementation described in Section 2.8, and there are inefficiencies in our implementation that can be improved upon. For example, the implementation code creates a new TCP stream each time a download is complete, and starts again in a loop. An easy optimization would have been to take

advantage of HTTP's persistent connections and download multiple files on the same stream. However, the results presented here are sufficiently motivating. As above, each data point represents the average of 10 experiments. The "Factor Increase" column refers to the increase relative to the "No Attack" baseline in Figure 2.3.

2.5.3 Performance of Skipped Segments Solution

In the final experiment, we evaluate the efficiency of our proposed randomly skipped segments solution. Specifically, we measure the time for a non-malicious client on the LAN to download a 100MB file from the server with and without the fix, with and without selective acknowlegement (SACK) enabled on the client, and with various threshold values for the fix. The download times were measured with the UNIX *time* utility. Each experiment was run ten times, the results were averaged, and they are presented in Tables 2.6 and 2.7, with and without SACK respectively. The two numbers in the first column of each table refer to the threshold values used for $sysctl\_tcp\_opt\_ack\_min$ and $sysctl\_tcp\_opt\_ack\_max$ in each experiment.

The results show that the performance hit from the proposed fix is negligible for most parameters. Even when we chose the threshold to be intentionally inefficient, i.e., skipping a segment every 10 to 20 ACKs, the fix maintained 99.457% efficency. We found that varying $sysctl\_tcp\_opt\_ack\_min$ value had little effect when combined with SACK, but made a 1% difference without SACK. We believe the loss from skipping segments every 100-200 ACKs, i.e., less than 0.1% with or with SACK, is an acceptable price for defeating this attack.

Experiment	Time(s)	Deviation	%
Unfixed	89.136	0.007	100%
Fixed: 10-20	89.623	0.980	99.457%
Fixed: 1-200	89.158	0.0234	99.975 %
Fixed: 100-200	89.167	0.0256	99.965 %

Table 2.6: Time to Download a 100MB File for Various Fix Options - *SACK Enabled*

Experiment	Time(s)	Deviation	%
Unfixed	89.143	0.0152	100%
Fixed: 1-200	90.048	0.3960	98.994%
Fixed: 100-200	89.145	0.0111	99.998%

Table 2.7: Time to Download 100MB File for Various Fix Options - *SACK Disabled*

2.6 Related Work

There is a long history of denial of service attacks against TCP, which we divide broadly into brute force attacks and more efficient attacks.

2.6.1 Brute Force DoS Attacks

The salient feature of brute force attacks is the fact that it is incumbent upon the attackers to provide the resource that ultimately overloads the victim. Example attacks include bandwidth flooding, connection flooding, and Syn flooding. The commonality among these attacks is that the attackers must be capable of draining more of a precious resource, be it bandwidth, file descriptors, or memory, than the victim's capacity. One possible defense against these attacks is to obtain more of the resource, i.e. buy

more memory or lease more bandwidth.

The danger of the opt-ack attack is that the victim's own resources are being turned against them. If the victim adds more bandwidth capacity, then the attacker can then use the additional bandwidth to generate more traffic. While there is a bound, $\mathcal{T}_{max}$, to the traffic the attacker can induce (see Section 2.3), the victim is not necessarily safe if it secures more than $\mathcal{T}_{max}$ in capacity. Increasing the victim's local capacity pushes the bottleneck link further into the network, injecting more traffic into the Internet backbone.

2.6.2 Efficient Attacks

We refer to efficient attacks as those that require little resources from the attacker but result in victims introducing large amounts of resources to the network, essentially performing their attack for them.

Smurf Attack

A smurf attack [135] consists of forging a ping packet from the victim to the broadcast address of a large network. In this way, a single packet is amplified by the size of the network, and redirected at the victim. A variant of this attack is to forge the ping from the broadcast address of the victim, forcing the victim's switches to do more work in duplicating the packet.

The amplification aspects of this attack are similar to the opt-ack attack. However, the attack signature of smurf makes it easy to detect and defend against: simply block traffic from a broadcast address or rate limit $ICMP\ ECHO$ traffic at the border router. In contrast, opt-ack is not known to have an obvious attack signature, and most site policies would not allow blocking TCP traffic.

Shrew Attack

The attack most similar to opt-ack is the Shrew [81] attack, in that it also attempts to exploit of TCP congestion control. In Shrew, an attacker sends traffic directly to the receiver/victim in short bursts, trying to force a retransmission due to packet loss. If the bursts are timed correctly, the sender's RTO period can be abused such that each retransmission coincides with another burst, and thus a DoS condition is created. Analysis shows that a simple square wave pattern of bursts forces the sender's RTO period to synchronize with the bursts. Further, the bursts can be sufficiently infrequent such that the average rate would not alert a potential intrusion detection system.

Despite these similarities, the two attacks are quite different. In Shrew, it is the receiver who is attacked directly, where with opt-ack, it is the sender who is attacked which indirectly impacts all receivers. Also, Shrew assumes that the attacker has enough bandwidth to directly force packet loss. This is reasonable when the path from the attacker to the receiver includes the bottleneck link from sender to receiver, but this not always the case. In contrast, opt-ack makes no such assumptions. With opt-ack, it is the sender's first-hop link that is saturated, which thereby becomes the bottleneck for all connections (assuming a single homed sender). Further, even a relatively weak opt-ack adversary, such as an attacker on a modem, presents a serious threat to a comparatively high bandwidth server.

Also, it seems that the main advantage of the Shrew attack is that the average attack traffic rate is low. However, if the attack became popular, it seems intuitive that intrusion detection systems could easily adapt by examining the maximum traffic rate in addition to the average traffic.

Lastly, as we note in Section 2.5, elements of the two attacks can be combined. An intelligent opt-ack attacker can vary the rate of ACKs sent to cause the return stream

to regularly burst like the Shrew attack. Using this method, it is apparent that more damage can be generated.

Misbehaving Receivers

As previously mentioned, Savage et al.[127] discovered the opt-ack attack as a method for misbehaving receivers to get better end-to-end performance. While they suggest that opt-ack can be used for denial of service, they did not investigate the magnitude of the amplification the attack can achieve. As a result, their cumulative nonce solution to the opt-ack attack does not consider global deployment as a goal. In this work, through analysis and implementation, we have shown that opt-ack is a serious threat. Further, we have engineered an efficient solution that does not require client-side modification, and thus is more readily deployable.

Reflector Attacks

In [112], Paxson discusses a number of attacks where the initiator can obscure its identity by "reflecting" the attack off non-malicious third parties. As a general solution, Paxson suggests upstream filtering based on the attack signature with the assumption that it is not possible to overwhelm the upstream filter with useless data. The work specifically mentions that if the attacker is able to guess the ISN of the third party, it is possible to mount a blind opt-ack attack against an arbitrary victim. No analysis is made of the amount of the amplification from the opt-ack attack, nor is it immediately clear what filter rules could be applied to arbitrary TCP data.

2.7 Discussion and Conclusion

We have described an analysis of the opt-ack attack on TCP and demonstrated that amplification from the attack makes it dangerous. We have also engineered an efficient skipped segments defense against attacks of this type that allows for incremental deployment. The opt-ack attack succeeds because it violates an underlying assumption made by the designers of TCP: that peers on the network will provide correct feedback. This assumption holds when clients are interested in receiving data, since false feedback will usually lead to worse end-to-end performance. However, the opt-ack attack shows that if malicious nodes do not care about data transfer integrity, they can cause widespread damage to other clients and to the stability of the network.

Since opt-ack violates an underlying assumption upon which TCP is based, we believe a proper solution for the opt-ack attack involves changing the TCP specification. Although new features can be added to TCP (e.g., cumulative nonces) to ensure the receiver TCP is in fact receiving all of the segments, this type of solution is difficult to deploy because it requires client modification. The skipped segment solution presented here requires modification of only high capacity servers, and is thus more readily deployable. In this chapter, we have described different mechanisms that can be used to defend against opt-ack attacks. We recommend a specific change to the TCP specification that we have shown to be easy to implement, efficient for fast connections, and which does not burden resource-poor hosts.

2.8 Implementing Opt-Ack

In this section, we describe an actual implementation of opt-ack against TCP. There are three reasons we chose to implement the attack in addition to simulating it. First, it

was clear that the opt-ack attack worked in theory, but we wanted to demonstrate that it was feasible in practice. Second, the implementation would allow us to test against deployed networks and gauge the effectiveness of the attack against real-world systems. Third, and most importantly, we hoped that in implementing the real attack, we would gather sufficient insight to design a viable solution. In the rest of the section, we describe our experience with implementing opt-ack, and highlight specific challenges present in real-world systems that we had to account for.

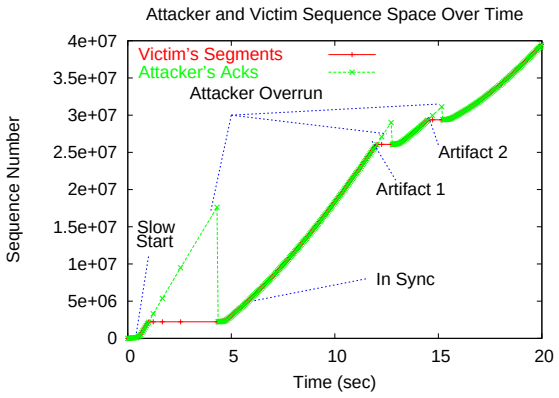


Figure 2.6: Attacker and Victim Sequence Space, Measured at Victim

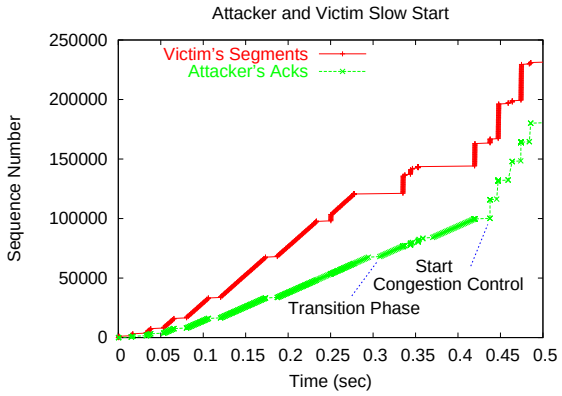


Figure 2.7: Detail: Attacker and Victim Slow Start

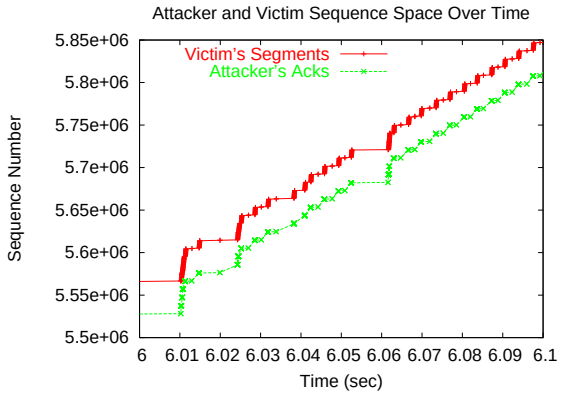


Figure 2.8: Detail: Attacker and Victim Synchronized

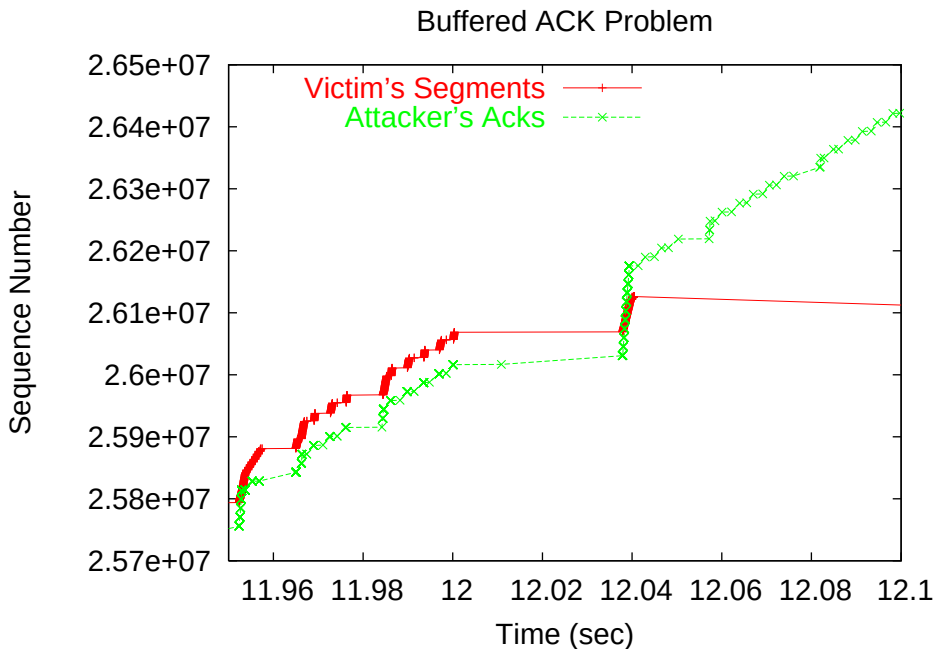


Figure 2.9: Artifact 1: Buffered ACKs

2.8.1 Recovery from Overruns

Compliant TCP streams are supposed to generate an empty segment upon receipt of an out of window ACK (Section 2.2.2). The attacker could use this empty segment to detect overruns, but the during the attack incoming link is typically saturated, so the empty segment will be dropped. Additionally, Linux ignores an out of window ACK, times out on previous unACK'ed packets, and retransmits them. Other OSes, specifically MacOS X 10.2 and Windows 2000, correctly generate the empty packet. However, while a stream is making progress, the sequence numbers of packets received increase monotonically (barring packet reordering). Upon a retransmission, or when an empty packet is received, the sequence number is less than or equal to the previous packet, breaking monotonicity. So, by monitoring the sequence numbers of packets actually received, the attacker can detect overruns when the sequence numbers

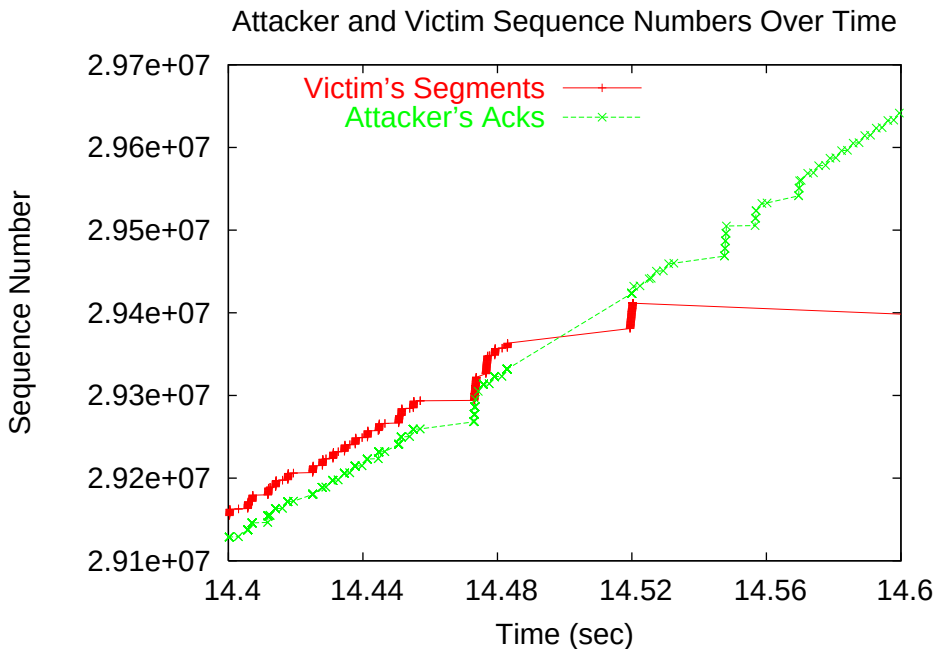


Figure 2.10: Artifact 2: Victim Delay and Buffered ACKs

no longer increase. When an overrun is detected, the attacker can resume slow start on the last received packet. This is an expensive process, as it potentially requires waiting on the order of at least one second [11] for the server to timeout.

Figure 2.6 shows the life cycle of an attack against a GNU/Linux 2.4.20 victim, across a wide area network, as measured at the victim. The "attacker" data points show the ACKs at the time the victim received them, and the "victim" data points show the segments being sent by the victim. Note that for the majority of the time the two lines are indistinguishable, i.e. the attacker is synchronized with the victim (Figure 2.8). However, on three occasions the attacker overruns the victim's sequence number, and is forced to recover, as described above. The attacker blindly continues sending ACKs that are ignored, as the victim stops making progress in sending the stream (as demonstrated by the flat line). In the first overrun, the victim actually retransmits three

times before the attacker recovered, because the retransmitted packets were also lost. However, in the next two overruns, the attacker recovered faster, each on the order of one second.

Recovery code must track the victim's slowstart threshold ($ssthresh$) in addition to the estimated congestion window ($ecwnd$). The variable $ssthresh$ is initialized to the maximum window size, is set to half $ecwnd$ with every recovery, and grows with the congestion window, as prescribed by [145].

2.8.2 Victim's Processing Time

One of the most difficult challenges in keeping the attacker synchronized is estimating the time taken for the victim to send the packets, which we call the processing time. Obviously, an attacker should not ACK segments faster than a victim is capable of generating them. Through experimentation, we find that an upper bound on the processing time of a victim if 50ms (Section 2.5). However, this is a lose bound and in this section, we present techniques for more exactly determining it.

If the attacker knows the victim's processor speed, server load, operating system, and local bandwidth, it may be able to estimate the processing delay time. However, this information is difficult to determine, and underestimating the delay time leads to the attacker getting ahead of the server as well as significant performance degradation. To address this challenge, we introduce the $TargetBandwidth$ variable. With this variable, we can derive the processing delay:

$$processing\ delay\ =\ \frac{\lfloor cwnd/mss \rfloor \times (54 + mss)}{TargetBandwidth}$$

The $TargetBandwidth$ variable represents the rate of traffic the attacker is trying to induce the server to generate (in bytes/second). While the value of $TargetBandwidth$

can be determined adaptively based on how often the attacker is forced to recover, for the purposes of the implementation code, we specify it as a runtime parameter.

The processing time of an idle server is significantly shorter than that of a busy server. This implies that an attacker needs to estimate a server's load before attacking it. However, we noted that as the attacker's flow rate increases, the other connections are forced to back off, which in turn decreases the processing time of the server. Thus, we introduce the concept of adaptive delay. By overestimating the initial processing time and the delay between ACKs, i.e. sending ACKs slowly, and then progressively ramping up the ACK speed to the desired rate, third party streams are "pushed" out of the way with minimal overruns. How to do this effectively in an aggressive manner, without causing the attacker to overrun and restart, is an open question. However, in the implementation, we start arbitrarily at 10 times the estimated processing time, and then decrease down to the target processing time in steps of 500 μs per window.

Another variable affecting the processing time is the coarse grained time slice in the victim's scheduler. Periodically, the victim process is suspended for a number of time slices, which can cause a delay in sending if the kernel buffer is drained before the process can be rescheduled. An example of this is the second artifact (Figure 2.6, blown up as Figure 2.10), where the server actually pauses for 36 ms. Note, it is less obvious from Figure 2.10, but the server starts sending less than one millisecond before the buffered ACKs arrive. We do not have a technique to predict these delays, and rely on the recovery/restart mechanism.

2.8.3 Multiple ACKs Per Window and the Transition Phase

We noted that during congestion avoidance, the server rarely sent a full 64KB window, even when the congestion window would otherwise have allowed for it. The effect was

that the number of segments in flight varied, and it became difficult for the attacker to ACK the correct number of segments. We speculate this is due to operating system buffering inefficiencies, and perhaps coarse grained time slices. Whatever the reason, we changed the attack algorithm to ACK half of the window at a time with the appropriate delay instead of the full window all at once. By ACKing half as much, twice as often, we were able to keep the amount of flooding high, reducing the chance the attacker gets ahead of the victim's sequence number. The downside is that by sending twice as many ACKs, we get only half of the performance listed in Section 2.3.

An additional benefit of sending two ACKs per window is resistance to lost ACKs. The basic algorithm assumes that each ACK successfully reaches the victim, which is obviously not true in general. To maximize this benefit in implementation, we send two ACKs slightly offset from each other twice per window for a total of four ACKs per window. The benefit here is two fold. First, the attacker can now lose three sequential ACKs in a row without overrunning the server. Second, with more ACKs the congestion window grows faster after recovery from overrun. The effect of sending four ACKs per window is we reduce the expected amplification by a factor of four.

It was difficult to track the exact state of the victim's congestion window and $ssthresh$, especially after recovering. It was common for the attacker to stay correctly synchronized with the victim through slow start and then get out of sync immediately when moving to the congestion avoidance algorithm. While we speculate there are many factors that cause this behavior, i.e. unpredictable server load, and the timing involved in the congestion avoidance phase may need to be more accurate than the slow start phase, it simply became easier to work around it. Thus, we introduce a "transition" phase for the attacker between slow start and congestion avoidance (see Figure 2.7). In this transition phase, we ACK every expected packet in turn for the full

window. The effect of the transition phase is that it allows for a larger margin of error in estimating the victim's $ssthresh$ variable. In practice, we ACK two full windows in the transition phase before transitioning to the full congestion avoidance portion of the attack.

2.8.4 The Attacker's Local Bandwidth

Algorithm 1 does not take into account the attacker's local bandwidth. Given a local bandwidth of β in bytes per second, ACKs can be sent at at most $\alpha = \beta/54$ bytes/second. At speeds faster than α, and the ACKs get buffered or even dropped, which interferes with the timing of the attack. When ACKs are buffered (as shown in the first artifact of Figure 2.6, and Figure 2.9))they arrive at the victim all at once. The victim is not able to send fast enough to keep up with the sudden flood of ACKs and this creates an overrun. To fix this, we limit the rate of outgoing ACKs from the attacker as a function of the available local bandwidth, which is specified at runtime. The main effect of rate limiting the ACKs is to maintain even spacing when they arrive at the victim, despite network jitter and buffering.

Algorithm 1 –**Attack**($\{v_1 \ldots v_n\}$, mss, $wscale$)

1: maxwindow$\leftarrow 65535 \times 2^{wscale}$
2: $n \leftarrow |\{v_1, \ldots, v_n\}|$
3: **for** $i \leftarrow 1 \ldots n$ **do**
4: connect(mss,$wscale$) to v_i , get isn$_i$
5: ack$_i \leftarrow$ isn$_i + 1$
6: $w_i \leftarrow$ mss
7: **end for**
8: **for** $i \leftarrow 1 \ldots n$ **do**
9: send v_i data request { http get, ftp fetch, etc. . . }
10: **end for**
11: **while** true **do**
12: **for** $i \leftarrow 1 \ldots n$ **do**
13: ack$_i \leftarrow$ ack$_i + w_i$
14: send ACK for ack$_i$ to v_i { entire window}
15: **if** $w_i <$ maxwindow **then**
16: $w_i \leftarrow w_i +$ mss
17: **end if**
18: **end for**
19: **end while**

Chapter 3

Shared Resources with Cooperative Users: Slurpie

3.1 Introduction

Consider a situation where many Internet hosts all try to simultaneously download a large file from a central server, e.g. when a new CD image or critical patch is released for a popular operating system. As the number of clients increases beyond a critical threshold, the data rate each client receives from the server tends towards zero. When the server is so stressed, the processing and storage resources the server needs to handle client connection state is exhausted, and new clients are denied access to the server. Unfortunately, existing clients do not get adequate service either, since their data connections (using TCP) compete with each other and with new connection requests. Under severe contention on the server access link, the network regresses to congestion collapse and no client is able to make progress. Thus, it is not uncommon for extremely popular downloads to take many (tens of) hours or longer, when uncontested, the file could be downloaded in minutes. It is also not uncommon for the downloads to fail entirely, because the TCP connections either do not get created or time out due to packet losses.

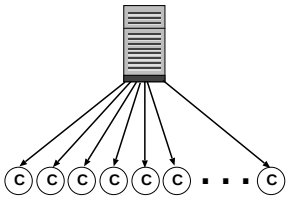


Figure 3.1: Traditional data transfer: all data is transferred from the server.

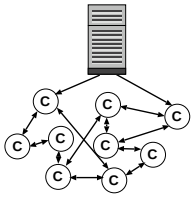


Figure 3.2: Slurpie: Clients form a mesh and most data can be gotten from mesh neighbors.

In this chapter, we present a protocol, named Slurpie[1], to handle this precise problem. Specifically, the goal of Slurpie is to minimize client-side wall clock time taken to download large, popular files. Our work on Slurpie is motivated by the following observation: while the resources, both bandwidth and processing, at the server are completely exhausted, the clients themselves usually have ample spare capacity. Using spare processing and bandwidth on inter-peer paths, Slurpie creates a dynamic peer-to-peer network (Figure 3.2) of clients who want the same file with the goal of reducing client download time and server load. Our design goals for Slurpie were the following:

1. *Scalable*: Slurpie should be scalable and robust: specifically, the protocol should be able to handle very large (10^3–10^6) simultaneous clients. Further, an explicit goal of our protocol is to maintain load at the server *independent* of the number of Slurpie clients. Thus, the entire Slurpie client set should appear as a configurable number of clients at the server, regardless of the size of the Slurpie

[1]Slurpie was originally designed as part of the CS 711 graduate networking course at the Univ. of Maryland.

network.

2. *Beneficial*: Clearly, the first property implies that a properly designed Slurpie protocol will reduce load at the server. However, clients will not use Slurpie unless their own download time is reduced. The second explicit design goal of Slurpie is to minimize the client download times; as we shall see, almost all clients decrease their download times by using Slurpie rather than getting the file directly from the server.

3. *Deployable*: A design goal of Slurpie is the ability to be deployed without infrastructure support. Specifically, we do not require deployment or router co-location of any new dedicated servers and it is reasonably easy for any ad-hoc groups of nodes to start their own instantiation of Slurpie. As described in the protocol description, Slurpie requires a demultiplexing host which it uses to locate other peers; we have designed the protocol such that the load —in terms of processing bandwidth, and state— on this host is minimal.

4. *Adaptive*: Slurpie is designed to adapt to different network conditions, and tailor its download strategy to the amount of available bandwidth and processing capacity at the client.

5. *Compatible*: Lastly, we designed Slurpie such that it requires no server-side changes. In fact, a server that is serving a set of Slurpie clients cannot determine whether these clients are using Slurpie (except for the reduction in server load). Thus, Slurpie can be used with existing data transfer protocols including HTTP and FTP.

Inherent to our solution is the assumption that the server is the data transfer bottleneck, and that clients have additional resources (both processing and bandwidth) that

they are willing to use to decrease their download times. Additionally, we also make the following assumptions:

- Slurpie will be used for bulk data transfer. Thus, latency and jitter are of secondary importance to overall throughput, and clients can receive and process data out of order.

- Users are *not* required to persist in the system after they finish downloading their file. Of course, system performance will increase if benevolent users choose to persist, since they can then serve parts of the the file to new users.

- An end-to-end data integrity check is available out of band. The download protocols we consider, HTTP and FTP, do not provide a cryptographically strong integrity check on transferred data. The concern due to the lack of a check is amplified when parts of the file are received from unknown nodes in the network. We assume that an application-level check is available out of band; note that this is the current norm as most popular downloads are accompanied with a MD5 checksum of the content.

3.1.1 Approach

Cooperative downloads, where the load on the server is mitigated by using other network hosts, have previously been studied and implemented in many forms. These prior efforts fall into three main categories: infrastructure-based solutions such as content-distribution networks (e.g. Akamai [7]) where server providers provision in-network hosts to alleviate load on the central server. The complementary approach is client-deployed cache hierarchies (e.g. Squid [143]) that reduce client access times (and in turn server load). There has been significant work in deploying and choos-

ing mirror servers that replicate content. All of these approaches require fixed investment in infrastructure support and work perfectly well as long as the demand can be anticipated (and hence provisioned for). A new generation of p2p protocols (NICE [17], Narada [36], CAN-multicast [118], Scribe [34], etc.) have been developed for application-layer multicast in which streaming content is replicated and forwarded using the only resources of peers who themselves want this data. The inherent advantage of these schemes is extreme scalability. This is because, these protocols proportionately increase the amount of resources devoted to transferring data as the number of clients who want the data increase. The research focus on application-layer multicast has been on building efficient topologies that provide low end-to-end latencies. Slurpie uses this same paradigm in which peers form a dynamic structure without any extra investment in infrastructure. However, unlike prior work, our focus is on creating an efficient structure for quickly locating and disseminating bulk data. The Slurpie protocol is loosely based on the following schematic:

> Suppose a popular file is available from a heavily loaded web server (called the "source server" in the rest of this chapter). When a node wants to download this file, it registers with a centrally known *topology* server and retrieves a complete list of other nodes downloading the same file. The file is logically divided into fixed sized blocks, and successful completion of the download consists of downloading this set of blocks. The set of nodes downloading the same file form a per file mesh. Update messages of which nodes have which blocks are propagated through the mesh. With the update knowledge, each node can either download a given block from a peer, or from the source server.

The schematic described above is appealing, and has a number of desirable properties (e.g. reduction in server load). However, in practice, a number of problems have to be solved in order to derive a usable solution. For example, the schematic requires the topology server to maintain exact state about all peers downloading a file. Clearly, this will not scale since flash crowds of many tens of thousands can often request the same file within a very small period of time. There are many other practical problems, such as deciding on a "good" number of blocks to divide the files into, and deciding how many connections each peer should open. We also need to decide precisely how the mesh is formed, how updates are propagated, and how a peer decides to approach the server as opposed to downloading a block from the peer network. Finally, any cooperative download protocol must have a good solution for the "last block" problem, where all the nodes in the system have all but one block, and they all try to get the last block from the server! This focus of this chapter is on solving precisely this set of problems, and developing a protocol that meets our stated design goals.

3.1.2 Roadmap

The rest of this chapter is structured as follows: in the next section, we describe prior work, and compare Slurpie to related work. In Section 3.3, we present specifics of the Slurpie protocol. We present experimental results in Section 3.4, discuss deployment issues in Section 3.5, and conclude in Section 3.6. This work was supported in part by NSF CAREER Award ANI 0092806.

3.2 Related Work

The general problem of getting popular content off of heavily loaded servers is well studied. We divide existing approaches down into categories of multicast, infrastructure-based solutions, and existing peer-to-peer efforts. We also discuss the effects of erasure encoding the data transfers.

3.2.1 Multicast

One method of reducing load at a server is to replace a number of unicast streams with one single multicast stream. This can be done either at the IP layer [44], e.g. using cyclic multicast [12], or in the application layer [17, 36, 118, 34]. The main difference between these approaches and Slurpie is that Slurpie incorporates both a *discovery* and a separate data transfer phase, i.e. in Slurpie the decision of where to get the next piece of data is made dynamically depending on network conditions and on which nodes have what data. In contrast, in all multicast-based schemes, the data source is, by default the original server, and alternate paths are used primarily for loss recovery [20, 53, 120]. Slurpie is also designed for bulk data transfer, and downloads blocks in a random order, while a number of the multicast protocols are optimized for streaming. Compared to Slurpie, most multicast protocols are much more careful about creating a topology that approximates a shortest path tree (or some some other good topological property). The Slurpie topology is essentially ad-hoc, and data transfer links are added and kept only for transferring a few blocks. We could potentially incorporate a more sophisticated topology construction algorithm in Slurpie, but Slurpie peers stay in the network for a very short period of time and our main objective in creating the topology is minimizing control overhead, and not

necessarily network-level efficiency. Many (if not most) multicast protocols will not operate well if peers stayed in the network for only a few minutes, as is the norm in Slurpie. Finally, Slurpie provides complete reliability, while for the most part, reliable multicast is still has many difficult open research issues.

3.2.2 Infrastructure-based Solutions

Content distribution networks (CDNs) such as Akamai [7, 19] and web-caching hierarchies [143] are often used to alleviate load on popular servers. CDNs are deployed by the content providers (i.e. the servers), and web-caches are usually deployed by clients. A similar solution employed by some content providers is to employ a fixed number of static content mirrors (e.g. See `http://www.gnu.org/prep/ftp.html` for GNU software mirrors). Regardless of how these mirrors, caches, or CDN nodes are deployed, they are explicitly provisioned for certain load levels, and if a flash crowd exceeds this provisioned amount, then the performance of the system degrades again. In contrast, resources available to Slurpie *increase* as the client set increases, and thus, we believe Slurpie is able to handle larger client sets.

3.2.3 Peer-to-peer Bulk Transfer Protocols

Two peer-to-peer projects, CoopNet and BitTorrent, implement cooperative downloads.

CoopNet

In CoopNet [108], clients get redirect messages from the server to clients that have previously downloaded the same file. Clients are expected to remain in the system for some amount of time after they are finished downloading to serve files to future

clients. The server provides multiple peers in the redirect, and an estimate of the best client is calculated. The server stores the last n (n=5–50 in simulations) clients to have requested the file, and the redirects are useful as long as one of the n clients is still serving the file. All state is stored at the server, and it is assumed that both the client and servers are CoopNet aware.

The intended application of CoopNet is downloading small HTML files, unlike Slurpie which targets bulk data transfer. There is no notion of serving a partially downloaded file, and all data transfers necessarily involve the server (in order to get the redirect list).

BitTorrent

BitTorrent [29, 41] is the work closest to Slurpie, as it targets bulk data transfer and has similar assumptions. A "tracker" service is set up to help peers downloading the same file find each other. A random mesh is formed to propagate announcements, and peers download from as many other peers as they can find. A novel feature of BitTorrent is connection choking. Peer A will stop sending blocks to peer B (this is called "choking" the connection) until peer B sends A a block, or a time out occurs. The choking encourages cooperation, as well as implicitly rate limits the data going out of a loaded peer. It is assumed that a BitTorrent client was started a priori on the web server, and that the client stays in the system indefinitely serving the file. The web server itself serves a file with a ".torrent" extension, which contains both a set of hashes for the files contents, and a URL for the tracker. From the BitTorrent documentation, it is not clear how much state the tracker keeps, but from examining the source, it appears to be $O(n)$, where n is the number of nodes downloading the file.

Compared to Slurpie, BitTorrent does not adapt to varying bandwidth conditions,

or scale its number of neighbors as the group size increases. Each client appears to keep $O(n)$ state, and they periodically reconnect to the tracker to provide update information. The tracker system limits the scalability of the system to the order of thousands of nodes [41]. In Section 3.4, we present performance comparisons that show that Slurpie out performs BitTorrent, with respect to both average download times and also download time variance.

3.2.4 Erasure Encoding

Erasure codes have been used to efficiently transfer bulk data [30, 31]. With modest overhead, they have the benefits of resilience to packet loss and eliminate the need for stateful data transfers.

As pointed out in [30], the limitations of a stateful system, like Slurpie, typically include: lack of data distribution, per connection state, and the "last block" problem. Slurpie explicitly addresses each of these concerns via random block selection, fixed state per node, and backing off from the webserver, respectively. Finally, it is possible to incorporate erasure coding and similar encodings into Slurpie to potentially further improve performance. This is an avenue of future work.

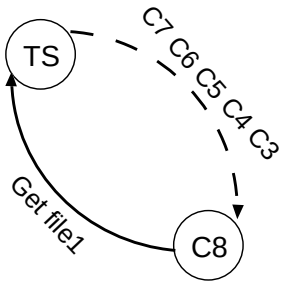


Figure 3.3: Get seed nodes from topology server; topology server keeps constant per file state.

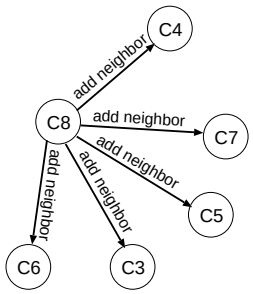


Figure 3.4: Discover alive peers and form mesh; mesh degree depends on number of peers.

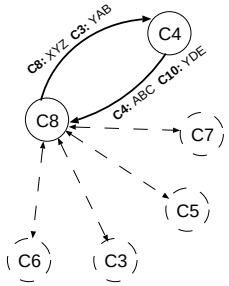


Figure 3.5: Exchange updates with mesh peers; update rate controlled by bw adapatation alg.

3.3 Slurpie: Protocol Details

The Slurpie protocol implements the basic schematic introduced in Section 3.1, but includes a number of refinements that are necessary for proper functioning with large client sets. At a high level, all nodes downloading the same file initially contact a topology server (Figure 3.3). Using information returned by the topology server, the nodes form a random mesh (Figure 3.4), and propagate progress updates to other nodes (Figure 3.5). The updates contain information about which blocks are available where, and this information is used to coordinate the actual data transfer (Figure 3.6). Slurpie

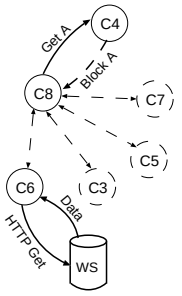


Figure 3.6: Data Transfer. Server visited only if no peer has needed block.

uses an available bandwidth estimation technique, described below, that returns three states: *underutilized*, *throttle-back*, and *at-capacity*. Using this information, the protocol makes informed decisions about the number of edges to keep in the mesh, the rate at which to propagate updates, and the number of simultaneous data connections to keep open. Slurpie coordinates group downloading decisions without global information by employing a number of techniques, such as a random *back off* which controls load at the source server. It is not feasible for Slurpie clients to keep per-peer state for large download groups; we employ a mesh size estimation technique to determine the mesh size using only data stored locally. In the rest of this section, we describe different components of the Slurpie protocol, beginning with the mesh formation.

3.3.1 Mesh Formation and Update Propagation

The join procedure discussed in Section 3.1 did not scale because it assumed that the topology server kept state for the entire set of nodes downloading the same file. However, note that given a single *seed* node downloading the same file, a newly joined node can receive updates from that seed, and use the update messages to discover new peers and add new edges in the mesh. Thus, the topology server only needs to maintain information about a single node that is currently downloading a file (instead

of all nodes that are downloading the file). But the question then becomes: which node id. does the topology server store, and how does it guarantee that the node is still in the system? In Slurpie, we always return the identity of the *last* node to query the topology server (for that same file). The intuition is that the node that most recently started downloading a file is the node that most likely to be still in the system. In practice, the topology server maintains and returns the last ψ nodes, where ψ is a small constant. Note that this procedure is identical to the mesh joining procedure in Narada [36].

Given a set of seed nodes, the newly joined node makes bi-directional "neighbor" links to a random subset of these nodes. Each node has a target number of neighbors (η) that it seeks to maintain. The value of η is continually updated depending on available bandwidth, and as new neighbors are discovered. The bandwidth estimation algorithm is run once a second, and if it consistently returns *underutilized*, a new neighbor, picked uniformly at random from the set of known peers, is added. In general, each node tries to maintain $\eta \geq O(\log n)$, where n is the estimated size of the total number of nodes in the mesh. Since the mesh is, at a first approximation, a random graph, the $O(\log n)$ degree implies that the mesh stays connected with high probability [23].

Update Propagation

Along each neighbor link, update messages of the form $\langle$ IP-addr, port, block-list, hopcount, node-degree $\rangle$ are passed. These form the basic information units that alert peers of new nodes joining the system, and of who has which blocks. The rate of updates passed along each link per second, σ, is subject to an AIMD flow control algorithm [72, 74] which additively increases and multiplicatively decreases update

rates depending on available bandwidth estimates. The intuition behind controlling the update rate in this manner is the following: when a node does not have enough peers to download from to fill its bandwidth capacity, it should increase its knowledge of the world (and thus increase the rate at which it receives updates). Correspondingly, as the node's bandwidth becomes consumed with useful data downloads, information about other peers becomes less useful.

The Update Tree In Slurpie updates, the block list is simply represented as a bit vector. There are certainly a number of more sophisticated data structures, e.g. Bloom filters [22] and approximate reconciliation trees [30] that we could use, but for our purposes a simple bit vector has been sufficient.

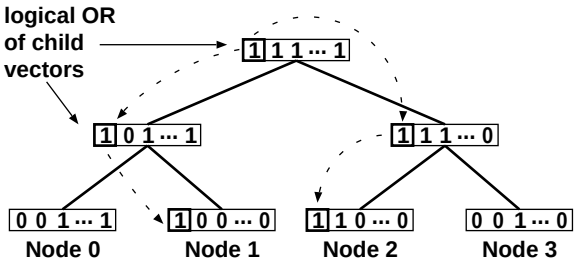


Figure 3.7: Update Tree: nodes with block zero are highlighted

Each node stores information about U other nodes, where U is a constant chosen locally. The bit vectors within an update are locally stored in a data structure known as the *update tree* (see Figure 3.7). Bit vectors corresponding to individual nodes form the leaves of the tree, each parent is a bit vector of the logical OR of its children, and the root of the tree is the logical OR of all updates. This structure can then be used to efficiently answer queries of the form "which blocks have not been retrieved from the web server", and "which set of machines has downloaded this specific block". Only a single bit vector is stored for any peer, and newer vectors from a peer (with more bits

set) replace any existing vectors from this peer. The least hop count for a given node id is also saved; this approximates the shortest path to the node, and is used in estimating the mesh size (described next).

3.3.2 Group Size Estimation

A number of the algorithms that Slurpie uses assumes that we know n, the total number of nodes downloading a given file, so it is important to be able to accurately estimate that number. Recall that U is the number of updates that any node stores. If $n \leq U$, then as time progresses and updates propagate, each node receives information about every other node in the system, and can very accurately estimate n. However, the case where $n > U$ is more interesting.

We know from random graph theory that for an r-regular graph, the mean distance d between nodes is proportional to $log_{r-1}n$. Solving this equation for n, we get $n = O((r-1)^d)$. The mesh formed by Slurpie is not exactly an r-regular graph, as nodes have different numbers of edges, and it is impossible for a single node to know the exact distance counts to all nodes in the system when $n > U$. However, using the U updates in the update tree, it is possible to estimate averages for both hop counts and degrees to gain estimates for d and r, and thus an estimate for n. Note that such an estimate becomes more accurate as n increases. In Section 3.4, we show that in simulations, this approximation provides reasonable estimates for n, even for relatively small values of U.

3.3.3 Downloading Decisions

In Slurpie, blocks served by peers are downloaded before blocks served by the source server. When multiple peers have the same block, we choose a peer uniformly at

random. In an effort to take advantage of an open TCP window, once a connection to a peer has been established, the node downloads any blocks that it does not have from that peer.

In general, multiple downloading connections are opened in parallel, and it is a non-trivial question to decide how many connections is optimal. Here, Slurpie again makes use of the bandwidth estimation algorithm. The algorithm is queried every second, and if it returns *underutilized*, and there exist hosts that have blocks that the local node does not have, a new connection is opened.

3.3.4 Backing Off

Slurpie nodes only connect to the server if they have excess capacity, and know of no other peers that can provide them useful data blocks. Recall, however, that a design goal of the Slurpie protocol is to control the load on the source server independent of the number of peers in the Slurpie mesh. We ensure this constant load property by employing a random backoff, and in effect, system throughput *increases* as peers do *not* go to the server, even if the server is the only node that has a block they need. This is because if a large enough set of nodes opened simultaneous connections to the server for even a single block, none of the nodes would get their data, and overall system throughput would tend to zero.

Ideally, the host with the best connection to the server would be the sole machine connected to the server, and everyone else would receive their data from this host. There are, however, two problems with this method:

- The best host could download the data and then leave the system, and the entire process would have to repeat again; and
- Finding the best host is probably difficult, especially since this has to be deter-

mined quickly, dynamically, and without server support, and without probing the server (path).

Instead, we use the following scheme: Every time period τ, each eligible peer decides to go to the server with probability k/n where n is the estimate of the nodes in the system, and k is a small constant. The effect is that, on average, there will be k connections from the Slurpie mesh to the server at any time, and the number of connections to the server over time is exactly modeled by a binomial distribution with mean k. Intuitively, $k = 1$ is optimal, as it is closest to the ideal on average. However, setting $k = 1$ is too pessimistic, and results in *no* connections at the server for extended periods of time (about 30% of the time). In practice, we choose $k = 3$, which assuming $k << n$ implies there is at least one connection at the server about 90% of the time.

If we view this backoff scheme as essentially time division multiplexing, then the parameter τ becomes the length of the time slice. Logically, τ should be chosen to be long enough to guarantee some amount of progress, but short enough to ensure some amount of fairness. In this way, even a set of hosts with diverse bandwidth resources can make progress, as statistically over long downloads all hosts will eventually fetch some blocks from the server.

3.3.5 Block Size

The number of blocks a file is divided into presents a trade off between download parallelism and overhead. A small number of blocks is more efficient since it allows TCP connection overheads to be amortized, but smaller blocks reduce parallelism. As number of blocks increases, the size of the bit vector and the Slurpie control overhead increases. Instead of picking the number of blocks, we choose a fixed block size,

256KB, and let the number of blocks vary with the size of the file. We chose 256KB after conducting experiments on an unloaded system with different block sizes. A 256KB block was the smallest size at which the TCP overhead was effectively amortized ($< 1\%$). Further, the 256KB block size keeps the bit vector to a manageable size for large files (50 bytes for a 100MB file). It is worth noting that modern HTTP and FTP servers support downloading blocks of arbitrary sizes and offsets. HTTP 1.1 implements this functionality via the *Range* tag, and FTP can be made to simulate this behavior with the restart (*REST*) command.

3.3.6 Bandwidth Estimation Technique

Slurpie requires that the bandwidth estimation algorithm only report three different states: *underutilized*, *at-capacity*, and *throttle-back*. The main design criterion of our bandwidth estimation algorithm is efficiency: Slurpie peers cannot use expensive probes [33, 82, 69] to determine precise bandwidth usage or availability. Instead, the following simplistic approach suffices: we assume that the user inputs a coarse grained bandwidth estimate of the form "Modem", "T1/DSL", "T3", etc... that forms the initial maximum bandwidth estimate B_{max}. Next, we measure the sum of actual achieved throughput over all data connections over a 1 second interval, and label that B_{act}. We maintain a moving average of successive B_{act} values, calculating an *average* throughput, and the standard deviation *std* of that distribution. Using these numbers, if B_{act} drops more than one standard deviation than the average, we report **throttle-back**. If B_{act} is more than one standard deviation less than B_{max}, we report **underutilized**, else we report **at-capacity**. If at any time $B_{act} > B_{max}$, we set $B_{max} = B_{act}$.

3.4 Experiments

In this section, we present results from our implementation of Slurpie, and compare against existing protocols. We begin with a description of our implementation (Section 3.4.1), and describe our experimental setup next.

3.4.1 Slurpie Implementation

Slurpie has been implemented in multi-threaded C on the GNU/Linux system. It currently has a command line interface similar to the popular program *wget* [155], taking a URL and various options as parameters. The source code is available from the Slurpie sourceforge project[134], and should be portable to a number of platforms.

3.4.2 Experimental Setup

We experimented with Slurpie on two different networks: one on the local area network the other on the wide-area network. We used a 48-node local testbed for runs where we could precisely control the background traffic. These experiments were useful to precisely quantify Slurpie overheads and benefits, and also to compare Slurpie against BitTorrent in a predictable environment. We also deployed both Slurpie and BitTorrent on the PlanetLab wide-area testbed.

Local Testbed Setup

The testbed that was setup consisted of an Apache 2.0.45 web server running on an unused Linux machine with a 2.4.20 version kernel. The machine was connected to a 10Mb hub, and the the hub to a 100Mb switch, so as to force a 10Mb bottleneck at the server. The clients consisted of 48 GNU/Linux machines with 100Mb connections to

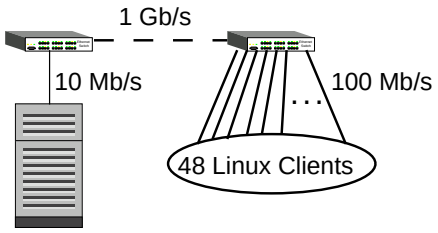


Figure 3.8: Local area testbed setup. The server is connected using a 10Mbps link to force a bottleneck.

a separate 100Mb switch, and the two switches were connected by a series of gigabit Ethernet links, as shown in Figure 3.8. Each client machine was a 650Mhz Pentium III with 768MB of RAM. In each experiment, a 100MB file was downloaded from of the web server by variable numbers of clients concurrently. The 10Mb hub is important, as by assumption, it is the server, not the client, that is the bottleneck.

PlanetLab Setup

We ran Slurpie on the PlanetLab[114] wide area network. PlanetLab consists of 55+ different sites, and 160+ different machines distributed geographically around the world. The same web server was used from the local area network tests, but with different network connectivity to the clients. From the 100MB switch connected to the web server, there is a 1Gb/s link to machines participating in Internet2, and a 95Mb/s link to machines on the general Internet. A list of machines was retrieved from the PlanetLab website, and one machine per site was chosen at random.

3.4.3 BitTorrent Setup

To compare Slurpie's performance to a comparable protocol, we downloaded the most current version of BitTorrent (version 3.2.1). To facilitate scripting, all experiments

were done using Bit Torrent's "headless" mode, as opposed to GUI or Curses. BitTorrent's normal mode of operation is not to terminate after finishing downloading the file, but instead to persist indefinitely. For our experiments, we modified the BitTorrent code to terminate clients after a configurable wait after the file download is complete. In all experiments, both Slurpie and BitTorrent clients persist for the same amount of time after each experiment.

3.4.4 Results

In the results that follow, unless otherwise stated, we use the parameters listed in Table 3.1. By default, for experiments with concurrent clients, each successive client is started 3 seconds after the previous one. (We also present results in which all clients start simultaneously). In all of the experiments, we consider the following performance metrics: total completion time and server load. The first determines client benefit from using Slurpie, and the second quantifies the benefit to the server. Finally, we present simulation results that show how our network size estimation algorithms perform.

Parameter	Description	Value
k	k/n clients go to server	3
τ	Server connection length	4 seconds
σ	Initial Update Rate	8/second
η	Initial Number of Neighbors	10
m	Mirror Time (described below)	2 seconds
U	Number of Updates Stored	100
ψ	Per File State at Topology Server	5

Table 3.1: Default Slurpie Parameters

Local Testbed Results

First, we compute a baseline measure by measuring the time for a *single* client to download the 100MB file uncontested using HTTP. The baseline was measured 5 times, and the average value was used. It was assumed that all machines would have the same baseline. In the first experiment, we vary the number of concurrent clients that download the 100MB file from the server. In Figure 3.9, we plot the completion time for plain HTTP, BitTorrent, and Slurpie as a function of the pre-computed baseline time. For example, with 48 concurrent clients, each client, on average received only 2% of their baseline bandwidth with plain HTTP. The performance was restored to 88% with BitTorrent, and improved to 1.76 times the baseline with Slurpie. Each data point is the average measurement across active clients and then averaged across 10 runs.

As expected, these results clearly show how performance deteriorates with plain HTTP as files gain popularity. In our experiments, the BitTorrent protocol restores performance to essentially the baseline. For the vast majority of clients using Slurpie, performance *increases* as the number of peers in the network increases (recall that in these experiments, we require clients to persist only for 2 seconds after they have downloaded the entire file). Overall, this is an encouraging result indeed, and as we show later in this section, clients that join the network late are able to download the entire file at their own maximum download rate, regardless of the server capacity.

In Figure 3.10, we plot the cumulative distribution of the completion times of clients from the 48 concurrent node runs. Once again, each data point is an average of 10 runs. Compared to BitTorrent, Slurpie decreases average download time by 51%; more importantly, Slurpie provides more consistent performance, and the worst Slurpie client (which is the first client that joined) completes more than 5.4 times faster than the worst BitTorrent client.

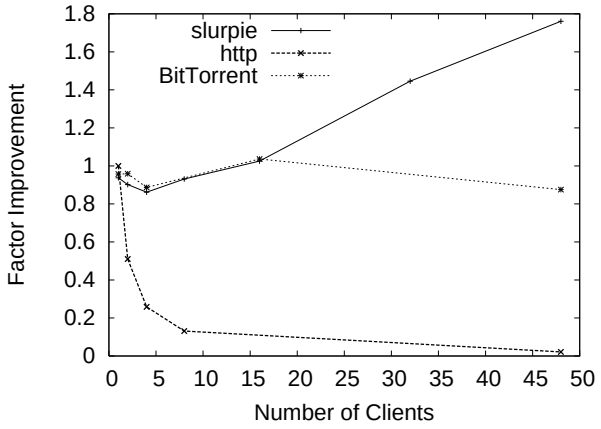


Figure 3.9: Normalized completion time for varying number of clients

To understand the steady-state dynamic of Slurpie better, we conducted a different experiment in which 245 clients joined the network, once again separated by 3 seconds each. In Figure 3.11, we plot the completion times of these clients. The x-axis is ordered by the order of the clients' arrival times into the system. The horizontal line is the baseline completion time (i.e. the amount of time a single client takes to download the file using plain HTTP, if there are no other clients in the system). There are several points to note: the first few clients take longer than the baseline — this is because they have to download the data mostly from the server, and pay for Slurpie overheads as well. However, once the file permeates the Slurpie mesh, the vast majority of clients get the file 2–4 times faster. There is an interesting periodic behavior evident in the completion times. This is because once the complete file is downloaded into the Slurpie network, it is distributed quickly using the mesh. However, soon clients who have the complete file leave the network (2 seconds after their download is complete), and some blocks have to be fetched from the server. This slows down completion time for a few clients who have to wait for the slow source download. However, as soon

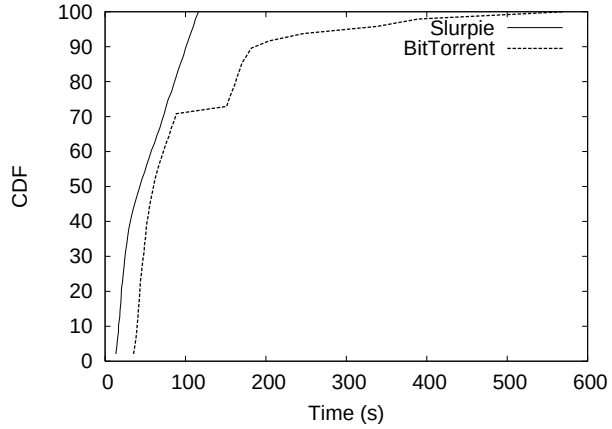


Figure 3.10: CDF of completion times, 48 concurrent nodes

as these blocks reappear in the Slurpie network, performance increases back up until these nodes leave the network and the cycle repeats. The periodic behavior is mitigated if clients persist longer in the network.

3.4.5 PlanetLab Results

We repeated the same experiment over the wide-area PlanetLab testbed. In Figure 3.12, we present the normalized completion times of varying numbers of clients using both BitTorrent and Slurpie. Once again, Slurpie outperforms BitTorrent across the client set, and our results show that both the average and maximum time taken by Slurpie is better than BitTorrent in all runs. Note that as the number of clients increases, the relative performance with respect to the baseline reduces somewhat on the PlanetLab testbed (whereas on our local area network, the relative performance *increases*). This is because the PlanetLab hosts were being rather heavily used during the period we conducted our tests, and many of the hosts do not have much excess capacity for down-

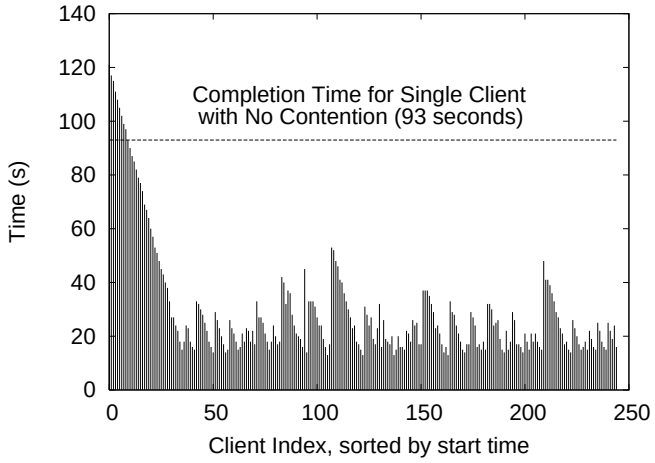


Figure 3.11: Absolute completion times, 250 nodes

loading faster from peers. Thus, as the client set increases, the number of clients with extra resources decreases as a proportion, and the average with respect to the baseline also decreases. We believe the PlanetLab hosts are uncommonly loaded compared to most Internet hosts, and in a "real" deployment, Slurpie performance would indeed increase with larger client sets.

Mirror Time

In Slurpie, we do not require nodes to persist in the system after they finish downloading their file. It is nevertheless interesting to study the effects of benevolence, i.e. consider how completion times decrease as users stay longer after completing their download. In Figure 3.13, we plot completion times (again normalized against the baseline completion time), for 48 concurrent users, as users persist in the system. Interestingly, for Slurpie, almost all benefits of such mirroring is achieved if users stay in the system for only 3 extra seconds. For much larger files, we expect this number

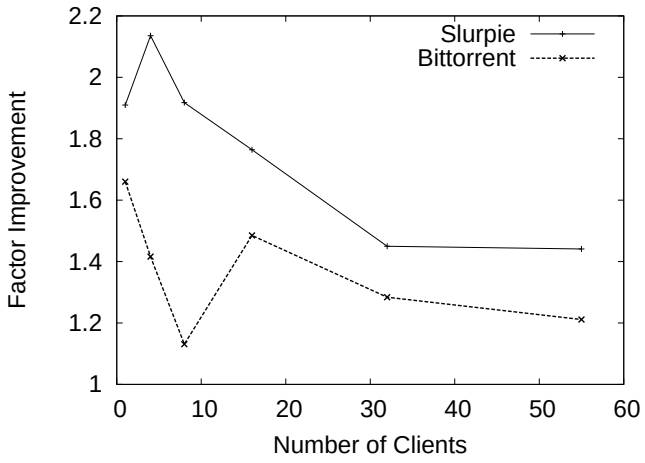


Figure 3.12: Normalized completion time vs. number of clients on the PlanetLab

to increase, but it is clear even nominal amounts of benevolence leads to substantial benefit.

3.4.6 Coordinated Backoff

The most novel component of Slurpie is its coordinated backoff algorithm. In this section, we show how performance increases as the number of clients that go to the server is carefully controlled. In Figure 3.14, we plot the number of connections at the server with 48 concurrent clients with and without the backoff algorithm enabled. Without backoff, clients eventually all go the server together because some blocks are not available in the Slurpie network. The backoff algorithm carefully controls the number of clients that visit the server, and on average, the Slurpie network maintains the expected number of connections (3) to the server. Note that the number of connections drops off around 100 seconds because almost all clients complete their download by that time. As expected, the backoff algorithm controls server load. Client-side performance is

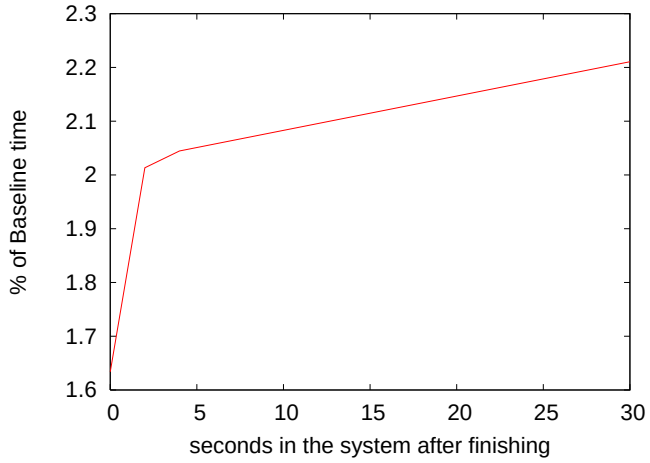


Figure 3.13: Normalized completion time vs. mirror time

also improved (Figure 3.15). Specifically, without backoff, the Slurpie protocol is not able to ultimately gain from the larger numbers of nodes in the network. A closer look at our data shows that without backoff, the clients all quickly download almost all blocks, and than all visit the server for a few (sometimes just one) blocks. However, since the server is heavily loaded, all benefits from having received the other blocks quickly is negated.

Effects of Flash Crowds

In our previous experiments, we start concurrent clients 3 seconds apart deterministically. We have also experimented with random offsets between clients. However, in the worst case, all clients would start exactly at the same time. In Figure 3.16, we plot the number of open connections at the web server over time as the number of clients on the LAN that start *at the same time* is varied from 10–48.

Recall that a client tries to estimate the number of nodes in the mesh n, and tries to

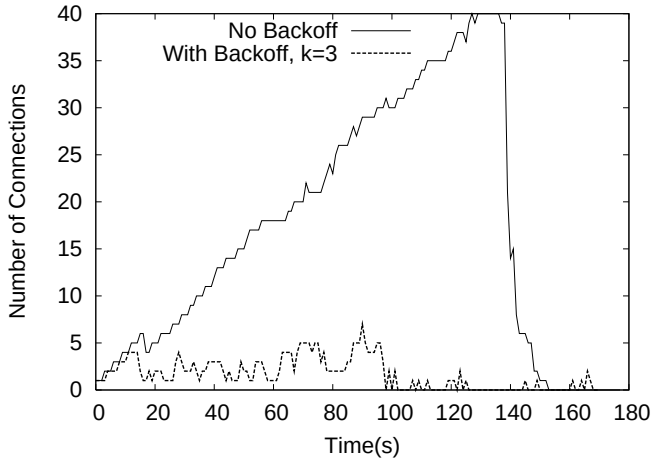


Figure 3.14: Number of Connections at the server, over time

connect to the server with probability k/n, where k is set to 4. The y-axis in the plot is set to the same scale as Figure 3.14. Recall that in that experiment, without backoff, even with clients started 3-seconds apart, the number of simultaneous connections increased to more than 40. In Figure 3.16, there are different curves for 10, 20, 32, and 48 simultaneous connections, but it is difficult to distinguish these cases. Thus, the Slurpie size estimation algorithm is effective: server load is independent of the Slurpie mesh size. We note that 48 clients arriving at exactly the same time is indicative of severe congestion (several thousand new connections per second), and Slurpie is able to easily contend with such load spikes.

3.4.7 Group Size Estimation

In an effort to gauge the quality of the group size estimation, we simulated the neighbor mesh algorithm with large group sizes. The simulator took three parameters, n, the number of nodes in the system, r, the target degree of each node, and U, the number

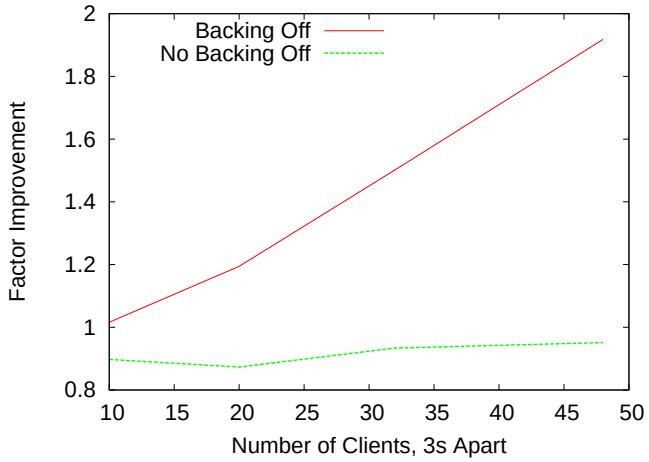


Figure 3.15: Performance effects of the back off algorithm

of updates stored. Then, using the formula described in Section 3.3, the simulation returned n', the average estimate of the system size. We present results in Table 3.2 for meshes with target out degree fixed at 10. The estimation error levels decrease as the state per node increases, and as the number of nodes in the system increases. This is because the estimation is derived from an asymptotic formula which provides better bounds with larger group sizes. Note that in almost all realistic scenarios, we do not expect to use the estimation with less than 1000 nodes in the system (with 1000 nodes, each client has to keep a maximum of 6MB of update state for a 100MB file). Finally, note that the backoff algorithm does not require very precise estimations of group size, e.g. estimating n with $\pm 33.3\%$ error and $k = 3$ will, on average, result in $\pm$ one extra connection to the source server.

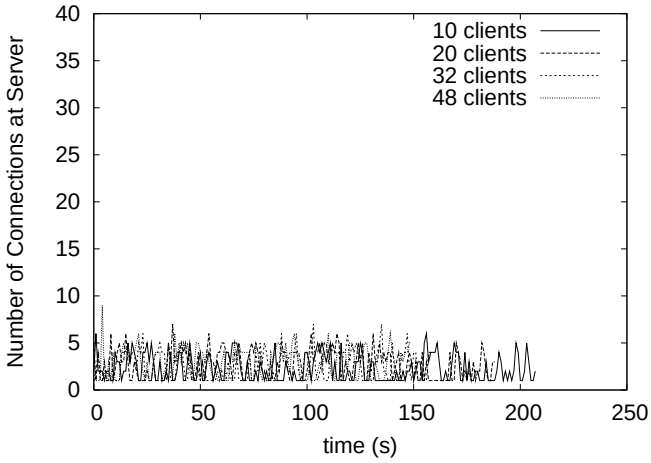


Figure 3.16: Number of connections at server with different numbers of clients, all started simultaneously

3.5 Discussion

In the results section, we have concentrated entirely on the data transfer dynamics of Slurpie. In this section, we discuss the implementation and deployment of the two other components: the topology server and security issues.

3.5.1 Topology Server

The topology server in Slurpie serves the same purpose as the rendezvous point in Narada [36] or the BSE in the NICE [17] protocol. One possible concern is the scalability of the Slurpie topology server: a scalable network does no good if clients cannot join because the server required for joining is overloaded! In practice, the topology server stores the IP address and port of the last five nodes to request a given file. This amounts to state of 30 bytes per file plus the file name, so any reasonable machine can

n	20	50	100
200	17.5%	13.2%	10.9%
1000	5.9%	4.2 %	2.6%
5000	11.3%	7.5%	6.0%
10000	3.8%	0.8 %	0.4%

Table 3.2: % Error in Group Size Estimation

store state for millions of files. Since the server performs no client-specific processing, the processing requirements at the server are minimal.

Of more concern is the network overhead at the topology server. Upon joining the system, every node makes a TCP connection to the topology server, tells it which file they are downloading, and then receives the IP address/port pairs of the last 5 nodes to download that same file. The entire transaction uses one packet in either direction, plus TCP overhead, so it is conceivable for a single server to handle tens of thousands of downloads per second. If the Slurpie system grows to the point where this is insufficient, the topology server functionality could be distributed. Specifically, a number of hosts that provide this service could form a DHT [146, 117, 125], and the file name could be used to look up the server responsible for the specific file. However, we do not believe the scalability of the topology server will be the limiting factor in the deployment of a system such as Slurpie.

3.5.2 Security Concerns

Using Slurpie introduces potentially new security and data integrity concerns for end hosts. In the best case, Slurpie clients will download almost all parts of files from unknown nodes on the Internet. However, we argue that this does not add significantly

new security risks. A security integrity check should be performed for sensitive files, even if it is downloaded from the source server. As we mentioned in Section 3.1, servers often publish an MD5 or similar checksum which is used to verify file integrity. Such a checksum could be used by Slurpie clients as well. It is possible for a determined adversary to attack the Slurpie network by propagating both false blocks and a corresponding *false checksum*. Note that this is a problem even in the source download case, since a determined adversary can mount any number of attacks that base IP is susceptible to, including DNS spoofing or TCP connection hijacking. The solution, of course, is to distribute a signed integrity check, where the clients can independently verify the checksum since it is signed by a trusted server. Such a solution requires an out-of-band channel by which clients get the server's public key, and once implemented, is sufficient for both plain IP and for Slurpie.

Another potential problem is a DoS attack against the Slurpie topology server. If the topology server does not function, new nodes cannot join the network. Once again, we believe this is a general problem and not specific to Slurpie, and the solutions are no different from the ones that can be employed to protect any source server.

3.6 Conclusions and Future Work

In this chapter, we have described the Slurpie protocol for scalable downloading of bulk data over the Internet. We believe the Slurpie protocol fulfills its design goals of system scalability, improved client performance, and insulation of the server from load variance in the client population. We have presented extensive experimental analysis of different components of the Slurpie protocol using a complete implementation over a local- and wide-area testbed. Specifically, our results show that client performance

increases as the client population increases. This is because clients can now download parts of files from other clients without accessing highly contested server resources. Further, our results indicate that the Slurpie randomized back off scheme is effective, and is able to precisely control server load regardless of the size or variation in client population.

There are a number of interesting open issues with Slurpie design. One issue is that a cooperative system like Slurpie is only useful if many people use it, i.e. no benefits are gained from using Slurpie if only one Slurpie client is downloading a given file. One possible path to encourage Slurpie usage is to make the topology server aware of well known mirror sites (e.g. `ftp.gnu.org` has many mirrors), and in the joining phase, can communicate multiple mirrors to the client. Then, the client can adaptively download in parallel from multiple source servers if there are not many Slurpie peers in the system. A nice property of this type of system would be increased performance independent of the number of users in the system. Also, we believe it is possible to implement better estimates of the network size, especially if the underlying graph structure of the Slurpie mesh was studied in more detail. One problem with the current interface is it is insufficient for mass deployment, since it requires users to explicitly invoke the Slurpie protocol to download popular files. An obvious extension is to deploy a Slurpie proxy that intercepts all user download requests, and automatically routes requests for popular files to a Slurpie network. A number of the contributions of this work are independent of the data transfer path, so another avenue of research might be to implement Slurpie's data transfer using more sophisticated encoding schemes, e.g. erasure codes.

It is also worth considering schemes where (possibly with a small amount of server side assistance), clients can quickly tell whether a particular block they have down-

loaded is corrupt or not. It is trivial to implement such a scheme with $O(\#blocks)$ overhead, but it is not clear if an asymptotically better scheme is feasible. Lastly, our evaluation was constrained to fifty node testbeds. While this is a good beginning, evaluation on larger networks would obviously provide more compelling evidence.

Chapter 4

Shared Resources with Selfish Users: NICE Cookies

4.1 Introduction

NICE[1] is a platform for implementing *cooperative* applications over the Internet. We define a cooperative application as one that allocates a subset of its resources, typically processing, bandwidth, and storage, for use by other peers in the application. We believe a large class of applications, including on-line media streaming applications, multi-party conferencing applications, and emerging peer-to-peer applications, can all significantly benefit from a cooperative infrastructure. However, cooperative systems perform best if all users do, in fact, cooperate and provide their share of resources to the system. In this chapter, we present techniques for identifying cooperative and non-cooperative users. Using our schemes, individual users can assign and infer "trust" values for other users. The inferred trust values represent how likely a user considers other users to be cooperative, and are used to price resources in the NICE system.

We focus on distributed solutions for the trust inference problem. We decompose the distributed trust inference problem into two parts: a local trust inference compo-

[1] NICE is a recursive acronym for "NICE is the Internet Cooperative Environment" (See `http://www.cs.umd.edu/projects/nice`).

nent that requires trust information between principals in the system as input and a distributed search component that efficiently gathers this individual trust information to be used as input for local inference algorithms. There already exist systems, e.g., e-bay[151], that have a centralized user-evaluation system. Our goal is to enable open applications where users do not have to register with an authority to be a part of the system. Centralized solutions do not scale in open systems, since malicious users can overwhelm the central "trust" server with spurious transactions. The most widely used decentralized trust inference scheme is probably the PGP web of trust [163], which allows one level of inference. We present a new decentralized trust inference scheme that can be used to infer across arbitrary levels of trust. There is no trusted-third-party or centralized repository of trust information in our scheme. Users in our system only store information they explicitly can use for their own benefit. We show that our algorithms scale well even with limited amount of storage at each node, and can be used to efficiently implement large distributed applications without involving explicit authorities. Further, our solutions allow individual users to compute local trust values for other users using their own inference algorithm of choice, and thus can be used to implement a variety of different policies.

4.1.1 Cooperative Systems

The notion of a cooperative system is not unique in networking; in fact, packet forwarding in the Internet is a cooperative venture that utilizes shared resources at routers. Our overall goal in NICE is to extend this notion to include end-applications and provide an incentive-based framework for implementing large distributed applications in a cooperative manner. Clearly, an immense amount of distributed resources can be harvested over the Internet in a cooperative manner. This observation is key in the recent

surge of peer-to-peer (p2p) applications, and we believe the next generation of such p2p applications will be based upon the notions of cooperative distributed resource sharing.

A number of interesting distributed algorithms for p2p systems, most notably in the area of distributed resource location, have recently been introduced. All of these schemes, however, assume that all peers in the system implicitly cooperate and implement the underlying protocols perfectly, even though it may not be explicitly beneficial to do so. Consider the following examples:

- In Gnutella [60], peers forward queries flooded on behalf of other users in the system. Each forwarded message consumes bandwidth and processing at each node it visits.

- In Chord [146], a document is "mapped" to a particular node using a hash function. Thus, a peer serves a document that is, in fact, owned by some other node in the system. Thus, peers in the system expend their own resources to serve documents for other nodes in the system. This situation is not unique to Chord; all hash-based location systems, including CAN [117], Bayeux [162], Pastry [125], have this property. It is possible to build a system in which nodes only serve a pointer to the document data and also to implement various load balancing schemes; however, even in the best load-balanced system, there can be temporary overloads when a large amount of local resources are expended due to external serving.

- A number of relay-based streaming media protocols have been developed and demonstrated. In these protocols, nodes devote resources such as access bandwidth for serving their child nodes.

In each example above, any individual user may choose *not* to devote local resources to external requests, and still get full benefit from the system. On the other hand, the integrity and correct functioning of the system depends on each user implementing the entire distributed protocol correctly and selflessly. However, experience with deployed systems, such as Gnutella and previously Napster, show that only a small subset of peers offer such selfless service to the community, while the vast majority of users use the services offered by this generous minority [5]. The goal of this work is to efficiently locate the generous minority, and form a clique of users all of whom offer local services to the community.

4.1.2 Model

In this chapter, we assume that a (p2p) system can be decomposed into a set of of two-party transactions. A single transaction can be a relatively light-weight operation such as forwarding a Gnutella query or a potentially resource intensive operation such as hosting a Chord document. Next, we assume that the system consists of a set of "good" nodes that always implement the underlying protocols correctly and entirely, i.e. good users always fulfill their end of a transaction. The goal of our work is develop algorithms that will allow "good" users to identify other "good" users, and thus, enable *robust* cooperative groups. These are peer groups in which, with high probability, each participant successfully completes their end of each transaction. Specifically, we propose a family of distributed algorithms which can be used by users to calculate a per-user "trust" value. The trust value for node B at a node A is a measure of how likely node A believes a transaction with node B will be successful. In our system, users store a limited amount of information about how much other users trust them, and we present algorithms for choosing what information to store and how to retrieve

this trust information. Once relevant information has been gathered, individual users may use different local inference algorithms to compute trust values.

It is important to note that we assume that good nodes are able to ascertain when a transaction is successful. Clearly, in many cases, it is not possible to efficiently determine whether a transaction fails (e.g. when a node sometimes does *not* serve Chord documents that it hosts). It is even more difficult to determine whether a transaction fails because of a system failure or because of non-cooperative users. For example, consider the case when all users are cooperative but a document cannot be served due to a network failure. We believe this problem is inherent in any trust-inference system that is based on transaction "quality". We discuss different policies for assigning values to transaction quality in Section 4.4.

The overall goal of this work is to identify cooperative users. An ideal trust inference system would, in one pass, be able to classify all users into cooperative or non-cooperative classes with no errors. However, this is not possible in practice because non-cooperative users may start out as cooperative users. The specific goals of our work are as follows:

- Let the "good" nodes find each other quickly and efficiently: Good nodes should be able to locate other good nodes without losing a large amount of resources interacting with malicious nodes. This will allow NICE to rapidly form robust cooperative groups.

- Malicious nodes and cliques should not be able to break up cooperating groups by spreading mis-information to good nodes. Specifically, we want to develop protocols in which malicious nodes are rapidly pruned out of cooperative groups. Further, we assume malicious nodes can disseminate arbitrary trust information, and the cliques formed of good nodes should be robust against this form of

attack.

In Section 4.4, we describe algorithms that achieve our goals with low run-time overhead, both in terms of processing and network bandwidth usage. We believe this algorithm is the first practical, robust, trust inference scheme that can be used to implement large cooperative applications.

The rest of this chapter is structured as follows: in the next section, we discuss prior work in distributed trust computations. In Section 4.3, we present an overview of the NICE system, and describe how distributed trust computations are used by NICE nodes. We describe our algorithms and local node policies in Section 4.4, present simulation results of the trust search in Section 4.5, and simulated results of a NICE resource trading system in Section 4.6. We discuss our conclusions in Section 4.7.

4.2 Related Work

In this section, we discuss prior work in trust inference and present a brief overview of systems that are based on notions of trust and incentive.

The concept of "trust" in distributed systems is formalized in [97] using social properties of trust. This work considers an agent's own experience to obtain [-1, 1]-valued trust, but does not infer trust across agents. Abdul-Rahman et al. [1] describe a trust model that deals with direct experience and reputational information. This model can be used, as is, in NICE to infer trust. Yu et al. [158] propose a way to compute a real-valued trust in [-1, 1] range from direct interactions with other agents. A product of trust values is used for reputation computation, and undesirable agents are avoided by having an observer of bad transactions disseminate information about the bad agent throughout the network. This work is primarily about using social mechanisms for

regulating users in electronic communities, and the techniques developed here can be used in NICE. In this chapter, we focus on algorithms for efficiently storing and locating trust information.

Another scheme [2] focuses on management and retrieval of trust-related data, and uses a single p2p distributed database which stores complaints about individuals if transactions with them are not satisfactory. When an agent p wants to evaluate trust for another agent q, it sends a query for complaint data which involves q, and decides q's trustworthiness with returned data, using a proposed formula. However, this system implicitly assumes that all participants are equally willing to share the communal data load, which may not be true in many p2p systems [5]. Such a system is also vulnerable to DoS attacks, as there is no preventative measure from inserting arbitrary amounts of complaints into the system.

PGP [163] is another distributed trust model that focuses on proving the identity of key holders. PGP uses user defined thresholds to decide whether a given key is trusted or not, and different introducers can be trusted at finite set of different trust levels. Unlike NICE, trust in PGP is only followed through one level of indirection; i.e. if A is trying to decide the trust of B, there can be at most one person, C, in the trust path between A and B. There are also a number of popular web sites, e.g. e-bay and Advogato (see `www.advogato.org`) that use trust models to serve their users. However, all data for these sites is stored at a trusted centralized database, which may not be ideal for open systems, and lead to the usual issues of scalability and single point of failure.

The Eigentrust [77] system focuses on taking pairwise trust values, i.e. the trust $c_{i,j}$ for all pairs i, j, and attempts to calculate a single global trust value for each principal. It accomplishes this task by computing in a distributed manner the principal eigenvec-

tor of the entire pair-wise trust matrix. In the current Eigentrust system, when node i and node j have not interacted, $c_{i,j}$ is assumed to be zero. The protocol presented in this chapter is complementary to Eigentrust, and can be used to infer unknown $c_{i,j}$ values.

A system similar to NICE is Samsara[43]. While NICE attempts to solve distributed resource allocation problems in resource neutral manner, Samsara focuses strictly on remote file storage. In Samsara, nodes exchange chunks of objects, and query each other to verify that nodes are correctly storing the objects they claim to. If a particular node fails a query, other nodes in the system begin to probabilistically drop the node's objects. The probability of a given object being dropped increases as the number of consecutive failed queries increases. The argument is made that it is not possible with current bandwidth limitations for a malicious node to replace dropped objects fast enough to counteract the dropping rate. However, it is not clear how Samsara's bandwidth restriction defense fairs against a more sophisticated attacker that employs forward error correction (FEC) to entangle multiple objects. Samsara does not have a notion of trust inference.

4.3 Overview of NICE

In this section, we present a brief overview of the NICE platform. Our goal is to provide context for the distributed trust computation algorithms presented in Section 4.4. NICE is a platform for implementing cooperative distributed applications. Applications in NICE gain access to remote resources by bartering local resources. Transactions in NICE consist of secure exchanges of resource *certificates*. These certificates can be redeemed for the named (remote) resources. Non-cooperative users may gain

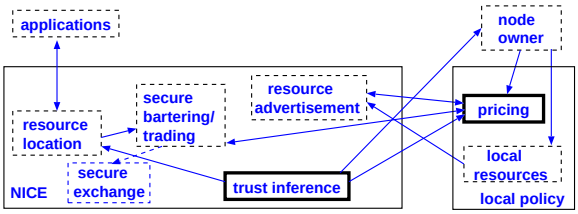


Figure 4.1: NICE component architecture: the arrows show information flow in the system; each NICE component also communicates with peers on different nodes. In this chapter, we describe the trust inference and pricing components of NICE.

"free" access to remote resources by issuing certificates that they do not redeem.

NICE provides a service API to end-applications, and is layered between the transport and application protocols. The NICE component architecture is presented in Figure 4.1, with this chapter's contributions in bold. Applications interact with NICE using the NICE API, and issue calls to find appropriate resources. All of the bartering, trading, and redeeming protocols are implemented within NICE and are not exposed to the application. These sub-protocols share information within themselves and are controlled by the user using per-node policies. NICE peers are arranged into a signaling topology using our application-layer multicast protocol [18]. All NICE protocol-specific messages are sent using direct unicast or are multicast over this signaling topology. The exact details of the remaining sub-protocols, e.g., bartering, resource location, secure exchange, and the NICE API itself remain the subject of future work. For the purpose of building and inferring trust, we treat resources and transactions as abstractions throughout the chapter.

4.3.1 NICE Users and Pricing Policies

Until now, we have used the terms user and node in an imprecise manner. In NICE, each user generates a PGP style [163] identifier which includes a plaintext identification string and a public key. The key associated with a NICE identifier is used for signing resource certificates, for trading resources, and for assigning trust (Section 4.4) values.

It is important to note that neither the NICE identifier nor the associated key needs to be registered at any central authority; thus, even though NICE uses public keys, we do not require any form of a global PKI. Thus, NICE can be used to implement open p2p applications without any centralized authority. Since there is no central registration authority in NICE, a single user can generate an arbitrary number of keys and personas. However, in NICE, pricing is coupled with identity, i.e., *new users have to pay more for services* until they establish trust in the system. Thus, it is advantageous for nodes to maintain a single key per user and not to change keys frequently. This property makes NICE applications robust against a number of cheap identity based denial-of-service attacks that are possible on other p2p systems[57].

The goal of the default policies in NICE is to limit the resources that can be consumed by cliques of malicious users. These policies work in conjunction with the trust computation which is used to identify the misbehaving nodes. In practice, NICE users may use any particular policy, and may even try to maximize the amount of resources they gain by trading their own resources. The primary goal of the default policies is to allow good users to efficiently form cooperating groups, and not lose large amounts of resources to malicious users. The pricing and trading policies are used to guard against users who issue spurious resource certificates using multiple NICE identities. We use two mechanisms to protect the integrity of the group:

- **Trust-based pricing**

  In trust-based pricing, resources are priced proportional to mutually perceived trust. Assume trust values range between 0 and 1, and consider the first transaction between Alice and Bob where the inferred trust value from Alice to Bob is $T_{Alice}(Bob) = 0.5$, and $T_{Bob}(Alice) = 1.0$. Under trust-based pricing, Alice will only barter with Bob if Bob offers significantly more resources than he gets back in return. Note however that as Bob conducts more successful transactions with Alice, the cost disparity will decrease. This policy is motivated by the observation that as Alice trades with a principal with lower trust she incurs a greater risk of not receiving services in return, which, in turn, is reflected in the pricing.

- **Trust-based trading limits**

  In these policies, instead of varying the price of the resource, the policy varies the *amount* of the resource that Alice trades. For example, in an scenario with Alice and Bob, Alice may trade some small, trivial amount of resources with Bob initially, but once Bob has proven himself trustworthy, Alice increases the amount of resources traded. This policy assures that when trading with a principal with relatively low trust, Alice bounds the amount of resources she can lose. The simulated results presented in Section 4.6 use trust based trading limits.

4.4 Distributed Trust Computation

We assume that for each exchange of resources, i.e., a *transaction*, in the system, each involved user produces a signed statement (called a *cookie*) about the quality of the transaction. For example, consider a successful transaction t between users Alice and Bob in which Alice consumes a set of resources from Bob. After the transaction

completes, Alice signs a cookie c stating that she had successfully completed the transaction t with Bob. Bob may choose to store this cookie c signed by Alice, which he can later use to prove his trustworthiness to other users, including Alice [2]. As the system progresses, each transaction creates new cookies which are stored by different users. Clearly, cookies have to be expired or otherwise discarded; the algorithms we present later in this section require constant storage space.

We will describe the trust inference algorithms in terms of a directed, weighted graph T called the trust graph. The vertices in T correspond exactly to the users in the system. There is an edge directed from Alice to Bob if and only if Bob holds a cookie from Alice. The value of the Alice→Bob edge denotes how much Alice trusts Bob and depends on the set of Alice's cookies Bob holds. Note that each transaction in the system can either add a new directed edge in the trust graph, or relabel the value of an existing edge with its new trust value.

Assume that a current version of the trust graph T is available to Alice, and suppose Alice wishes to compute a trust value for Bob. If Alice and Bob have had prior transactions, then Alice can just look up the value of Alice→Bob edge in T. However, suppose Alice and Bob have never had a prior transaction. Alice could potentially *infer* a trust value for Bob by following directed paths (ending at Bob) on the trust graph as we describe below.

[2]It is also possible for Alice to keep a record of this transaction instead of Bob. In this alternate model of trust information storage, users themselves store information about whom they trust, and can locally compute the trust of the remote nodes they know of. This model, however, is susceptible to a denial of service attack that we describe later in this section.

4.4.1 Inferring Trust on the Trust Graph

Consider a directed path $A_0 \rightarrow A_1 \rightarrow \ldots \rightarrow A_k$ on T. Each successive pair of users have had direct transactions with each other, and the edge values are a measure of how much A_i trusts A_{i+1}. Given such a path, A_0 could infer a number of plausible trust values for A_k, including the minimum value of any edge on the path or the product of the trust values along the path; we call these inferred trust values the *strength* of the $A_0 \rightarrow A_k$ path. The inference problem is somewhat more difficult than computing strengths of trust paths since there can be multiple paths between two nodes, and these paths may share vertices or edges. Centralized trust inference is not the focus of our work, but it is important to use a robust inference algorithm. We have experimented with different inference schemes, and we describe two simple but robust schemes. In the following description, we assume A (Alice) has access to the trust graph, and wants to infer a trust value for B (Bob):

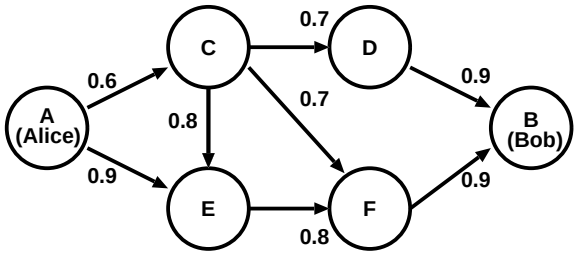


Figure 4.2: Example trust graph: the directed edges represent how much the source of edge trusts the sink.

- **Strongest path:** Given a set of paths between Alice and Bob, Alice chooses the *strongest* path, and uses the minimum trust value on the path as the trust value for Bob. The strength of a path can be computed as the minimum valued edge

along the path or the product of all edges along a path. Given the trust graph, this trust metric can easily be computed using depth-first search. In the example shown in Figure 4.2, we use the min. function to compute the strength of a path. In this example, the strongest path is $AEFB$, and Alice infers a trust level of 0.8 for Bob.

- **Weighted average of strongest disjoint paths:** Instead of choosing only the strongest path, Alice could choose to use contributions from all disjoint paths. The set of disjoint paths is not unique, but the set of strongest disjoint paths (modulo equi-strength paths) is and can be computed using network flows with flow restrictions on vertices. Given the set of disjoint paths, Alice can compute a trust value for Bob by computing the weighted average of the strength of all of the strongest disjoint paths. The weight assigned to the Alice→X→ ... →Bob path is the value of the Alice→X edge (which represents how much Alice directly trusts X). In the example in Figure 4.2, $ACDB$ is the other disjoint path (with strength 0.6), and the inferred trust value from Alice to Bob is 0.72.

Both these algorithms are robust in the sense that no edge value is used more than once and trust values computed are always upper-bounded by the minimum trust on a path. Before any of these local algorithms can be used, the trust graph has to be realized in a scalable manner, and (edge) values have to be assigned to cookies.

Also note that the trust graph is not necessarily connected. For example, if Alice is new to the system, then there exists no path from her to Bob in the graph. In this case, we must have some application specific policy to assign a *default* level of trust. This policy is highly application specific, for example the level of trust for an unknown node in a file sharing application should be higher than the default trust in a medical professional referral service. Additionally, if we assume that each user maintains more

than $\ln N$ cookies, where N is the number of users in the system and that the trust graph approximates an r-regular random graph[23], then with high probability the random graph remains connected. While a random graph is not necessarily a good model for the trust graph itself, we use it to motivate our simulation parameter selections, and show in Section 4.5 that it performs well.

Note that in order to infer trust for Bob, Alice does not need to access the entire trust graph, but only needs the set of paths from her to Bob. In the rest of this section, we describe schemes to store the trust graph and to produce sets of paths between users in a completely decentralized manner over an untrusted infrastructure. We begin with a discussion of different techniques for assigning cookie values, and describe our distributed path discovery protocol in Section 4.4.3.

4.4.2 Assigning Values to Cookies

Ideally, after each transaction, it would be possible to assign a real number in the [0,1] real-valued interval to the quality of a transaction and assign this as the cookie value. In some cases, transactions can be structured such that this indeed is possible: e.g. assume that Alice transcodes and serves a 400Kbps video stream to Bob at 128Kbps, and according to a prior agreement, Bob signs over a cookie of value 0.75 to Alice. The same transaction may have resulted in a cookie of value 0.9 if Alice had been able to serve the stream at 256Kbps. In many cases, however, it is not clear how to assign real-valued quality metrics to transactions. For example, in the previous example, Alice could claim that she did serve the stream at 256Kbps, while network congestion on Bob's access link caused the eventual degradation of the quality to 128Kbps. It is, in fact, easy to construct cases when it is not easily feasible to check the quality of service. In most cases, however, we believe it is somewhat easier to assign a $\{0,1\}$

value to a transaction, i.e. either the transaction was successful, or it was not. As applied to the previous example, Bob and Alice could negotiate a threshold rate (say 64Kbps) at which point he considers the entire transaction successful, and assigns a 1-valued cookie to Alice, regardless of whether the data was delivered at 64.5Kbps or 400Kbps. Further, for many transactions, such as streaming media delivery, it is possible for one party to abort the transaction if the initial service quality is not beyond the 0-value threshold.

In the rest of this chapter , we assume that cookies are assigned values on the [0,1] interval. However, it is possible to assign arbitrary labels to cookies, and to conduct arbitrary policy-based searches as long as the requisite state is kept at each user. For example, it is possible to construct a system where cookies take one of four values (e.g., "Excellent", "Good", "Fair", and "Poor"), and users search for "Excellent"-valued cookies that are less than one week old. All of the NICE path enumeration and inference schemes work correctly as long as cookies have a comparable value, regardless of how users assign these values, and what range these values take.

One issue is who decides cookie values: by direct user intervention or by program. There is a trade-off in the two approaches between the latency in assigning values versus the accuracy of values. Cookies values that are assigned automatically, e.g., by heuristic or rule, do not have to wait for an interactive user before being posted to the trust network. However, in many cases it is possible to trick a heuristic into giving good cookie values for marginal service. For example, Alice offers Bob 10GB of disk space for one hour in exchange for one hour of computing time on Bob's machine. Bob takes the disk and allows Alice to use his computer, but runs additional jobs on his computer such that Alice's job is slowed. An automated cookie valuation might give a high valued cookie for this transaction, but a human valuation would catch

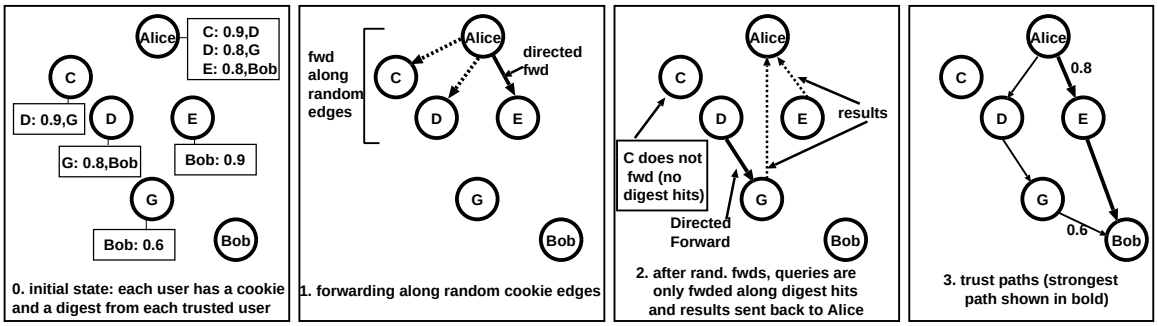


Figure 4.3: Different stages in the operation of the Alice→Bob search protocol. Edges in this figure represent message flow. It is important to note that corresponding edges in the trust graph point in the *opposite* direction.

Bob's cheating, and return a low valued cookie.

4.4.3 Distributed Trust Inference: Basic Algorithm

In this section, we describe how users locate trust information about other users in our system. This distributed algorithm proceeds as follows: each user stores a set of signed cookies that it receives as a result of previous transactions. Suppose Alice wants to use some resources at Bob's node. There are two possibilities: either Alice already has cookies from Bob, or Alice and Bob have not had any transactions yet.[3] In the case Alice already has cookies from Bob, she presents these to Bob. Bob can verify that these indeed are his cookies since he has signed them. Given the cookies, Bob can now compute a trust value for Alice.

The more interesting case is when Alice has no cookies from Bob. In this case,

[3] There is yet a third possibility in which Alice has discarded cookies from Bob, but we assume that this case is equivalent to Alice having no cookies from Bob

Alice initiates a search for Bob's cookies at nodes from whom she holds cookies. Suppose Alice has a cookie from Carol, and Carol has a cookie from Bob. Carol gives Alice a copy of her cookie from Bob, and Alice then presents two cookies to Bob: one from Bob to Carol, and one from Carol to Alice. Thus, in effect, Alice tells Bob, "You don't know me, but you trust Carol and she trusts me!" In general, Alice can construct multiple such "cookie paths" by *recursively* searching through her neighbors. In effect, Alice floods queries for Bob's cookies along the cookie edges that terminate at each node, starting with her own node. After the search is over, she can present Bob with an union of directed paths which all start at Bob and end at Alice. Note that these cookie paths correspond exactly to the union of directed edges on the trust graph which we used for centralized trust inference. Thus, given this set of cookies, Bob can use any centralized scheme to infer a trust value for Alice.

This basic scheme has several desirable properties:

- If Alice wants to use resources at Bob, *she* has to search for Bob's cookies. This is in contrast with the analogous scheme in which nodes themselves keep records of their previous transactions. Under such a setting, if Bob did not know Alice, *he* would have to initiate a search for Alice through nodes he trusted. A malicious user Eve could mount an easy denial-of-service attack by continuously asking other nodes to search for Eve's credentials. In our system, nodes forward queries on behalf of other nodes only if they have assigned them a cookie, and thus, implicitly trust them to a certain extent.

- Alice stores cookies which are statements of the form "X trusts Alice". Thus, *Alice only devotes storage to items that she can use explicitly for her own benefit*, and thus, there is a built-in incentive in the system to store cookies. In fact, if Bob assigns a low-value cookie for Alice, she can discard this cookie since this

is, in effect, a statement that says Bob does *not* trust Alice. In general, users store the cookies most beneficial to their own cause, and do not forward messages on behalf of users they do not trust.

- The transaction record storage in the system is completely distributed, and if two nodes conduct a large number of spurious transactions, only they may choose to hold on to the resultant state. In contrast, in a centralized transaction store, these nodes could easily mount a denial-of-service attack by overwhelming the transaction store with spurious transaction records.

Note that a malicious node may choose to drop or corrupt trust queries sent to it. However, as the network evolves, trust lookup queries are sent primarily to nodes that are *already* trusted, so dropping a trust lookup query is less common. A possible modification to the protocol is to change the trust lookups from a recursive process to an iterative process. Specifically, Alice queries Cathy for any of Bob's cookies, Cathy returns a cookie for Doug, which Alice then uses to query Doug for Bob's cookies, iteratively. However, this iterative protocol has higher network overhead, so we do not consider it further.

4.4.4 Refinements

While the flooding-based scheme we have described is guaranteed to find all paths between users and has other desirable properties, it is not a complete solution. Flooding queries is an inefficient usage of distributed resources, and as pointed out before, malicious nodes can erase all information of their misdeeds simply by throwing away any low valued cookies they receive. We next describe three refinements to the basic scheme that address these issues.

Efficient Searching

The recursive flooding procedure described above does find all cookies that exist for a given principal. However, it is extremely inefficient, since it visits an exponentially growing number of nodes at each level. Further, unless the flooding is somehow curtailed, e.g., by using duplicate suppression or by using a time-to-live field in queries, some searches may circulate in the system forever.

It is obvious to consider using a peer-to-peer search structure, such as a distributed hash table (DHT) [146, 125], to locate cookies. However, this is not possible since in NICE because we do not assume the existence of anything more sophisticated than plain unicast forwarding. NICE is the base platform over which other protocols, such as Chord, can be implemented. The NICE protocols are much like routing protocols on the Internet: they cannot assume the existence of routing tables etc., and must be robust against packet loss and in the case of NICE, against malicious nodes. Thus, we must employ other mechanisms to make the cookie searches more efficient.

Instead of flooding to all neighbors in the trust graph, nodes forward queries to a random subset of their neighbors (typically of size 5). However, the resulting search still increases exponentially at each hop, only with a smaller base! Thus, additionally we add the following extension to our base protocol: whenever node receives a cookie from some other node, it also receives a *digest* of all other cookies at the remote node. Since, in our implementations, the number of cookies at each node is quite small (typically around 40 for a 2048 node system), this digest can be encoded using around less than 1000 *bits* in a Bloom filter[4] [22]. Thus, the storage space required for the digests are trivial (around 128 bytes), but they allow us to *direct* the search for specific cookies with very high precision. The idea of using digests for searches has been used

[4]Such a filter, with only eight hash functions, would have a false positive rate of 3.16×10^{-5}.

previously, e.g., in lookaround caching [21] and summary caches [48]. It is, in fact, a base case of probabilistic search using attenuated Bloom filters [22]; in our experiments, we found that we did not need to use full attenuated Bloom filters — only one level of filters was sufficient. Lastly, each node also keeps a digest of recently executed searches and uses this digest to suppress duplicate queries.

In our implementation, when choosing nodes to forward to, we always choose nodes whose digests indicate they have the cookie for which we are searching. However, it is possible that there are no hits in any digest at a node; in this case, we once again choose nodes to forward to uniformly at random. However, we only forward to randomly chosen nodes if the query is within a pre-determined number of hops away from the query source. Thus, in the final version of the search, a query spreads from the source, possibly choosing nodes at random, but the flooding is quickly stopped unless there is a hit in a next-hop digest.

Note that for most applications, the cost of keeping the digests fresh will be small. With respect to Alice, digests are only kept at nodes that hold a cookie from her. Thus, when Alice updates her local cookie cache, she need only contact nodes that are one hop away from her on the trust graph. Rather than force Alice to maintain a list of nodes who hold her cookies, nodes could periodically contact Alice to refresh their digests.

Example Before we describe other extensions to the base protocol, we illustrate the digest-based search procedure with an example (corresponding to Figure 4.3). Alice wants to use resources Bob has, but does not have a cookie from him. She initiates a search for a cookie path to Bob. In Figure 4.3-0, we show the initial state of cookies and digests at each user, e.g., Alice has a cookie of value 0.9 from C, and her digest from C shows that C has a cookie from D. For this example, we assume the search

out-degree is 3, and the random flooding hop limit is 1. Alice first sends a query not only to nodes with a digest hit (e.g. E), but also to random nodes (e.g. C and D) as illustrated in Figure 4.3-1. After receiving the query, E finds Bob's cookie and returns the query to Alice. When C receives the query, he finds that none of his neighbors have a digest hit for Bob, so does not forward the query further. On the other hand, D does forward the query to G (Figure 4.3-2) who has a digest hit for Bob, and G returns the query to Alice with the cookie she received from Bob. Figure 4.3.3 shows two paths Alice finds, with the strongest path in bold.

Negative Cookies

A major flaw with the original scheme is that low-valued transactions are potentially not recorded in the system. Consider the following scenario: Eve uses a set of Alice's resources, but does not provide the negotiated resources she promised. In our original scheme, Alice would sign over a low-valued cookie to Eve. Eve would have no incentive to keep this cookie and would promptly discard it, thus erasing any record of her misdeed.

Instead, Alice creates this cookie and stores it *herself*. It is in Alice's interest to hold on to this cookie; at the very least, she will not trust Eve again as long as she has this cookie. However, these "negative cookies" can also be used by users who trust Alice. Suppose Eve next wants to interact with Bob. Before Bob accepts a transaction with Eve, he can initiate a search for Eve's negative cookies. This search proceeds as follows: it follows high trust edges out of Bob and terminates when it reaches a negative cookie for Eve. In effect, the search returns a list of people whom Bob trusts who have had negative transactions with Eve in the past. If Bob discovers a sufficient set of negative cookies for Eve, he can choose to disregard Eve's credentials, and not go

through with her proposed transaction. It is important to note that Bob only initiates a negative cookie search when Eve produces a sufficient credible set of credentials; otherwise, Bob is subject to a denial of service attack where he continuously searches for bad cookies. For efficiency, if Eve presents a cookie directly from Bob, Bob need only do a local search for negative cookies. For example, if Bob issued Eve a high valued cookie, and Eve later cheats Bob, Bob would then store a negative cookie for Eve locally. Afterward, when Eve presents Bob with the original high-valued cookie, Bob need only check his local cache for a negative cookie about Eve before rejecting the transaction.

In our implementation, we keep a set of digests for negative cookies as well, but perform Bloom filter-directed searches for these negative cookies only on neighboring nodes.

Preference Lists

In order to discover potentially "good" nodes efficiently, each user keeps a *preference list*. Intuitively, Alice's preference list contains nodes that she has yet to contact, but which have a higher probability of being good than purely random nodes. Nodes are added to a preference list as follows: suppose Alice conducts a successful cookie search for Bob, and let P be the cookie path that is discovered between Alice and Bob. If the transaction with Bob goes well, Alice adds all users in P who have very high trust value (1.0 in our implementation) to her preference list. Alice knows that these nodes have a higher probability of being good than purely random nodes, because she now trusts Bob, and Bob trusts them, creating an implicit trust path. Obviously, only users for whom Alice does not have transaction records are added to her preference list.

In summary, the NICE distributed trust valuation algorithm works as follows:

> *Nodes that request resources present their credentials to the resource owner. Each credential is a signed set of certificates which originate at the resource owner. Depending on the set of credentials, the resource owner may choose to conduct a reference search. The trust ultimately computed is a function of both the credentials, and of the references.*

There are a number of other pragmatic issues pertaining to cookies that we address in NICE, e.g., cookie revocations, and cookie time limits. Specifically, cookies issued from departed nodes no longer have value to the system. Additionally, nodes may change the behavior over time, so cookies should reflect their most current behavior. To address these concerns, cookies are created with an expiration timestamp, and nodes periodically flush expired cookies. The length of the timestamp is application-dependent, and creates a trade off between lookup efficiency versus data freshness. Finally, if a node loses its private key, the only effect is that useless cookies will persist in the network until they expire.

4.5 Results

We present simulations from different sets of experiments. In the first set (this section), we analyze the scalability and robustness of our inference scheme. In these experiments, our goal is to understand how well the cookies work, without regard to how a real system might use cookies. For example, in these experiments, "good" nodes continually interact with unknown nodes, even when they already know of a large set of other good nodes. While this assumption helps demonstrate how cookies propagate and nodes discover each other in the system, in a real system, we expect good nodes

to find a set of other nodes they trust and interact with these trusted nodes much more heavily. We examine this further in section 4.6, we present experiments in which cookies are used to establish trust, but then nodes follow a more conventional pattern, and try to interact with their trusted peers with higher frequency.

Experiments with the search algorithm In the rest of this section, we present results from our simulations of the trust inference algorithm proposed in Section 4.4. In all our results, we use the minimum cookie value as path strength, and use the highest valued path strength as the inferred trust between users. We have experimented with other functions as well, and the results from this simple inference function are representative. Each search carries with it the minimum acceptable strength, and searches stop if no cookies of the minimum acceptable value are present at the current node. Using the minimum cookie value as the strength measure (instead of product of cookie values) consumes up to an order of magnitude more resources in the network and represents a worst-case scenario for our schemes.

We divide our results into two parts. First, we analyze the cost of running the path search algorithm in terms of storage and run time overhead. The storage cost is entirely due to the caching of positive and negative cookies; the run-time overhead comes from the number of nodes that are visited by each query, and the computation cost for forwarding a query. The computation cost of forwarding each query is negligible: we have to generate random numbers, compute eight MD5 hash functions, and check eight bits in a 1000-bit Bloom filter. In these experiments, the digests were assumed to be always fresh. We did not simulate updating of the digest, but we believe a periodic soft-state refreshing algorithm will work adequately. The main overhead of the search algorithm comes in terms of the number of messages sent and number of nodes visited. The bandwidth consumed by the searches is proportional to the number of nodes

visited, and we report this metric in the results that follow. In the second part of our results (Section 4.5.2), we show that our trust inference schemes do indeed form robust cooperative groups, even in large systems with large malicious cliques and with small fractions of good nodes. We begin with an analysis of the scalability and overhead of our path searches.

4.5.1 Scalability

In this first set of results, we simulate a stable system consisting of *only* good users. Thus, we assume that all users implement the entire search protocol correctly. Before the simulations begin, we populate the cookie cache of each user with cookies from other users chosen uniformly at random. Each query starts at a node s chosen uniformly at random and specifies a search for cookies of another node t chosen uniformly at random. In the next section, we will show how long the system takes to converge starting from no cookies in the system, and how robust groups are formed when there are malicious users in the system.

In our first experiment, we fix the number of good cookies at each user to 40. The cookie values are exponentially distributed between [0,1], with a mean of 0.7[5]. Note that all nodes with constant number of cookies is a *worst case* for searching performance. If cookies were distributed unevenly, for example, with a Zipf distribution, then the average diameter of the trust graph would reduce, and the lookup times would decrease [47].

We conducted 500 different searches for cookies of value at least 0.85, where the search out-degree at each node is set to 5. In Figure 4.4, we plot the average success

[5]It is not clear how cookie values should be distributed. We have also experimented with uniformly distributed cookie values with similar results.

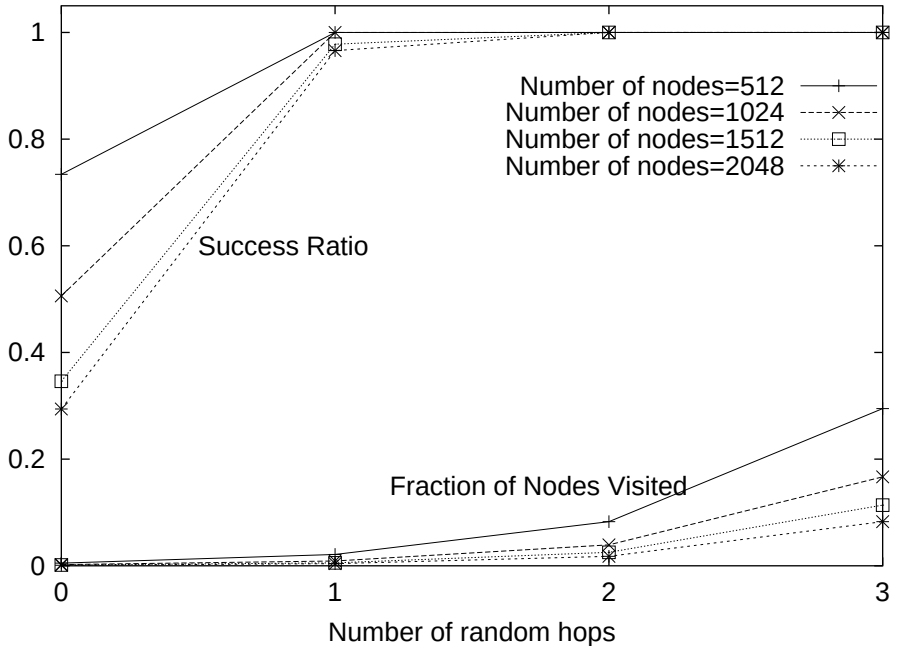


Figure 4.4: Success ratio and no. of nodes visited (40 cookies at each node).

ratio and the average fraction of nodes in the system visited by the searches. The x-axis in the plot corresponds to the number of hops after which random forwards were not allowed, and the search proceeded only if there was a hit in a Bloom filter. There are four curves in the figure, each corresponding to a different system size, ranging from 512 users to 2048 users. From the figure, it is clear that only one hop of random searching is enough to satisfy the vast majority of queries, even with large system sizes. It is interesting to note that even when the system size increases, the average *number of nodes* visited remain relatively constant. For example, the average number of nodes visited with 2 hop random searches range from 42.4 (for a 512 node system) to 36.2 (for the 2048 node system). Thus, the search scheme scales well with increasing system size. As we show next, the success ratio and the number of nodes visited depend almost entirely on the number of cookies held at each node, and the

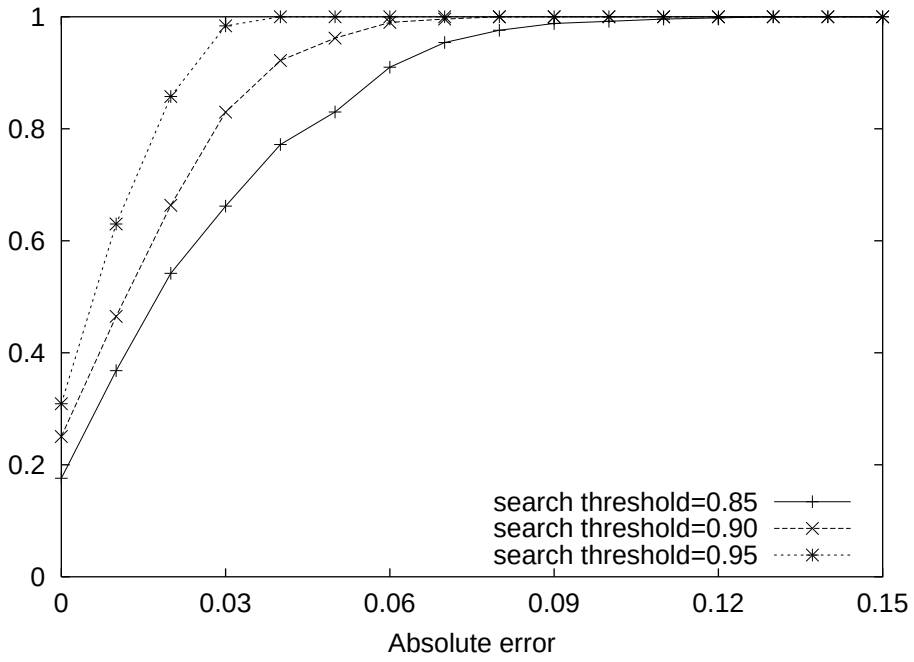


Figure 4.5: CDF of errors versus oracle (40 cookies at each node, out-degree set to 5) with varying thresholds.

out-degree of each search.

In Table 4.1, we fix the number of nodes to 2048 and show the effect of changing the search out-degree. Each row shows searches corresponding to a different minimum threshold ranging from 0.8 to 0.95. Each node holds 40 cookies, the average cookie value is still fixed at 0.7, and the number of random hops is set to 2. In the table, $\#N$ is the average number of nodes visited by a query and $\#P$ denotes the number of paths found on average. As expected, the number of nodes visited increases as the search threshold decreases, and also as the out-degree increases. In all cases, as the search threshold increases, the number of distinct paths found decreases. In Table 4.2, we show the effects of changing the number of cookies at each node. These experiments were conducted using the same parameters, except the out-degree was fixed at 5. With

	K=3		K = 5		K = 7		K = 20	
thrsh.	# N	# P	# N	# P	# N	# P	# N	# P
.8	14.5	4.2	37.5	10.7	71.1	20	499.1	132.5
.85	14.6	3.6	36.2	8.7	68.7	16.5	380.6	88.3
.9	14.6	2.9	35.1	6.8	66.0	12.8	222.9	41.5
.95	15.7	2.0	33.2	4.2	55.9	7	89.2	11.2

Table 4.1: Effect of changing out-degree (K): N, P=nodes, paths traversed

	C=20		C=40		C=102	
thresh.	# N	# P	# N	# P	# N	# P
0.8	34.4	2.9	37.5	10.7	32.8	23.6
0.85	34.7	2.4	36.2	8.7	37.3	25.4
0.9	34.7	1.9	35.1	6.8	40.8	25.4
0.95	28.6	1.4	33.2	4.2	41.9	21.5

Table 4.2: Effect of changing number of cookies stored (C)

small numbers of cookies and high thresholds, searches do result in no paths being found. In Table 4.2, the 0.9 and 0.95 threshold searches had 10% and 42% unsuccesful queries respectively; all other searches returned at least one acceptable path. In our simulator, when a search returns no acceptable paths, we retry the search once more with a different random seed. The numbers of nodes visited in the results above include visits during the retries and account for why the number of nodes visited does not decrease when the search threshold is increased.

In our system, there is a clear trade-off between how much state individual nodes store (number of cookies) and the overhead of each search (fraction of nodes visited).

Note that unlike in systems such as [2], users in our system do not benefit by storing fewer cookies since this effectively *decreases* their own expected trust at other nodes. There is a built-in incentive for users to store more cookies, which, in turn, increases search efficiency. Users may choose to store a large number of cookies but not forward searches on behalf of others. We comment on this issue when we discuss different models of malicious behavior in the next section. Lastly, we note that it is possible to further increase the efficiency of the searches by adjusting the two search parameters — out-degree and number of random hops — based on the threshold and results found. Such a scheme will minimize the number of nodes visited for "easy" searches (low search threshold) and find better results for searches with high thresholds. We have not implemented this extension yet.

The previous two results have shown that the number of cookies and search out-degree provides an effective mechanism to control the overhead of individual searches. However, in each case, we have only shown that each search returns a set of results. It is possible that the searches find paths that are above the search threshold, but are not the best possible paths. For example, suppose that a search for threshold set to .85 returns a path with minimum cookie value .90 . While this is an acceptable result, there may be a better path that the search missed (e.g. with minimum cookie value .95). In this case, the best path returned had an absolute error of .05. To quantify the quality of the found paths, we plot the absolute error in the paths returned by our searches as compared to an optimal search (full flooding). In Figure 4.5, we plot the CDF of the absolute error for the best path that we find versus the best possible cookie path given an infinitely knowledgeable oracle as the search threshold is changed. The higher threshold searches have a smaller possible absolute margin of error, and thus produce the best paths. However, very high threshold searches are also more likely to

produce no results at all.

4.5.2 Robustness

We analyze two components of the system: how long it takes for the system to stabilize and how well our system holds up against malicious users. Modeling malicious users is an important open research question: one for which we do not provide any particular insights in this paper. Instead, we use a relatively simplistic user model with three different types of users:

- Good users: Good users always implement the entire protocol correctly. If a good user interacts with another good user, then the cookie value assigned is always 1.0. Good users do not know the identity of any other good (or otherwise) users at the beginning of the simulation.

- Regular users: Regular users always implement the entire protocol correctly; however, when a regular user interacts with another user, transactions result in cookie value that range exponentially between 0.0 and 1.0, with a mean of 0.7. Regular users also do not know the identities of any other users when the simulations begin.

- Malicious users: All malicious users form a cooperating clique before the simulation begins. Specifically, each malicious user always reports implicit trust (cookie value 1.0) for every other malicious user. Once a malicious user interacts with a non-malicious user, there is a 20% probability that the transaction is completed faithfully, else the malicious user cheats the other party. The intuition behind this model is that nodes that are consistently malicious, i.e., that fail transactions 100% of the time, are easily detected and defeated. With malicious

nodes that periodically complete transactions, we are considering a slightly more sophisticated attack model.

At each time step in the simulation, a user (Alice), is chosen uniformly at random. Alice selects another user (Bob) from her preference list with whom to initiate a transaction. If Alice's preference list is empty, she chooses the user Bob uniformly at random. This transaction commences if Bob can find at least one path of strength at least 0.85 between himself and Alice and if Bob cannot locate a negative cookie for Alice. If no cookie path can be found, i.e., the transaction between Alice and Bob cannot proceed, Alice tries her transaction with a different user. After two unsuccessful tries, Alice chooses a random user Carol and the simulator allows a transaction without checking Alice's credentials. (Recall that in these first set of experiments, good users only want to form a large good user clique, and do not initiate transactions with other good users they know of with higher probability). When the cookie cache is full, cookies are removed from a user's cookie cache using the following rule: cookies of value 1.0 are not replaced; other cookies are discarded with uniform probability.

In the first result, we only consider good and regular users (there were 488 regular users and 24 good users in this experiment). In Figure 4.6, we plot the fraction of transactions between good users and the fraction of paths between good users. The x-axis shows the total number of transactions in which at least one party was a good node. (We choose this measure as the x-axis because in a real system, malicious nodes can fabricate any number of spurious transactions, and the only transactions that matter are the ones involving good nodes). The effect of the preference lists is clear from the plot: even though there is a less than 5% chance of a good node interacting with another good node, there is a path between any two good node within 1500 transactions. By 2500 total transactions, the majority of which were between good nodes and

regular nodes, all good nodes have cookies from all other good nodes, and the robust cooperative group has formed. This good clique will not be broken unless a good node turns bad, since 1.0 valued cookies are not flushed from the system.

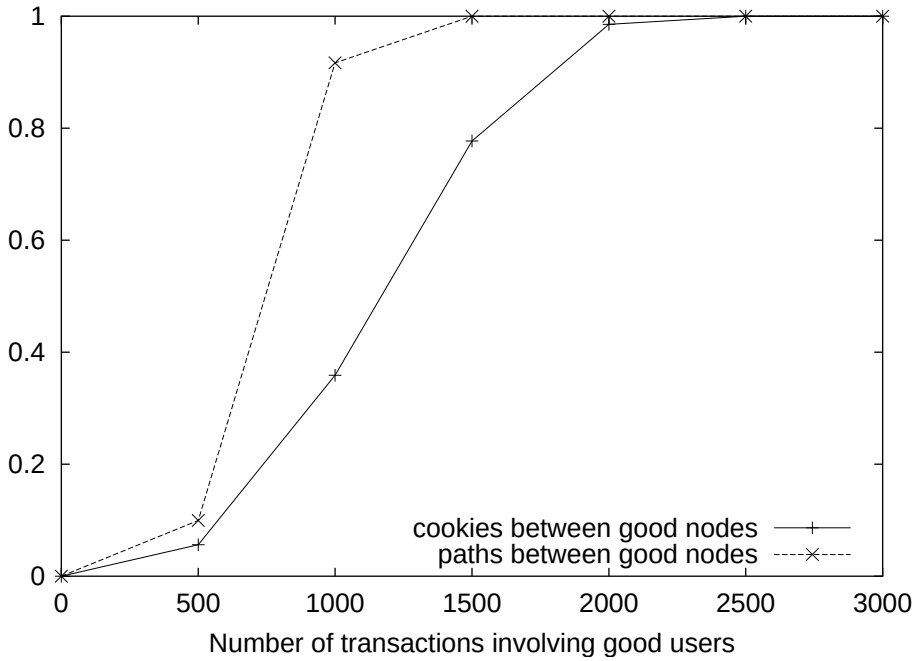


Figure 4.6: CDF of system initialization with good and regular users.

In the next set of results, we introduce malicious nodes. Figure 4.7 illustrates the fraction of failed transactions involving good nodes normalized by the total number of transactions involving good nodes. The curves in the figure show the number of failed transactions involving good nodes for varying numbers of bad nodes in the system, averaged over 1000 transaction intervals. For these results, we define failed transactions as those that produce a cookie of value less than 0.2. In the beginning of the simulation, the number of failed transactions are proportional to the number of bad users in the system. However, for all bad user populations, the good users identify all bad users and the number of good-bad transactions approaches zero. The effect of the

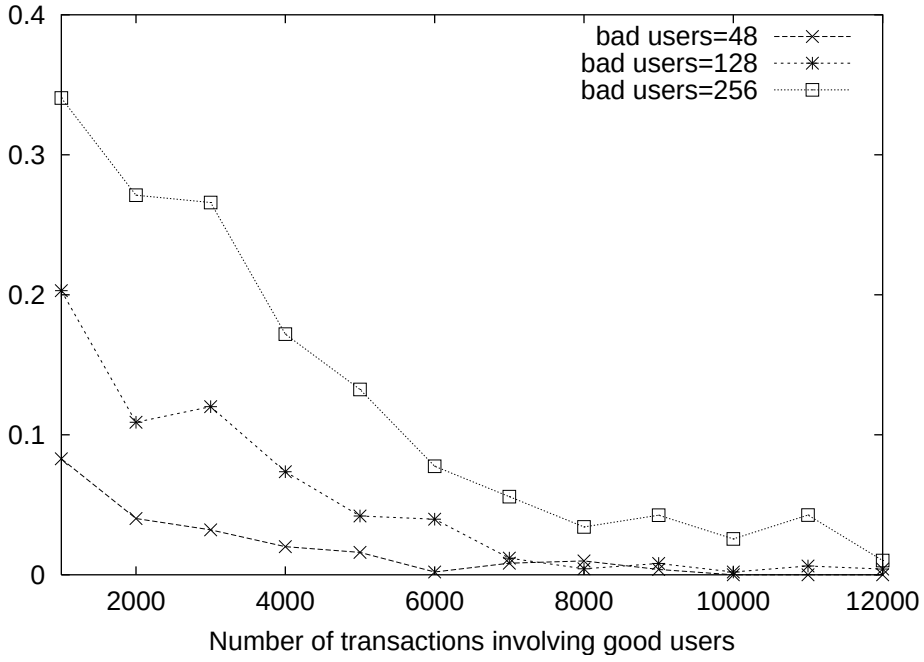


Figure 4.7: Fraction of failed transactions for good users (40 cookies at each node, 512 nodes total).

preference lists is again apparent in this experiment: recall that all bad nodes always report 1.0 trust for other bad nodes. Thus, bad nodes rapidly fill the preference lists of good nodes, but are quickly identified as malicious.

In our experiments, good users do not preferentially interact with other good users (as would be expected in a real system). Instead, if their preference lists are empty, they pick a random user to interact with. Recall that there are an order of magnitude more regular users in the system than good users. Thus, good users continue to interact with regular users and approximately 5% of good user transactions result in failures (not shown in Figure 4.7). In a deployed system, the fraction of failed transactions would be much smaller, since the vast majority of transactions initiated by good users would involve other good users.

It is important to note that even with many malicious users, a robust cooperative group *eventually* emerges in our system. This property is true, regardless of the number of positive or negative cookies good users keep, as long as *good users can choose random other users* to conduct transactions with and *can withstand failed transactions proportional to the fractions of malicious users*. Without random node selection, bad cliques can stop good users from ever communicating with another good user. However, as long as bad users cannot stop to whom good users communicate, a cooperative group emerges. In our simulations, this translates to assuming that the bootstrap node or topology server is trusted. However, in practice the boot strapping service could be logically implemented using a more secure system [129]. The good users eventually find and keep state from other good users, and this state cannot be displaced by malicious users. Obviously, the number of transactions required for good cliques to form depends on the number of malicious nodes in the system, but good users rapidly find other good users by using their preference lists. It is possible for a malicious node to infiltrate good cliques for prolonged periods, but as these bad nodes conduct transactions that fail, the negative cookies will cause these users to be rapidly discarded from the good clique.

We have varied other parameters in our experiments, and present a summary of our findings. We experimented with a different malicious node model in which the bad nodes do not forward queries from non-malicious nodes. The results for this model were not appreciably different from the model we have used in our above results. Also, it is not immediately clear how to choose the probability with which transactions with malicious users fail (recall that bad nodes succeed 20% of the time). If this probability is low, then malicious users can be identified relatively easily (usually after one transaction). If this probability is set too high, then in effect, the user is not malicious

since it acts much like a regular user. In our experiments, as the bad nodes reduce the transaction failure probability, the number of transactions required to identify all bad nodes increases, but the total number of bad transactions remain similar. We have also experimented with models in which bad users actively publish negative cookies for good users. As these users are identified as bad by the good users, these negative cookies are rendered useless. Lastly, we note that our *good* user model is probably too simplistic. Even good users may be involved in failed transactions, possibly due to no fault of their own. However, we believe our results will still hold as long as there is a definite and marked difference between the behavior of good and bad users.

4.6 Simulations on a Realistic System

As noted above, we have implemented the cookies protocol in a different simulator, which admits much larger user populations ($O(10^5)$ users). The goal of the new simulator is to model a more realistic resource discovery model, attempts to model work loads, and quantify the effect of our trust based trade limits (Section 4.3.1). We assume that there is a single bootstrap node that keeps track of the last 100 nodes to join the system. Each node periodically queries the bootstrap node to obtain a set of neighbors (if a node already has sufficient neighbors, it does not query the bootstrap node). The bootstrap node is not part of the trust inference system, and is used only to start the simulation. Each node in the system starts with a fixed number of "jobs" that they need completed, and a fixed number of jobs that they can serve for others. Of course, malicious nodes need not complete any job they accept. After discovering each other, pairs of nodes conduct a "transaction" to trade a set of jobs.

Node model Each good node maintains the following state:

- For each jobs, a status indicating whether that job is complete or not. If completed, the node remembers which node completed the job. Finally, the node also maintains state about jobs that have been issued but not completed.

- A fixed set of good cookies and a fixed set of negative cookies. These are used exactly as described above. Recall that if a node n has a good cookie from node p, then it also has a digest of the set of bad cookies that p has recorded.

- A preference list, which is a set of nodes to whom the next set of jobs will be issued.

In the simulator, bad nodes accept jobs but do not complete them with a fixed probability. We assume that there is a post-verification protocol that allows good nodes to realize that their jobs were not completed properly.

4.6.1 System Behavior

The simulations proceed as follows: the nodes initially populate their preference list using random information from the bootstrap server. Each node issues a maximum number of transaction requests (nominally set at 10 for each simulation) to nodes in their preference list. Each node maintains its preference list sorted in order of fraction of successful transactions with the nodes in the list ties are broken using the actual number of good transactions, and the transactions are issued in this sorted order.

Assume that n requests a transaction to be completed at node p. Each transaction requests a specific number of jobs to be completed. The number of jobs issued from node n to p increases exponentially with the number of successful transactions. (We have also experimented with a linear increase scheme, which we present in the results). With each transaction request, n tries to present a valid cookie path to p.

Node p accepts the request from n as follows: if n cannot present a valid cookie path, p searches the network for a bad cookie for n. If a bad cookie is found, then p rejects the transaction request. If a bad cookie is not found,i.e., p has no information about n good or bad, then p will accept the request with a fixed probability, 50%, else ask n to retry its request later. If p accepts the request, then it will, initially do one job for n. Recall that the number of jobs accepted at a node will increase with each successful transactions, as per the trust-based trading limits.

After p completes a set of jobs for n, it does not accept any other jobs from n until n performs an equivalent set of jobs for p. In general, the set of jobs accepted is constrained by the number of available resources at each node, the number of actual outstanding jobs each node has, etc.

After a transaction completes, n issues a "verification" message. This is how good nodes realize that malicious nodes have not completed their tasks properly. Once n finds that it had issued jobs to a bad node (say b), it records a bad cookie for b, and marks all the previous jobs done by b as failed. Note that allows us to model a relatively broad notion of a job. For example, jobs could be data blocks stored at other nodes, or jobs could be computations conducted at other nodes.

All simulation messages have latency between 10-11ms, distributed uniformly at random. Also, nodes issue the verification message a random amount of time after the transaction has occurred.

Preference List updates The efficiency of this system (like real systems) depends on which nodes are contacted in what order when node n wants to place jobs. In the simulator, this is reflected in the way the preference lists are maintained. When node n issues a bad cookie for any node b, n takes b out of its preference list. If n issues p a good cookie, then p gives n a copy of its preference list; n integrates this information

into its own preference list as follows. Initially, all new nodes in p's preference list are assigned the same trust (and transaction success parameters) as p. Then, these nodes replace existing nodes with lower trust value in n's preference list.

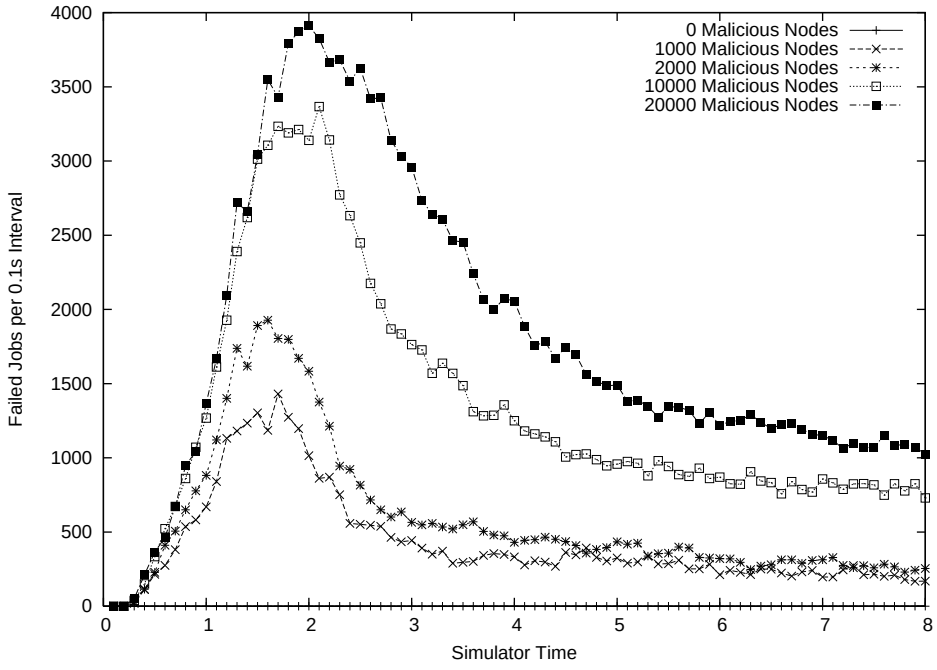


Figure 4.8: Failed jobs over time; 80 cookies

4.6.2 Simulation Results

Given our system description, there are two key metrics that we chose to measure in the simulation. Specifically, we consider how quickly good nodes place all of their jobs, i.e. completion time, and how many jobs are lost to bad nodes in the process, i.e. loss rate. Then we show results where we vary the number of malicious nodes relative to "good" nodes, and we vary the amount of state each node is allowed to hold. In all experiments, transactions only involving malicious users are disregarded.

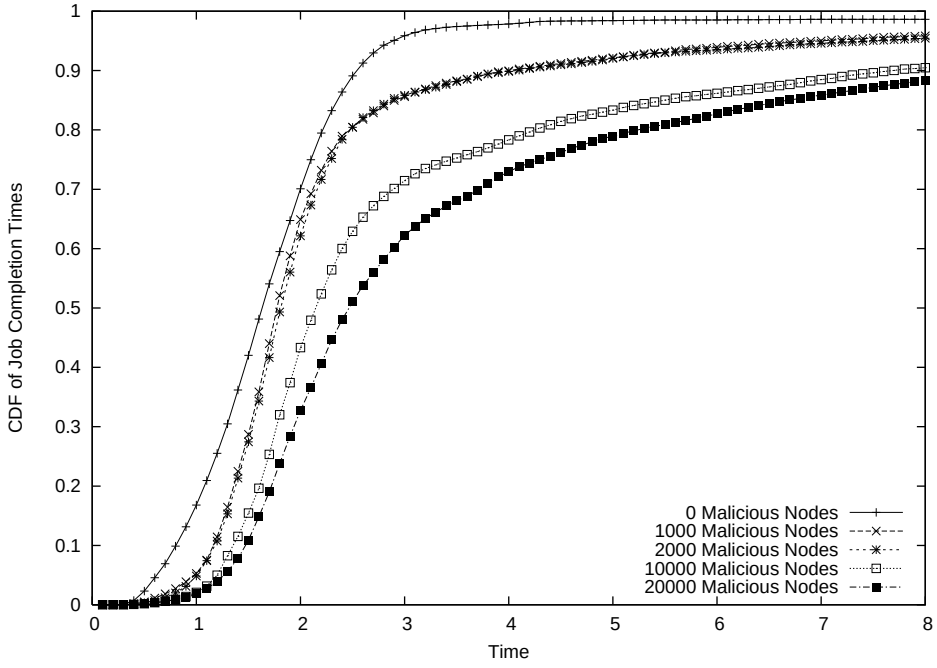


Figure 4.9: CDF of job completion times; 80 cookies

Number of Malicious Nodes In this experiment, we fix the number of good users (as defined in Section 4.5) at 1000 nodes. Each node carries state for 80 cookies, and joins the system distributed uniformly at random during the first second of the simulation. Each node has 100 jobs to place, and capacity to serve 100 jobs. A node stays in the system until it can place all of its jobs successfully; nodes time out after 30 seconds of inactivity. In all experiments, less than 1% of nodes time out. Malicious users fail jobs with 80% probability.

Figure 4.8 illustrates the number of failed jobs over time, as measured in 0.1 second intervals. As expected, the number of failed jobs initially increases as the ratio of malicious to good nodes increases. However, over time as good nodes discover each other, the number of failed jobs approaches zero. Specifically, in the case with 20,000 malicious nodes, i.e. a 20-to-1 ratio of malicious to good nodes, good nodes are still

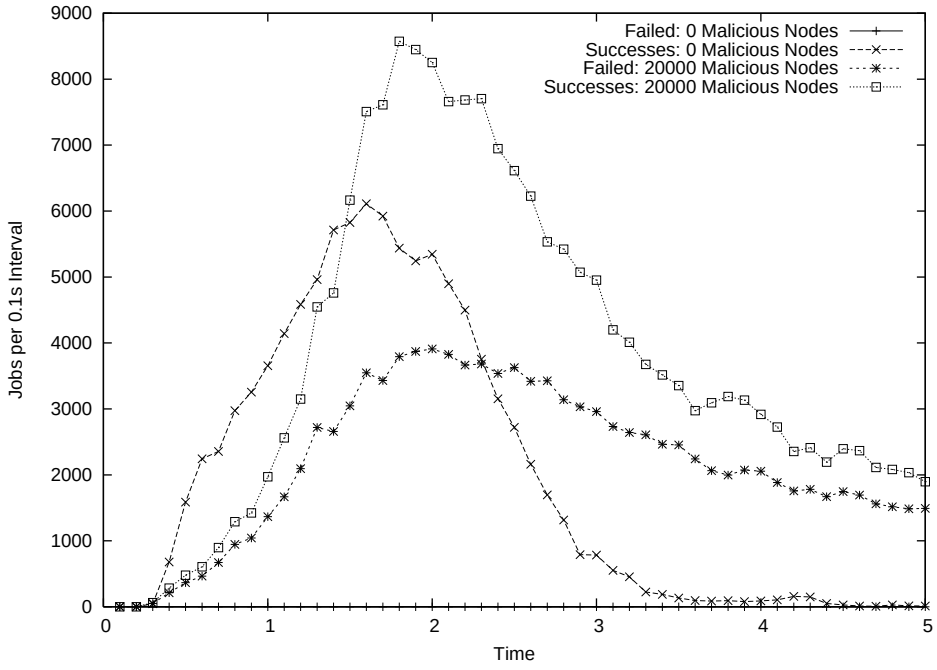


Figure 4.10: Succeed vs. Failed Jobs; 80 cookies

able to make progress as shown in Figure 4.10. By simulation time 3 seconds, over 62% of the total jobs in the system have been placed (Figure 4.9).

Note that in Figure 4.10, the number of successful jobs also reduces over time (unlike in Figure 4.7 from Section 4.5). This is because, in these simulations, good nodes leave the system after all of their jobs are satisfied. Since new nodes do not join, after a few simulation seconds, almost all the good jobs in the system are done (unlike in the previous case, where there was an unbounded number of jobs in the system). We believe that a system that incorporated continuous node arrivals and departures, as would be expected in a practical system, could result in better average performance.

Amount of State Per Node Even in this more realistic system (in which good nodes visit each other preferentially), the number of cookies a node keeps is important. We

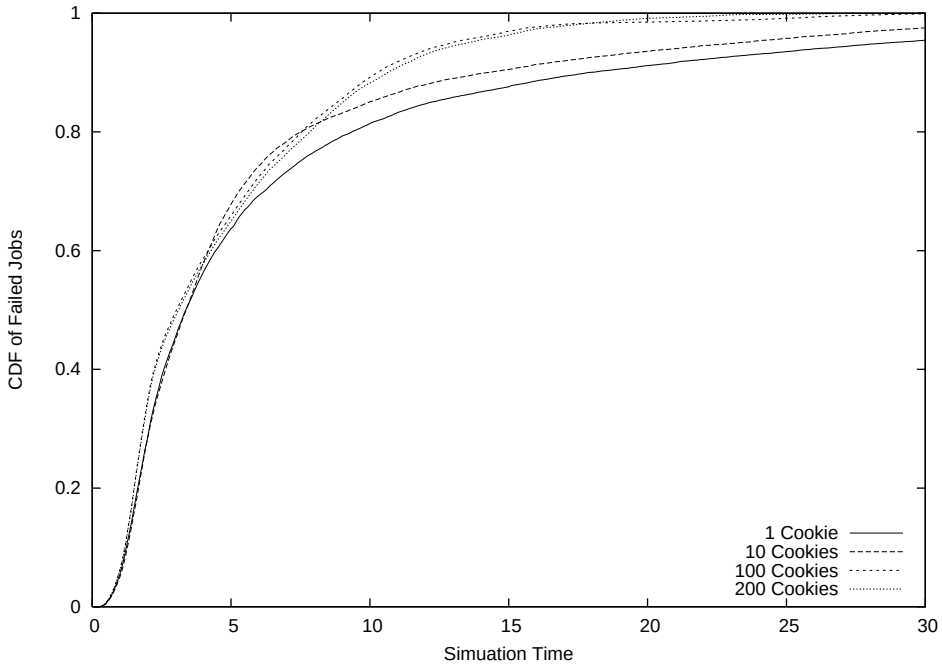


Figure 4.11: Cumulative distribution of failed jobs; 2000 malicious nodes

have experimented with varying the amount of cookie state each node keeps. In this experiment, 1000 good nodes join uniformly at random within the first second of simulation time. The number of malicious nodes is fixed at 2000, and the amount of state varies. As above, each node has 100 jobs to place, and capacity for 100 jobs. Malicious users fail jobs with 80% probability.

Figure 4.11 depicts the cumulative distribution of failed job placements over time. As expected, the number of failed jobs is initially high, but reduces as time progresses and as cookies are traded throughout the system. Figure 4.12 represents a count per 0.1 second interval of failed job placements over time. Note that as the number of cookies increase, the completion times and the number of failed jobs decrease. Observe the benefit from doubling the number of cookies from 100 to 200 is minimal, and thus a node can achieve practically full benefit from the cookie protocol from storing merely

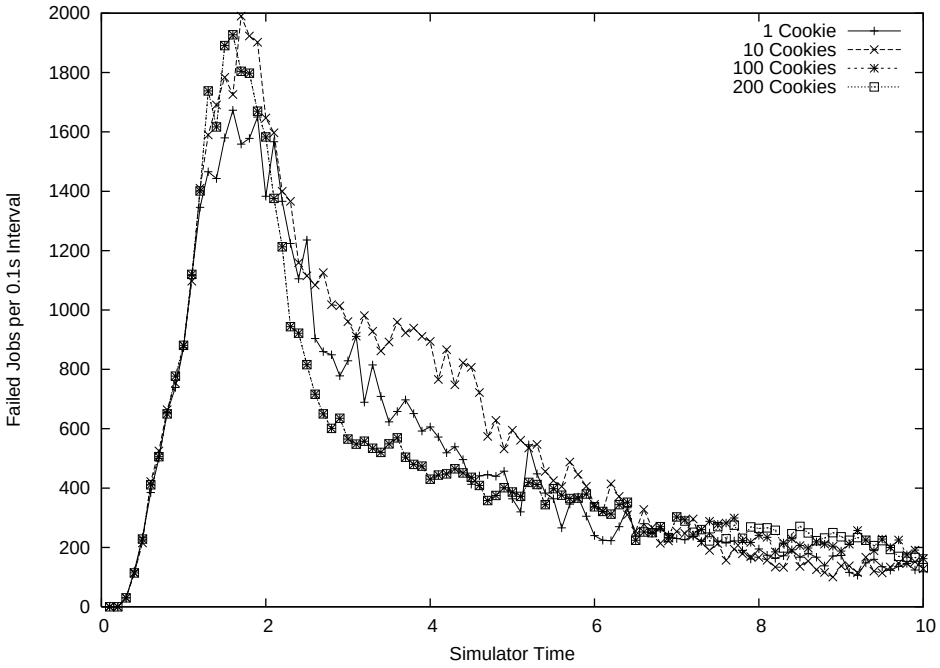


Figure 4.12: Failed jobs over time; 2000 malicious nodes

100 cookies. Figure 4.13 shows the cumulative distribution of completed jobs. Note that nodes place over 90% of their jobs in the first 7 seconds in all experiments.

4.7 Summary and Conclusions

The main contribution of this chapter is a low overhead trust information storage and search algorithm which is used to implement a range of trust inference and pricing policies. Our scheme is unique in that the search and inference performance for the whole group increases as users store more information, and information is explicitly beneficial for the storer's cause. We have presented a scalability study of our algorithms, and have shown that our technique is robust against malicious users. We have experimented with networks with over 20,000 nodes. Our results show that the pro-

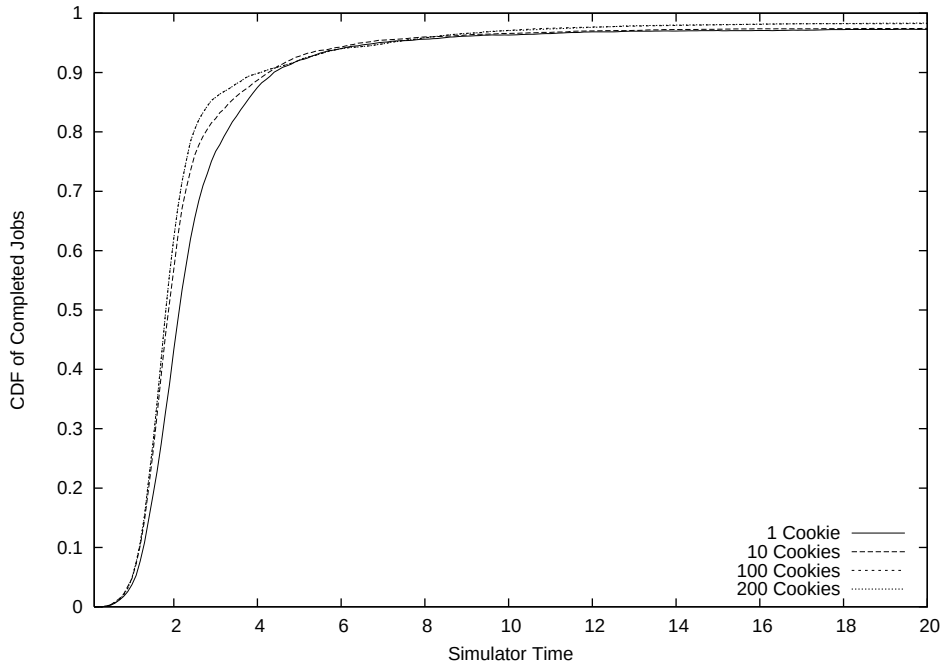


Figure 4.13: CDF of completed jobs; 2000 malicious nodes

tocol presented here scales to networks of this size, however, it is not clear that the protocol presented here will be efficient on systems an order of magnitude larger. It is likely that in very large systems, a fundamentally more sophisticated solution, e.g. based upon a DHT [146, 125] or random walks [104] would be preferred. We should note that structures such as a DHT implicitly assume that all participants are trustworthy, and we expect a solution such as ours will be required as the basis for booting a trusted DHT. We also believe techniques presented in this chapter are a crucial piece for building large peer-to-peer systems for deployment over the Internet.

Chapter 5

Resource Discovery Framework: Sidecar

5.1 Introduction

Internet measurement is key to optimizing performance, building overlay topologies, developing improved transport protocols, understanding the influence of network policy, and many other research tasks [42]. Yet the scope and detail of network measurement is limited more by the potential for soliciting abuse reports and administrative headache than by the bandwidth required to measure every interesting property [141]. Traffic designed to measure the network is often out-of-the-ordinary, interpreted by intrusion detection systems (IDSs) as anomalous or as attempts to exploit unknown vulnerabilities. To make a network measurement "safe," not just for the network but also to avoid abuse reports, requires techniques beyond those of Scriptroute [142]: it requires a fundamental shift in the design of network measurement probes and responses.

We present a measurement platform for reduced intrusiveness called TCP Sidecar. Sidecar's main insight is that soliciting abuse reports and triggering IDSs can be avoided by injecting carefully-crafted probes into externally-generated, non-intrusive network traffic. Where typical measurement tools select which hosts to probe and in

what order, Sidecar does not control the source, destination, or the exact time of the measurement. Much like a sidecar attaches to a motorcycle, TCP Sidecar attaches to a TCP connection and is just "along for the ride." The Sidecar is also a container: it can carry various measurement techniques for discovering different network properties.

In this chapter, we describe Sidecar-based topology inference, round-trip time measurement, and bottleneck location. We show how Sidecar obtains measurements, through network address translators (NATs) and firewalls, unavailable to traditional measurement techniques. Also, we describe our experience with Sidecar on PlanetLab. In Chapter 6, we describe Passenger [132], a Sidecar-based tool that makes use of the IP record route option for topology discovery.

This chapter is organized as follows. In Section 5.2, we describe the Sidecar platform and API. We present our experience from running Sidecar on PlanetLab in Section 5.3 and two examples of Sidecar-based tools in Section 5.4. Last, we describe our plans for future work in Section 5.5.

5.2 Sidecar Design

Sidecar (Figure 5.1) is a platform that supports transparently injecting measurement into TCP streams. Probes consist of acknowledgments and replayed data segments, carefully crafted not to interfere with the ongoing TCP connection. Sidecar requires no modification to end-points, requires no firewall rules (unlike Sting [126]), and can run at either end-point of a stream or even in a network middle box. Sidecar's only requirement is that it be on both the forward and reverse paths of a connection. Sidecar probes require an external source of TCP traffic, but the characteristics of the application being instrumented matter little.

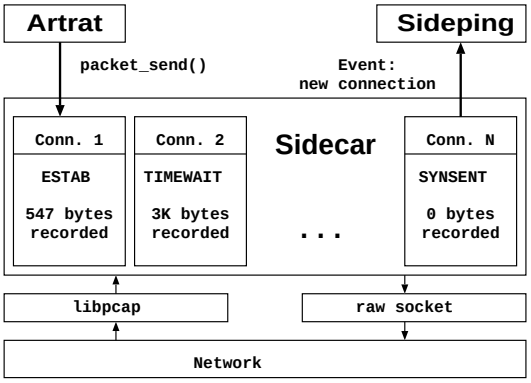


Figure 5.1: Sidecar is a platform for unobtrusive measurements that provides an event-driven interface and connection tracking to higher-level tools, e.g., artrat, sideping.

5.2.1 Unobtrusive Probing

Sidecar probes are TCP packets that look like retransmitted data. Upon receiving retransmitted data, TCP receivers send a duplicate ACK because the original ACK could have been lost (Figure 5.3). TCP senders ignore single duplicate ACKs because they could be caused by delays (Figure 5.2) or reordering in the network. Sidecar records application data passively so that segments can be retransmitted accurately (Figure 5.4).[1] Because packet loss and duplication are expected in TCP, IDSs are unlikely to generate alerts from Sidecar probes. Thus, Sidecar probes solicit responses from end-hosts without affecting applications or alerting IDSs.

Because Sidecar probes seamlessly attach and follow application streams, they can reach places unsolicited probes cannot. For example, if a Sidecar-enabled tool instrumented web server traffic, Sidecar probes could follow web connections from the server back to the corresponding web clients, even if they were behind firewalls or

[1] Paxson [111] notes that retransmitted data can change the data stream sent if the original and retransmitted data are not consistent.

NATs.

The size of the probe can be varied by changing the amount of traffic replayed, only limited by the connection MTU and the amount of data recorded. Probes can be sent even after the connection has closed by replaying the final FIN-ACK packet, as long as the receiver is in the TIME-WAIT state. The last is possible because the final ACK of the three-way close might have been lost, so replaying the FIN-ACK causes a retransmission of the final ACK.

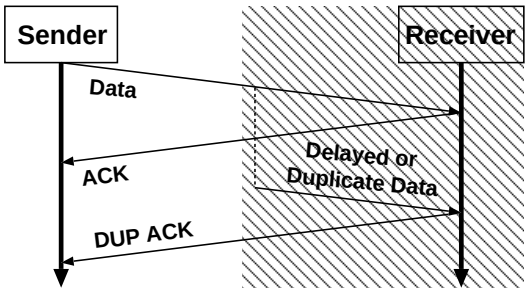


Figure 5.2: Sender incorrectly assumes (shaded region) that duplicate ACKs are from delayed, reordered, or duplicated packets.

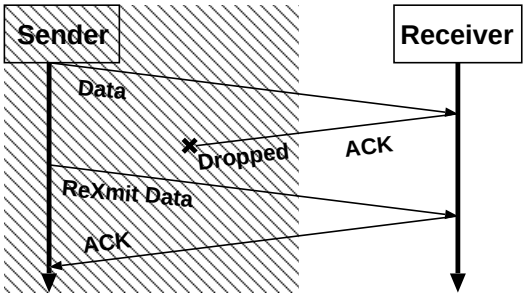


Figure 5.3: Receiver incorrectly assumes (shaded region) that probes are valid retransmissions from sender due to lost ACK.

Typically, a Sidecar-enabled tool would further modify probes. For example, one could implement a Sidecar traceroute-like [70] topology discovery tool by setting the

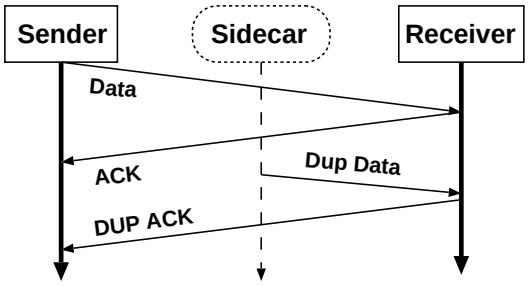


Figure 5.4: Reality: Sidecar probes are replayed data packet that generate duplicate ACKs. Probes are transparent to both sender and receiver applications.

IP TTL field of the Sidecar probe to 1, and then incrementing until an ACK was received from the end-host. With Sidecar running on a web server, this tool would obtain the path back to any client without out-of-stream packets.[2]

TTL-limited Sidecar probes can also detect NATs. If a probe is sent to IP address A at TTL=t, but the response is an ICMP time-exceeded message with source address A, we can infer that there is a NAT at hop t. We can then continue to increase the TTL to find the actual distance to the end-host, effectively probing behind the NAT. Passenger [132] is a Sidecar-enabled topology discovery tool that combines TTL-limited traceroute data with data from the IP record route option. We present two further examples of Sidecar tools in the Section 5.4.

5.2.2 Sidecar API

The Sidecar API (Figure 5.1) provides connection tracking, probe identification, round trip time estimation as well as bandwidth and memory usage limits. The Sidecar tools are event-driven applications that receive event notifications such as new connections,

[2]Discovering the topology between server and web clients is precisely the measurement by Padmanabhan *et al.* [107].

incoming and outgoing probes. The Sidecar initialization function takes a libpcap [88] filter string, e.g., "host www.google.com and port 80", as a parameter, and ignores events that do not match the filter. To construct packets for retransmissions, Sidecar tracks state for each connection, including sequence numbers and the last 3000 bytes (two full standard Ethernet packets) of application data in both directions. Sidecar automatically matches probes to their sent and received libpcap timestamps for increased accuracy over gettimeofday() [140]. Sidecar differentiates probes from legitimate traffic by changing the probe's IP identifier field.

5.3 Sidecar on PlanetLab

In this section, we discuss the lessons learned by running Sidecar on PlanetLab. We divide these lessons into two categories: problems we expected that turned out to be non-issues, and problems we did not expect.

5.3.1 Non-Issues

No abuse complaints from embedded probes. We run a Sidecar-based topology discovery tool, Passenger [132], for seven days on all traffic generated by the CoDeeN [152] web-proxy project. CoDeeN is a content distribution network hosted on PlanetLab that supports approximately 1 millions requests per day [40]. Of the 13,447,011 unique IP addresses, we ran a Sidecar based traceroute scan back to each client, using the algorithm described in Section 5.2. No abuse reports were generated from our probes.

PlanetLab VNET worked with Sidecar. PlanetLab implements a connection tracking and traffic isolation system called VNET [68] to prevent researchers from interfering with each other. With VNET, all connections are owned by a specific slice, and

slices can only read and write raw packets that come from connections that they own. It was not immediately clear that Sidecar would be compatible with VNET, because Sidecar assumes that processes in the same slice can write to each other's connections and slices can write packets to sockets after they have gone to the timewait state. It is a measure of the success of the VNET design that it was able to accommodate Sidecar.

5.3.2 Unanticipated Issues

Clocks changed and went back-in-time. PlanetLab machines run on a variety of hardware and loads, causing variable clocks and inconsistent measurements. As part of our future work, we are adding a periodic sanity check to Sidecar to compare the elapsed time to the elapsed processor cycles as returned by the *RDTSC* instruction. In this way, Sidecar can notify a Sidecar tool that significant clock skew has occurred, and to adapt accordingly, potentially discarding timing data.

Libpcap on PlanetLab drops and reorders packets. Packets drops and reordering occur more frequently on PlanetLab than our development machines. Particularly problematic was that the final ACK of the three-way-handshake would appear before the SYN-ACK packet. As a result, Sidecar's connection tracking had to be rewritten to be more resilient to these issues.

Firewalls unset DF. In an attempt to reduce the number of packets traversing libpcap, we decided to mark probe packets for which the sent timestamp was unimportant (those that are merely payload intended to cause delay, as in RPT [66]) to separate them from important traffic in the libpcap filter. We marked uninteresting packets by unsetting the Don't Fragment (DF) bit in the IP header, and adjusted our libpcap filter appropriately However, firewalls around some PlanetLab nodes unset the DF bit on

incoming packets, foiling our scheme.

IO system calls occasionally took seconds. We saw intermittent multi-second delays when running Sidecar. Using *strace -T*, we found that some open() and write() system calls would take seconds to complete. Because the problem was intermittent, we could not isolate the cause.

PlanetLab web servers don't implement persistent connections. RedHat Fedora Core 2, PlanetLab's base distribution, ships with persistent web connections disabled, despite RFC2616's recommendation that they *should* be enabled. Many of the results in Section 5.4 used connections from one PlanetLab machine to the web server of another PlanetLab machine as the source of external traffic. The lack of persistent connections shortened connection time so the majority of PlanetLab-to-PlanetLab measurements relied on the post-connection FIN-ACK Sidecar probes as described above.

Sidecar required resource limits Because Sidecar probes are triggered in response to external traffic, and the rate of external traffic is not under Sidecar's control, it quickly became necessary to implement resource metering. Sidecar implements an internal rate limiting scheme on all outgoing probes and monitors the size of the outgoing queue. If the queue size exceeds a threshold value, Sidecar ignores new connections until the queue falls below the threshold. In this way, Sidecar tools need not be exposed to the underlying details of the connection tracking, traffic bursts, or rate limiting.

Generate artificial traffic carefully Sidecar is unobtrusive because it attaches to pre-existing traffic sources. However, for testing or probing specific portions of the network, it is sometimes useful to artificially generate a seemingly legitimate traffic

source for Sidecar. In one experiment [132], we created a custom web client to visit a list of 160,000 websites from each PlanetLab node and mimic the presumed innocuous behavior of a web crawler. For each web server, the custom web client connected and performed a full HTTP session while Sidecar attached to the traffic stream to send Sidecar probes.

This experiment generated ten abuse reports, but surprisingly from the web traffic, not the Sidecar probes. The reports were not triggered by automated intrusion detection systems, but apparently by administrators noting the similarity of PlanetLab machine names after manual inspection of HTTP access logs entries. We failed to anticipate the prevalence of virtual hosting and the need to randomize the list of websites. The first caused individual clients to query a single server repeatedly, increasing the number of log entries, and the second caused PlanetLab clients to accidentally synchronize and query the same server simultaneously, decreasing the time between log entries. These issues were exacerbated due to a coding error where the *User-agent* string in the HTTP requests had been reset to a default "Mozilla"-like string. We believe that if the correct User-agent string had been in place, i.e., one pointing to an explanatory web page with contact information, fewer abuse reports would have been directed to PlanetLab.

We plan to explore new less intrusive, artificial traffic generation techniques for future Sidecar experiments.

5.4 Sidecar Tools

We present two examples of reduced intrusiveness Sidecar-based tools that suggest the generality of the platform. The first tool, sideping, provides accurate round trip

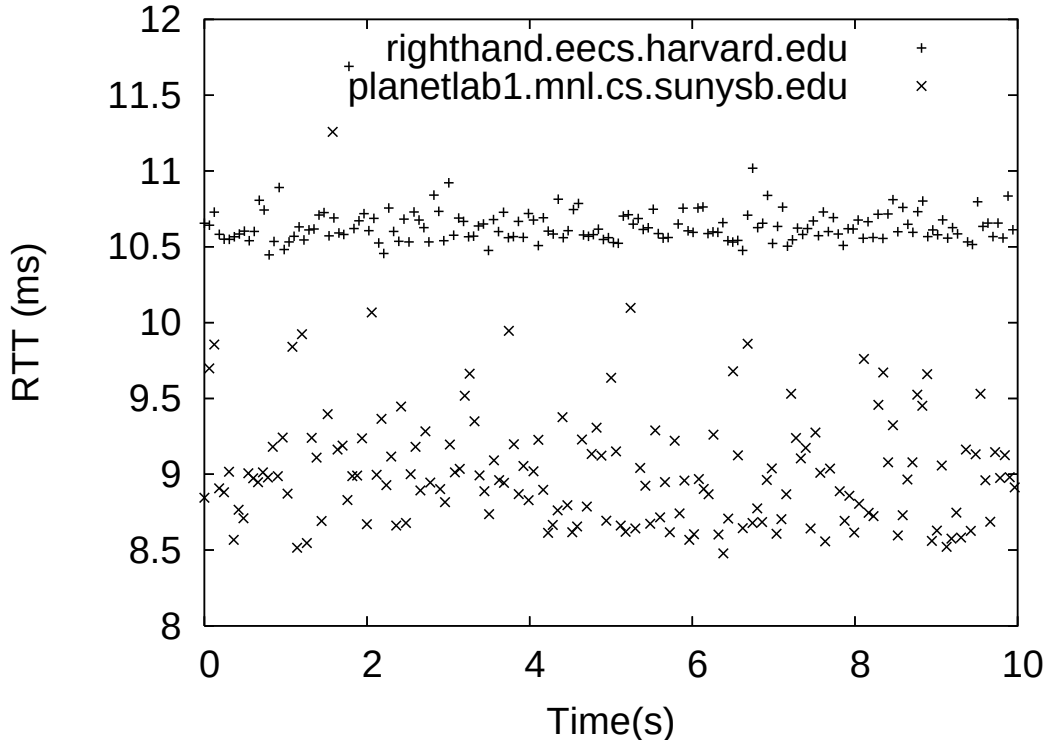


Figure 5.5: Sideping RTT measurements from UMD to two ICMP echo filtered PlanetLab nodes.

measurements with increased accuracy. The second tool, artrat, performs bottleneck location at the receiving end of a connection. As Sidecar modules, both require a separate source of connections, though we use a driver that creates new connections on demand for debugging. In other work, we describe Passenger [132], a Sidecar-enabled tool that combines traceroute probes with the record route IP option for topology discovery.

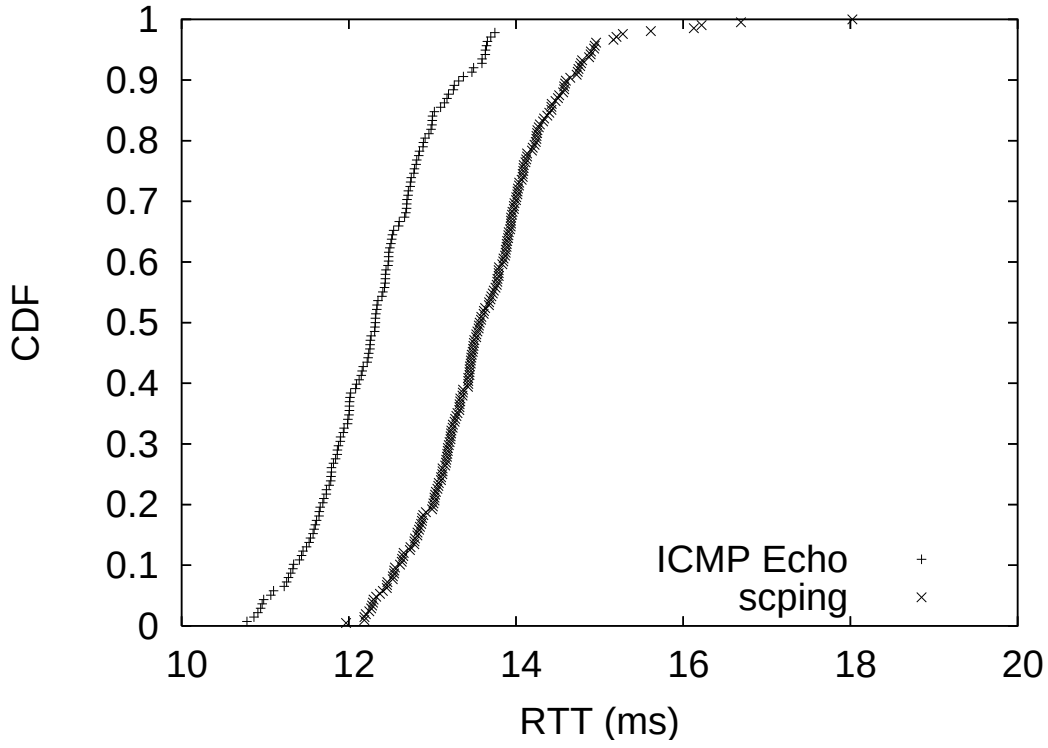


Figure 5.6: Sideping RTTs vs ICMP Echo: Difference exposes NAT + wireless network.

5.4.1 sideping: Round Trip Time Estimator

Sideping estimates network latency by measuring the round trip time of Sidecar probes in a TCP connection. Although latency is an unsophisticated measurement, the extensive use of the all-pairs PlanetLab ping [115] data set demonstrates its importance.

Sideping seeks to avoid over- and underestimation of round trip times. Tools like ping can overestimate RTTs because they assume that the probe's sent time is close to the sendmsg() call. By contrast, Sidecar records the timestamp from libpcap for the time the probe was given to the network interface device. Rate limiting means that

the probe can reach the interface well after the application asked it to be sent. Ping can also underestimate RTT when probing a host behind a NAT. Because sideping can follow TCP connections to their end-points, researchers can gain insight into network dynamics behind NATs. Compared to TCP's internal RTT estimation protocol, sideping does not inflate RTTs by including delayed ACK time.

Figure 5.5 shows sideping collecting previously-unavailable RTT measurements from two PlanetLab nodes that filter ICMP echo packets. Figure 5.6 shows the difference between ICMP echo and sideping RTT measurements traversing a NAT to a wireless network. The ICMP echo reply packets return with a larger TTL than the sideping responses. The difference between the two techniques, 0.797 ms on average, is extra delay from the wireless network.

5.4.2 artrat: Receiver-side bottleneck detection

Artrat[3] is a Sidecar-based tool that attempts to locate local bottlenecks, from the receiver's perspective. This information could be used to decide whether local network resources were sufficiently-provisioned or if they should be upgraded. Although tools [9, 66] exist to perform bottleneck location by instrumenting the sender, we believe we are the first to focus on the perspective of the receiver. We believe that this tool will be of use to PlanetLab researchers who are concerned with local bandwidth conditions during their experiments.

Artrat correlates the congestion delay in the connection with the queuing delay at local routers. Sidecar reports any router whose queuing delay correlates over time with the congestion delay as a suspected bottleneck. Similar to TCP Vegas [28], we measure the congestion delay as the difference between the current RTT and the baseline

[3] Artrat: Active Receiver TCP Rate Analysis Tool

(minimum) RTT.

To measure the queuing delay at routers, artrat first discovers the router, a, five hops[4] into the network using a TTL limited probe. Then, artrat periodically sends ICMP echo probes with the IP timestamp option [116] to router a, and parses the response (Figure 5.7). The IP timestamp option records the time at each router in milliseconds and (by RFC792) the ICMP echo response packet has the same options payload as the echo request. In this way, artrat learns the local time of each router along the outgoing path to a, and, most importantly, of each router on the path from a back to the receiver. Similar to our definition of congestion delay above, we define the queuing delay between two routers as difference between the current jitter and the minimum observed jitter. If we label the IP option timestamps for the jth probe as $S_{1,j} \ldots S_{9,j}$ and call $S_{0,j}$ and $S_{10,j}$ the send and received times for the ICMP probe j, we can calculate the queuing delay, $q_{i,j}$, between router i and $i+1$ as computed by the jth probe as:

$$q_{i,j} \quad = \quad S_{i+1,j} - S_{i,j} \quad - \quad min_k(S_{i+1,k} - S_{i,k}) \tag{5.1}$$

Then we compute the correlation between the congestion delay and $q_{i,j}$ for all routers i, and output the $i \rightarrow i+1$ link with the highest correlation as the likely bottleneck.

We ran artrat on a local network testbed to test the scheme. The testbed consisted of a client connected with a 10Mbps Ethernet card to a 100Mbps network. We ran artrat while the network was idle (Figure 5.8) and while downloading a 20MB file (Figure 5.9). When the network was idle, artrat found no significant queuing delay. While the

[4] *Five* hops into the network was chosen because the IP header limits the number of recorded timestamps to nine. If the local path is symmetric, five hops is the maximum distance the probe can travel away from the receiver so that the return path does not exhaust the IP option array.

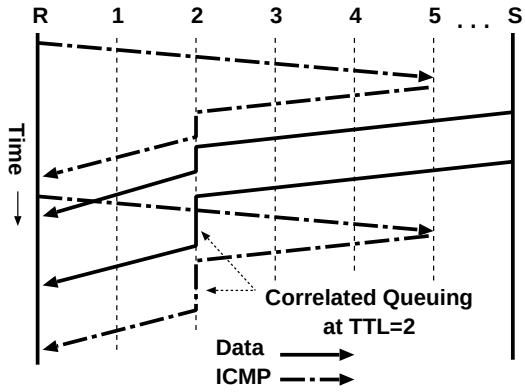


Figure 5.7: Overview: Artrat correlates congestion and queuing delays to do receiver-side bottleneck location (example: bottleneck from S to R at TTL=2).

network was in use, artrat successfully found queuing delay on the inbound portion of the 10Mbps link (labeled "1→ R" in Figure 5.9).

The coefficient of correlation between the congestion delay and routing delay at router 1 in Figure 5.9 is 0.24. Although this is low, the second highest coefficient of correlation was 0.072, so artrat successfully found the 10Mbps link as the bottleneck. This correlation analysis simply compared the ith ICMP probe with the ith Sidecar probe, ignoring timing information and dropped probes. Implementing robust time-series analysis techniques is future work, but the technique shows promise.

Artrat makes two assumptions that must be validated before use. First, due to the baseline measurements, this technique is sensitive to clock skew, both locally and at routers. Thus, artrat must periodically compare the time elapsed on all clocks against some external source, like the *RDTSC* instruction or a remote NTP server, using the techniques of Moon *et al.* [102]. Second, artrat requires some symmetry in local routing. Specifically, artrat can only discover bottlenecks on the path of the returning

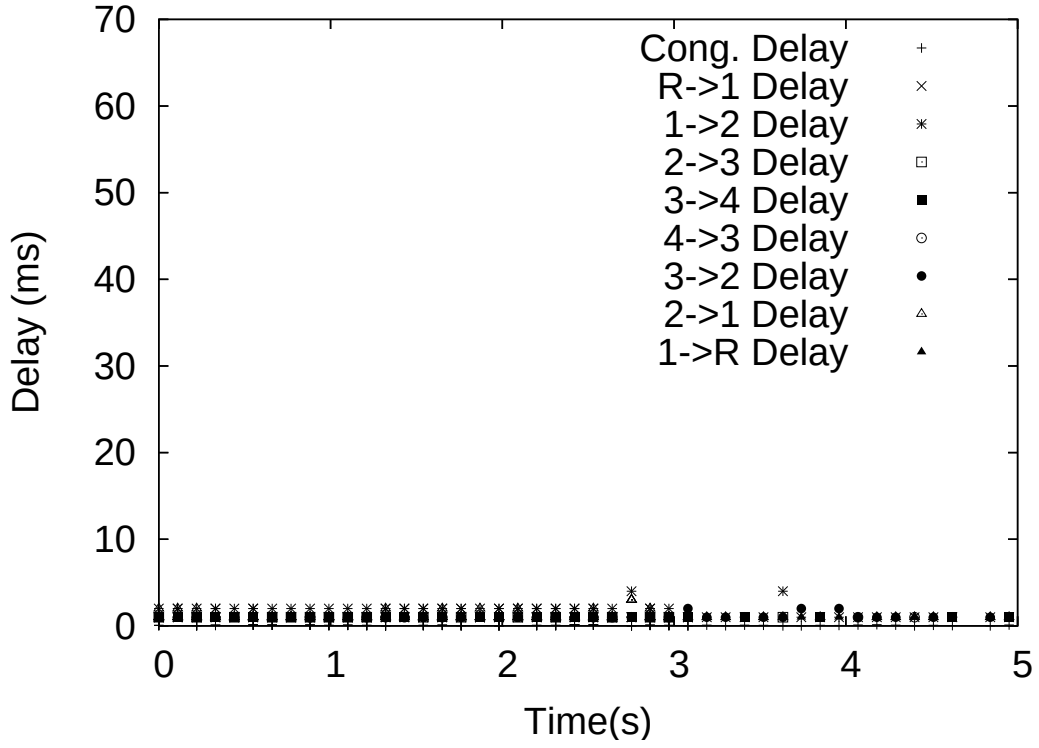


Figure 5.8: Artrat Experiment: Idle connection: no bottlenecks.

ICMP probe. It is the subject of future work to integrate artrat with topology-aware tools to verify the symmetric nearby network assumption (perhaps using remote traceroute servers as in Rocketfuel [139] or another service).

While one could create a version of non-Sidecar enabled artrat, the Sidecar version benefits from increased accuracy in RTT measurements (Section 5.4.1) and already written connection tracking libraries.

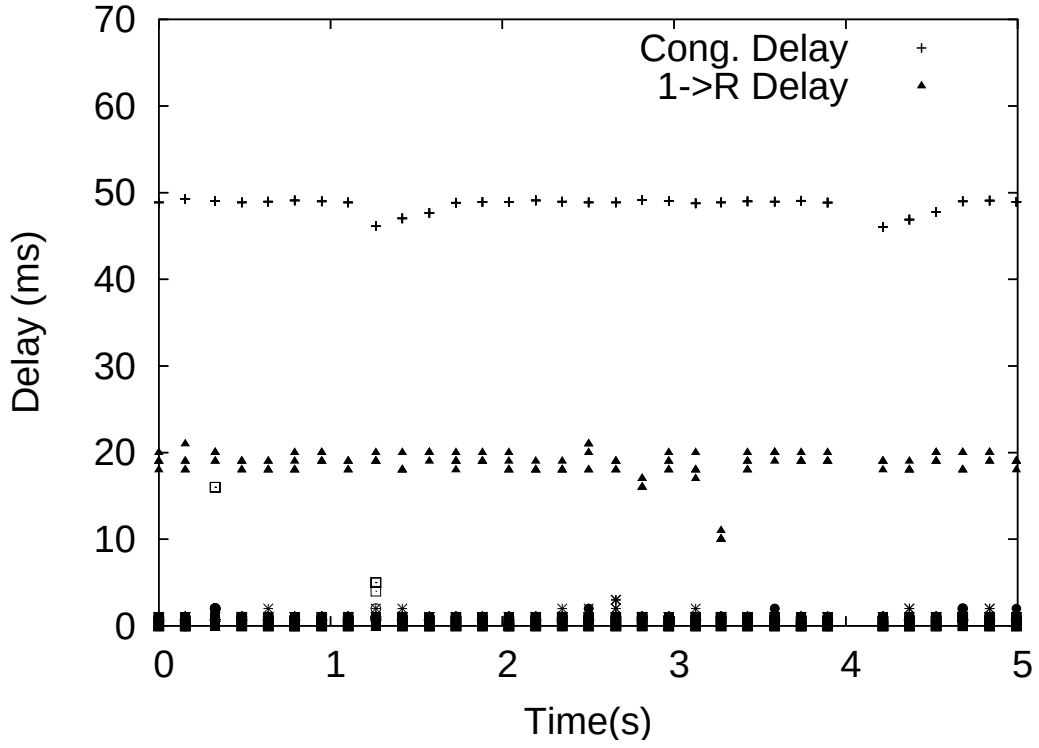


Figure 5.9: Artrat Experiment: Data Transfer: bottleneck at 1→R, i.e., 10Mbps link. (Data labels as in Figure 5.8)

5.5 Conclusion and Future Work

TCP Sidecar provides a platform for non-intrusive network measurements by injecting probes into external sources of traffic. One potential source of external traffic is PlanetLab, which supports many high volume, publicly accessible services, e.g., CoDeeN [152], OpenDHT [122], Meridian [156], and CoralCDN [56]. Many of these services perform their own measurements, and might benefit from less intrusive Sidecar-based approaches. This creates an interesting symbiotic relationship between measurement studies and PlanetLab hosted services.

In our future work, we plan to complete a large-scale Sidecar-based measurement study. Our current work is in concert with the CoDeeN project, but we plan to expand to other PlanetLab services to avoid host selection bias. We also plan to better document the Sidecar API so that other researchers might benefit from our work. Sidecar is available for download at `http://www.cs.umd.edu/projects/sidecar`.

Chapter 6

Resource Discovery With Record Route

6.1 Introduction

Complete and accurate maps of the Internet backbone topology can help researchers and network administrators understand and improve its design. Good maps are unavailable, however, and while some of the reason for this may be social—publishing accurate information is not obviously in a network operator's interest—various research projects [38, 61, 109, 138] have shown that information helpful for research [161, 14] can be collected through traceroute-like probing.

Unfortunately, the increasing use of MPLS, the size of the network, filtering of traffic directed toward router addresses, and fine-grained multi-path routing add intolerable error to traceroute-based studies. Further, large-scale traceroute-based studies typically yield abuse reports from destination hosts behind intrusion detection systems that interpret an incoming traceroute as a port-scan or intrusion attempt [140].

We present a topology discovery tool, Passenger, that revisits IP's record route option to yield more accurate (corroborated) path information, and Sidecar, a system for embedding probes within TCP connections to reduce the intrusiveness of network probing.

With Passenger, we find that the record route option has been prematurely dismissed as a useful tool for network topology discovery: specifically, that its noted limitations are not severe. Its first limitation is that only nine hops of a trace are recorded. PlanetLab [114] allows us to deploy our tools within nine hops of 87–98% of observed addresses (Section 6.5); distant portions of the network are poorly sampled by traceroute anyway [84]. Second, routers may forward packets with options at lower priority, but we are interested in topology, not performance. Third, firewalls may block packets with record route, yet firewalls also often block traceroute, so little is lost. Finally, intrusion detection systems are likely to report IP options as exceptional events; we find that TTL-limited record route packets can keep destination hosts from seeing and objecting to IP options.

Our challenge is *not* in these obvious limitations: it is matching traceroute-observed addresses with record-route-observed addresses to infer a path that is more correct and complete than either method can collect alone. This is challenging first because traceroute and record route observe distinct addresses with no overlap. We have also observed routers that (a) insert record route entries without decrementing TTL, (b) insert a record route entry when expiring a packet (most do not), (c) insert record route entries only sometimes, perhaps not when under load, and (d) do not insert record route entries at all. This diversity of implementation and configuration makes aligning traceroute and record route paths a daunting task.

Correct alignment of the addresses returned by both schemes is an instance of alias resolution [109, 136, 64, 80]: determining which IP addresses belong to the same router. This means that we can verify the alignment of paths using Rocketfuel's ally tool [138]: when the IP addresses discovered respond to direct probing and when they do not respond, we can discover new aliases not found by other tools. This ability to

find aliases for unresponsive routers is a significant step in improving the correctness of measured network topologies.

With Sidecar, we show how to embed Passenger's record route and TTL-limited probes within TCP connections to collect path information with limited intrusiveness. Embedding within TCP requires tracking connection state and disambiguating acknowledgments of probe TCP packets from those of the normal transfer. Although passive observation of TCP behavior and timing has been useful for measurement [83, 107, 160, 16], and traceroute can be embedded with paratrace [76], we believe this is the first demonstration of the feasibility of running traceroute-like probing within TCP connections to pass through firewalls and avoid false accusation by intrusion detection systems.

This chapter is organized as follows. In Section 6.2, we describe the problem of aligning record route with traceroute. We present Sidecar in Section 6.3. In Section 6.4, we describe Passenger and our data collection methodology with the results in Section 6.5. We then conclude and describe our plans for future work in Section 6.6.

6.2 Mapping with RR

6.2.1 Conventional Wisdom

In this section, we describe why the record route (RR) IP option has been unnecessarily discounted as a topology discovery technique and describe why its limitations are not flaws.

As an IP packet with the record route option traverses a router, the router enters its address into an array at a given offset in the IP header and updates the offset. Because space in the IP header is limited, the record route array can hold at most nine addresses.

Paths through the Internet are often longer than nine hops, so much of the network would be undiscovered by record route. Fortunately, we can send record route packets from PlanetLab [114]. In Section 6.5, we find that at least 87% of addresses in our survey are reachable in nine hops from at least one PlanetLab node.

IP options increase the chance of delay, discard, or alarm at intrusion detection systems (IDSs). Delay matters little for topology discovery. Discard is common at firewalls for traceroute packets and record route is not much different. Fonseca *et al.* [55] found that 46% the paths between PlanetLab hosts drop packets with RR set but that only 8% of those paths are blocked in transit networks. We believe that firewalls and IDSs are typically close to the end-hosts that they protect. To reduce the likelihood of intrusion alarm without sacrificing data from the core of the network, we prevent RR probes from reaching hosts by limiting the TTL. We set probe TTLs to the minimum of the hop count to the end host minus three or eleven, because more information in record route is very unlikely after eleven hops.

6.2.2 Simple Topology Discovery

We first describe the process of discovering network topology using record route when the network is simple: all routers always decrement TTL and append to the record route array when not expiring the packet. The diversity of router implementation and configuration means that this model is too simple to be directly applied, but it remains useful as an introduction. In the next subsection, we dive into this complexity.

The addresses discovered by traceroute and by record route do not overlap. RR records the address of the *outgoing* interface onto which the packet is sent or the router's designated "loopback" address. By contrast, the "time-exceeded" messages solicited by the TTL-limited probes of traceroute [70], by convention, come from the

incoming interface where the packet was received.

In Figure 6.1, we discover the incoming and outgoing interfaces of each router by sending probes with the RR option and different TTLs. We retrieve the RR array from the header of the packet encapsulated in the ICMP time exceeded message. (The IP header of the response does not include record route.) The ith address of the RR array is an alias for the router that sends the ICMP time exceeded message for TTL=i.

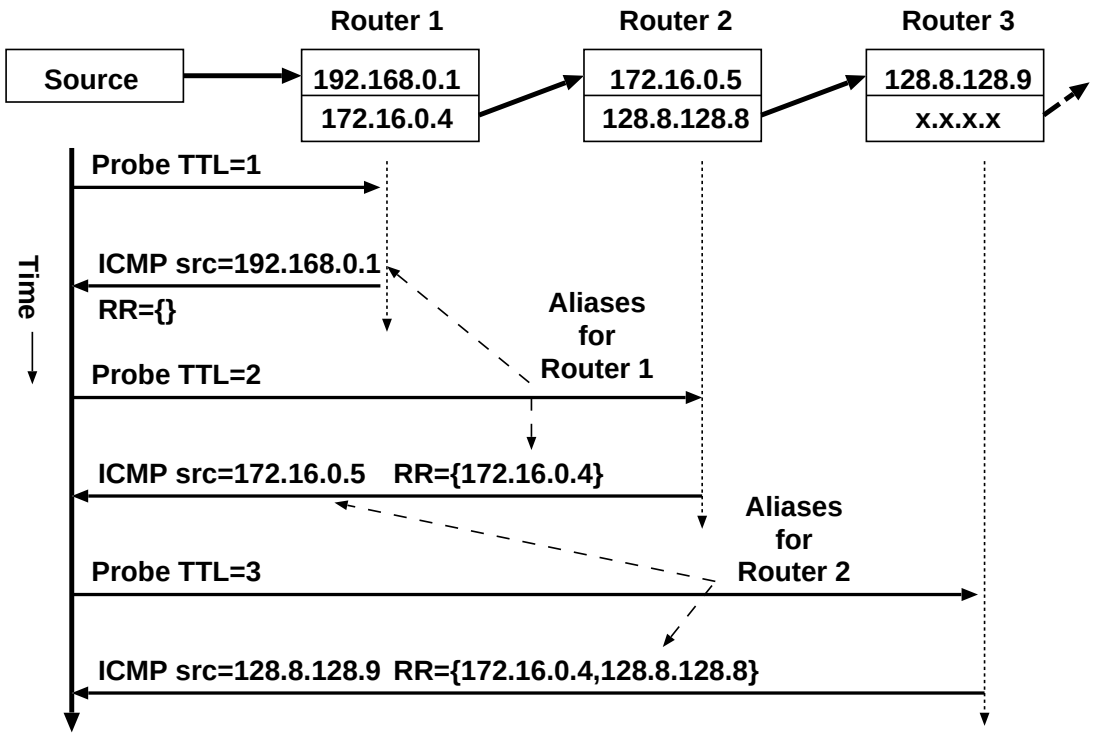


Figure 6.1: Alias resolution with TTL-limited record route.

Load-balancing can cause incorrect topology inference when only traceroute is used. When packets from the same traceroute traverse multiple paths, especially of different lengths, incorrect edges can be inferred (Figure 6.2). The traceroute-inferred network incorrectly links router B to router E because the third probe took a different path. However, the first entry in the RR array in the third probe changed from

A2 to A3, exposing the new path and providing feedback that more probes are necessary to discover the entire topology. With record route, the route changes problematic for traceroute become a benefit because they permit the discovery of more topology information.

6.2.3 Router Behavior Inference

Not every router has the same record route behavior. Specifically, with each additional TTL, our probes may record zero, one, or many new record route addresses. In this section, we list six distinct router implementations and describe the rules we use to classify individual routers into their respective implementations. By classifying routers, we are able to match traceroute and record route addresses on the same router, thus enabling alias resolution and topology discovery.

The router implementation variants we have discovered are:

Type A routers are common: they record the outgoing interface address only if that interface transmits the packet (not when it expires at the router) as described in the simplified examples above. The prevalence of this behavior is consistent with belonging to Cisco routers.

Type B routers are less common: they record the outgoing interface address the packet would have taken even when the packet expires at that router. Because we infer Type B behavior within Abilene [3], we believe it to be consistent with Juniper routers.

Hidden routers never decrement TTL, but always mark record route. Such routers are discovered only by record route probing. We believe these routers are typically part of an MPLS tunnel where decrementing TTL is considered optional.

Type N routers never mark record route packets but always decrement TTL. We believe this to be a configurable option.

Lazy routers do record the outgoing interface, but decrement the TTL only of packets lacking the record route option. We speculate that slow-path processing omits the TTL-decrementing step.[1]

Flaky routers sometimes, but not always, append record route entries. We speculate that they omit processing when under load.

As further diversity, some router configurations appear to not process RR options if the outgoing interface is part of an MPLS tunnel.

The variety of router types make router classification ambiguous. Because different topologies and router implementations can generate the same trace (Figure 6.3), a router may be misclassified, leading to mismatched addresses. Thus, "128.8.128.8" may be an alias for the router with address "128.8.128.9" (Figure 6.3, top topology) or an alias for a hidden router that does not appear in the traceroute data (Figure 6.3, bottom topology).

We describe the rules we use to classify routers from the available data.[2] In these rules, the *current* router originated the ICMP response we are attempting to classify, while *previous* and *next* refer to the routers one TTL closer and further. For clarity, we classify probes based on their RR *delta*: how many new RR entries were added since the previous TTL.

We evaluate the resulting inferences in Section 6.5.

---

[1] We have not confirmed that the TTL is decremented if the record route list is full. As such, we consider this a potential flaw in router software.

[2] Router OS fingerprinting, or similar additional probing may yield a more accurate classification; we avoided this because router addresses are often not routable.

A-to-B transition If a probe's delta is two, we classify the current router as Type B and the previous router Type A. The first new address belongs to the previous Type A, which it did not place in the previous TTL; the second new address belongs to the current Type B.

Types A and B transitivity If a probe's delta is one and the previous router is Type A or B, then we classify the current router as the same. Unfortunately, the transition from a type B router to another type B router in one hop is indistinguishable from a B-to-Hidden-to-A transition. We use the off-by-one rule to disambiguate.

Off-by-one Because address prefixes are assigned to networks, addresses that are numerically off-by-one are more likely to represent interfaces at either end of a point-to-point link (assigned a /31 prefix) than interfaces on the same router. This heuristic overrides the previous two rules and can assert the existence of hidden routers.

Double-zero The current router is Type N if the current and next routers' deltas are both zero.

Lazy detection The current router is Lazy if all probes *with* record route come from IP address X, all probes *without* from IP Y, $X \neq Y$, and all probes *without* for the next router also come from IP X.

As we apply these rules to Figure 6.3, the second probe has a delta of two, implying a transition from a Type A to Type B router. However, addresses 128.8.128.8 and 128.8.128.9 are "off-by-one," so because the off-by-one rule overrides the A-to-B transition rule, we declare that the bottom topology is most likely the correct topology.

We are unable to classify all traces: these rules can lead to ambiguity and contradiction, e.g., one trace might classify a router as Type A, when another trace could

classify the same router as Type B. Using these simple rules, we can classify 65.4% of our 65 million traces without contradiction. In Section 6.5, we use the alias resolution tool *ally* [138] to evaluate the correctness of our classifications. Robust and complete router classification and address alignment is the subject of our continued work.

6.3 Sidecar Design

Sidecar is our engine for injecting probes, including TTL-limited packets, into TCP connections from user level without altering TCP behavior. Probes sent from within TCP connections can traverse and expose the firewalls and NATs that traceroute probing cannot. The Sidecar system comprises connection tracking, probe identification, RTT estimation, and rate limiting without requiring kernel (firewall or module) support. These design choices make it possible to transparently instrument TCP connections, even from the middle of the network. Figure 6.4 shows how the TCP embedding logic in TCP Sidecar is separated from the higher-level probe-generation driver, Passenger, allowing easy development of new TCP probing tools. Other Sidecar applications [130] include round trip time and bandwidth estimation.

Injecting probes into TCP without harming connections requires careful design. Sidecar records state and application data via libpcap for many connections in parallel. Sidecar probes take the form of replayed packets carefully crafted to look like retransmissions. Sidecar probes are thus transparent to connections because TCP is robust to packet reordering and duplication. Responses to these probes are either time-exceeded messages from routers, which are ignored by the kernel, or duplicate acknowledgments from the destination host. Because three successive duplicate acknowledgments serve as congestion notification event, Sidecar is careful to not send probes when data is

outstanding. We accomplish this by delaying probes until the connection is idle for at least 500 ms.

By changing the TTL of replayed packets, Sidecar is able to accomplish traceroute-like functionality in a TCP stream. When UDP-based traceroute reaches the destination host, the type of response changes from "time exceeded" to "port unreachable": an unambiguous sign that the destination host has been reached. With Sidecar, the destination's equivalent response to a replayed packet probe is a duplicate acknowledgment. Because TCP acknowledgments are cumulative and do not identify the specific segment/probe that triggered them,[3] Sidecar cannot distinguish multiple responses from the destination. For efficiency, Passenger sends low TTL probes that will probably not reach the destination in parallel and higher TTL probes that might reach the destination serially.

If the connection closes before probes can be sent, Sidecar can replay the final FIN/ACK packet if the destination is in the "time-wait" state. FIN/ACK probing is not ideal, since the local TCP stack may generate unnecessary RSTs in response to receiver ACKs.

Sidecar permits trivial NAT detection. If Sidecar receives a "time exceeded" message from the destination IP address of the probe, we conclude that a node behind a NAT expired the packet and the source address of the error was rewritten by the NAT.[4] The destination's distance behind the NAT can be determined by incrementing the TTL until receiving a redundant ACK.

---

[3]The DSACK extension [54] does identify the duplicated segment but it does not appear widely deployed [99]. Identifying DSACK support and using it to match multiple probes and responses is future work.

[4]We have found an exception to this rule: a firewall near a PlanetLab source in China would forge "time-exceeded" responses as if from the distant destination address.

Sidecar parses ICMP extensions [25] allowing detection of MPLS tunnels that support them [24]. Although the utility of knowing MPLS labels is unclear, these extensions proved helpful in debugging the effects of MPLS on router classification.

6.4 Passenger Design

While Sidecar is the underlying engine for embedding probes into TCP streams, Passenger performs the higher level topology discovery probe generation (Figure 6.4). Our evaluation goals in exploring Passenger are to: (a) show that embedding record route probing within TCP connections is feasible, (b) quantify how much of network topology record route discovers, and (c) demonstrate reasonable correctness in address alignment over a variety of paths. To construct a dataset for this evaluation, we allow Passenger to observe and trace within the TCP connections of two applications: a web crawler and the CoDeeN web proxy.

6.4.1 Passenger Logic

Passenger implements the logic of our traceroute and record route probing. Sidecar determines the type of packet to send, determines the round trip time, and returns responses; here we are concerned only with the logic of the measurement. Passenger starts as soon as the completed connection has been idle for one half second. Because web-like connections terminate soon after becoming idle, we try to compress the traceroute into as little time as possible. Passenger remembers the addresses it probes so that it will not repeat a trace for the same source/destination pair.

Passenger traces have two phases. Let *safettl* represent an estimate of the number of hops that probes can be sent into the network without reaching destination firewalls

or IDSs. We set *safettl* to the minimum of eleven or three less than the TTL of the destination, as estimated from observing the TTL hops remaining of incoming packets. In the first phase, Passenger sends probes in parallel for all TTLs between 1 and *safettl* with record route set, and then waits for one RTO for them to return. Passenger repeats this process six times, alternating probes with and without the record route option. In the second phase, like traceroute, passenger sends three probes per hop starting at TTL= *safettl* +1 until it reaches the destination or TTL=30 is reached. In this way, Passenger records traceroute data for the entire path and record route data for TTL=1 to *safettl*.

6.4.2 Data Sources

CoDeeN CoDeeN [152] is a network of partially-open Web proxies deployed on PlanetLab. We ran Passenger for a week (May 17–24, 2006) observing CoDeeN servers. Although CoDeeN is installed on 671 hosts, because some were inaccessible, rebooted during that week, or had too little disk, we only recorded complete data from 234 sources. Passenger monitored connections on port 3128 to CoDeeN users, not proxied connections to origin servers. We collected 13,447,011 traces.

Web Crawler We connect to every web server we could discover. In the first phase, we ran the Larbin [6] web crawler, seeded with `http://slashdot.org` to find 316,094 websites. We then removed duplicate IP addresses to arrive at 166,745.[5] In the second phase, we ran a multi-threaded Web client from each available PlanetLab node to each address, using Passenger to instrument the connection. Our Web client holds each connection open for 30 seconds, as HTTP persistent connections would, to

[5]We initially failed to consider virtual hosting, leading to reports of abuse.

allow the measurement to complete.[6] The client retrieved the robots.txt file from each server. We collected 51,742,928 traces.

PlanetLab We also collect a PlanetLab-to-PlanetLab data set using the same web client and PlanetLab hosts as servers. This data set is a strict subset of the web crawler data, but manageable in size. We collected 151,688 traces.

6.4.3 Safety

We limit probes to 500 Kbps per second; above this rate, or the rate at which the raw socket accepts new packets, we skip connections to trace rather than significantly delay the probes of traces in progress. To run within CoDeeN and ensure little interference, we used kernel resource limits to prevent unexpected, excessive memory and processor consumption. We also fetched result data often to reduce disk consumption. We tested our Web crawler and CoDeeN instrumentation from a local machine, grew to a few PlanetLab nodes, and only eventually to all of them. This approach prevented early implementation errors from causing undue havoc.

6.5 Results

Our evaluation focuses on feasibility, coverage of the topology with record route, and correctness of address alignment (alias resolution). We also comment on the intrusiveness of the technique as measured anecdotally by the absence of abuse reports. Our experiments traversed 8,817 ASes (Table 6.1) and generated 65,189,939 unique traces.

Approximately 16% of IP addresses in our experiment were discovered by record

[6] If the remote server closes the connection earlier, it remains in TCP's "time-wait" state, allowing further measurement.

route alone (Table 6.1, row three). These routers are either anonymous (do not respond to traceroute), use MPLS to avoid decrementing the TTL, or were interfaces not crossed by traceroute. Of the remaining IP addresses, 55.9–79.4% were discovered by traceroute only.

6.5.1 Intrusiveness

We designed Sidecar to discover topology without the abuse reports of traceroute. Our CoDeeN experiment probed 22,428 IP addresses over a week with no incidents. Our web crawler experiment, however, generated ten abuse reports across 166,761 destinations. All reports noted frequent, unexpected, and synchronized accesses to robots.txt. One noted ICMP "time-exceeded" messages, but only after being alerted from web logs. We take this as indication that the Sidecar probing technique is in fact unobtrusive, but our custom web crawler needs improvement. We received no reports generated by automated intrusion detection systems. We describe our experiences in more detail in another paper [130].

6.5.2 Record Route Coverage

The record route option holds at most nine addresses; record route adds nothing to topology discovery further than nine hops from vantage points. To use record route for a broad topology discovery effort requires a measurement platform with sufficient network diversity that most of the Internet is within nine hops of at least one vantage point. By virtue of its geographic diversity, we believe that PlanetLab is, or is becoming, such a platform. Here we attempt to evaluate how well PlanetLab "sees" the broader Internet through record route, to show that record-route-based topology discovery is feasible with current infrastructure.

Record route provides additional information (aliases) or confidence in that information (multi-path detection) for nodes and links within nine hops of a vantage point. From the PlanetLab testbed, we found that 87.6–98.5% of end-hosts and routers in the data set are within nine hops of at least one PlanetLab node (Table 6.1, second to the last row). Of the links discovered in our topology 59.6–69.1% were found (or confirmed) by RR. Perhaps surprisingly, the fraction of addresses and links reachable in nine hops *increases* with the number of IP addresses. We speculate that the larger measurement set more completely explores the network that is near to PlanetLab nodes, while the measurement of the intra-PlanetLab topology discovers several paths too long for record route and little else. A full characterization of addresses outside of nine hops remains the subject of future work.

6.5.3 Correct Alias Resolution

Correct alias resolution in the context of record route requires accurate classification of routers in the path. Because the rule set is ambiguous (Figure 6.3), many traces, especially those experiencing multi-path routing or hidden routers, may be incorrectly classified. Rather than incorporate faulty data into the analysis, we remove any trace that results in a classification contradiction because the cost of erroneous data is impermissibly high. Due to the linear nature of the inference heuristics, one misclassified router may corrupt an entire trace. We believe that a more formal inference engine combined with active probing would reduce the number of ambiguous traces significantly.

Alias resolution, as reported in Rocketfuel [138] and confirmed by Teixeira *et al.* [149] is typically error prone. We use Rocketfuel's ally tool [138] to validate Passenger's asserted aliases (Table 6.2, "Alias pairs"), but we use only the IP identifier

and common source address techniques because others are too error prone [136]. We traced the false aliases (Table 6.2, "Ally: no") as reported by ally to B-to-Hidden-to-A transitions in which the Hidden interface address is falsely associated with the Type A router. We are seeking ways of resolving this ambiguity from within the trace to reduce the false positive rate further.

Many of these address pairs cannot be confirmed or disproven because at least one address is unresponsive or unroutable. Ally was only able to confirm or deny aliases for 50.6%, 18.4%, and 40.3% ("Ally:yes"+"Ally:no"/"Alias Pairs") of asserted aliases for the PlanetLab, Web crawler, and CoDeeN data sets. We report the false positive rate as the fraction of aliases disproven ("Ally: no") divided by aliases responsive ("Ally: yes"+"Ally: no").

6.5.4 MPLS Results

We use MPLS ICMP extensions to discover MPLS usage. Between PlanetLab hosts, we identify 2,546 distinct routers that advertise MPLS ICMP extensions, across 38 different ASes. Among CoDeeN hosts, there were 7,730 routers and 112 different ASes. Sidecar was not instrumented with MPLS detection at the time of the web crawler experiment. Further investigation of the possible uses of this information is the subject of future work.

6.6 Conclusion and Future Work

IP's record route option has the potential to provide more detailed topology information than previously available through traceroute. The diversity and size of the PlanetLab platform makes this primitive practical for topology measurement. We have

found 16% more addresses and discovered IP aliases that cannot be resolved by the techniques of prior mapping efforts, but at the same time, uncovered an interesting problem of inferring the aliases between the addresses discovered by traceroute and record route.

We believe TTL-limited record route packets can substantially improve the efficiency of methods like Doubletree [46] that attempt to reduce the number of probes required to collect a topology.

Sidecar is a new tool for network measurement. Its use of existing TCP connections enables unintrusive measurement and its support for IP options and ICMP extensions makes it potentially useful in developing new measurement techniques.

The code for Sidecar and Passenger, as well as the data generated from this experiment are available from `http://www.cs.umd.edu/projects/sidecar`.

The combination of Sidecar and record route as topology discovery tools provides many potential avenues of future work. The low rate of abuse reports opens the door for an in depth longitudinal study of network behavior. While we demonstrate that our heuristic solution for address assignment can classify approximately 65% of traces, we seek a more formal treatment of the problem, with solutions that can be verified. We also hope to investigate how the new information exposes anonymous routers [157] that do not generate ICMP responses for traceroute, how record route performance affects other traceroute-like measurements like RPT [67] or pathchar [69], how to assign ownership to routers with this new address information [96], how to adapt the frequency of record route probes to manage load on intermediate routers, how to use congestion control information to limit probe rates, how address alignment would benefit from other data sources like DNS or active probing, and many other questions.

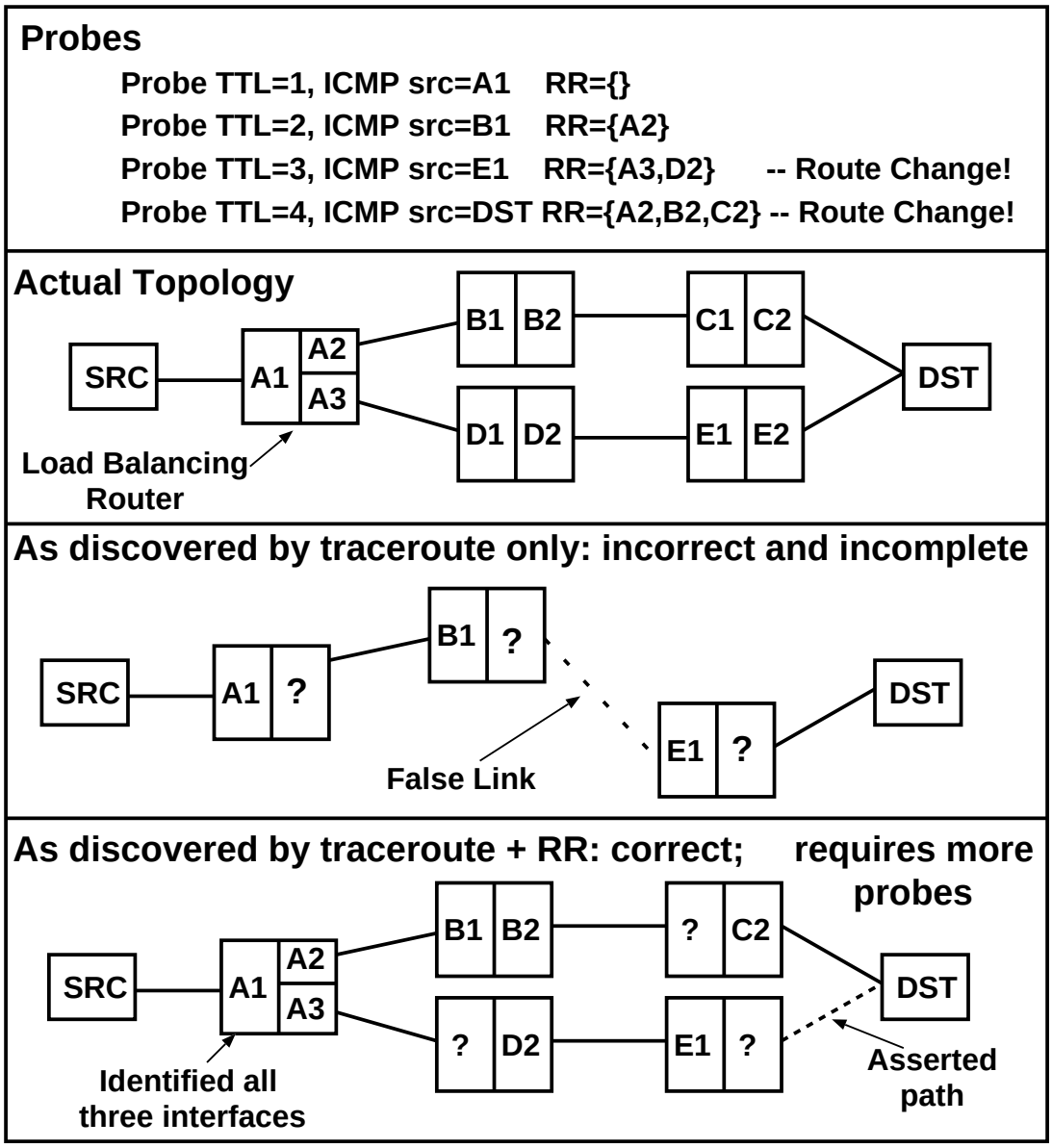


Figure 6.2: Multi-path route detection with TTL-limited record route ("A3" denoted the third interface of router A, etc.).

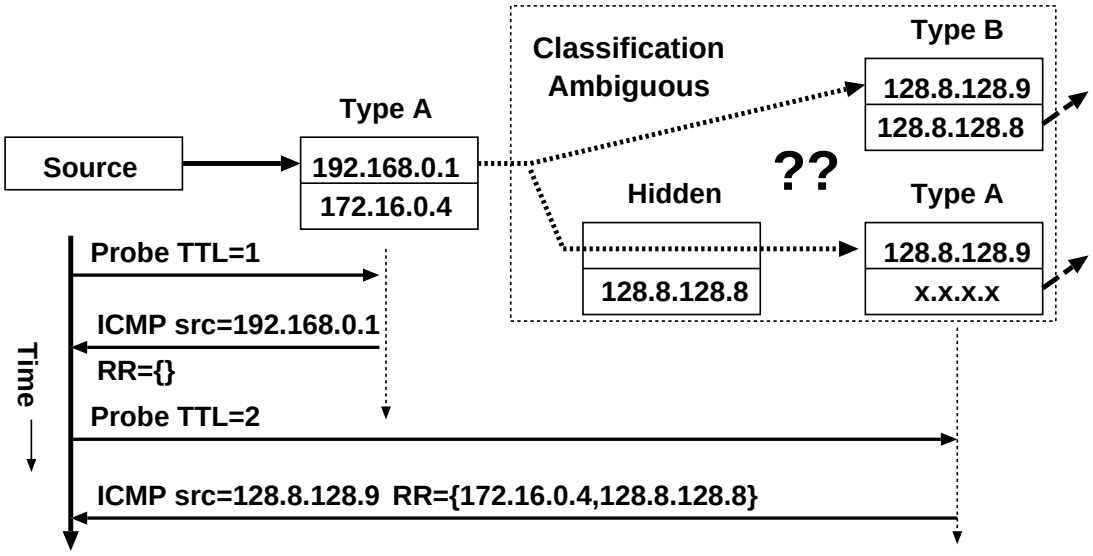


Figure 6.3: Variations in router implementations allow different topologies to generate the same trace, creating ambiguity.

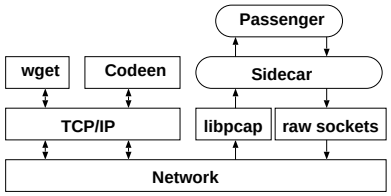


Figure 6.4: Design layout of TCP Sidecar and Passenger.

	PlanetLab		Web crawler		CoDeeN	
Autonomous Systems Traversed	331		8,739		891	
Total traces	151,688		51,742,928		13,447,011	
- Unclassified due to contradiction	35,450	(23.3%)	20,324,192	(39.2%)	1,616,079	(12.0%)
IP Addresses discovered	13,048		375,851		22,428	
- Found by Traceroute only	7,293	(55.9%)	298,455	(79.4%)	14,261	(63.6%)
- Found by Record Route only	2,059	(15.8%)	61,672	(16.4%)	3,268	(14.6%)
- Found by both	3,696	(28.3%)	15,724	(4.2%)	4,899	(21.8%)
% end-hosts and routers 9 hops from a PlanetLab Node	87.6%		98.5%		93.0 %	
% links found or confirmed with Record Route	59.5%		69.1%		65.8%	

Table 6.1: Summary of experimental results.

	PlanetLab	Web crawler	CoDeeN
Alias pairs	6,789	108,870	9,233
Ally: yes	3,389	17,972	3,291
Ally: no	46 (1.3%)	2041 (10.2%)	432 (11.6%)

Table 6.2: Router alias pairs as compared to *Ally*.

Chapter 7

Topology Analysis with DisCarte

7.1 Introduction

The global topology of the Internet allows network operators and researchers to determine where losses, bottlenecks, failures, and other undesirable and anomalous events occur. Yet this topology remains largely unknown: individual operators may know their own networks, but neighboring networks are amorphous clouds. The absence of precise global topology information hinders network diagnostic attempts [147, 95, 96, 67, 92, 78], inflates IP path lengths [128, 58, 137, 148], reduces the accuracy of Internet-models [159, 100, 75], and encourages overlay networks to ignore the underlay [13, 105].

Because network operators rarely publish their topologies, and the IP protocols have little explicit support for exposing the Internet's underlying structure, researchers must infer the topology from measurement and observation. A router-level network topology consists of two types of features: links and aliases. A *link* connects two IP addresses on distinct routers, and an *alias* identifies two IP addresses on the same router. The goal is to discover a router-level map that is both accurate—all inferred features reflect the actual topology—and complete—features are inferred for as many

pairs of IP addresses as possible.

The problem is that topology discovery techniques are error-prone. The current state-of-the-art [138, 92] uses TTL-limited probes, e.g., traceroute (TR), to infer links, and direct router probing [138, 61] to discover aliases. However, topologies inferred from these techniques are known to inflate the number of observed routers [149], record incorrect links [15], and bias router-degree [84]. These errors result from routers that do not respond to alias resolution techniques, MPLS [124], anonymous routers [157], mid-measurement path instabilities [110], and insufficient measurement vantage points. The Passenger tool [133] demonstrates that the record route (RR) IP option discovers aliases for unresponsive routers and exposes MPLS tunnels, anonymous routers, and mid-measurement path instabilities. However, RR's accuracy depends on correctly aligning RR and TR discovered IPs, itself an error-prone procedure. Passenger's preliminary work reports that almost 40% of their data could not be aligned and was unusable. Of the usable data, almost 11% of the inferred aliases were incorrect.

Unfortunately, accuracy and completeness can be at odds: for example, measuring more path data can help complete the map, but may also contribute inaccurate links. This is because topology errors accumulate—adding additional correct facts cannot "average away" a falsely asserted link. Similarly, alias inferences are transitive—a single false alias causes a cascading transitive closure of false aliases. We make the observation that if all topology data has error, then the vast data required for a complete Internet map must have a great deal of accumulated error. Thus, a topology inference system must actively remove error in order to achieve both accuracy and completeness.

Our insight is that the overall error can be reduced by *cross-validating* both TR and RR inference techniques against observed network engineering practices. For

example, a correctly implemented router would never forward packets directly back to itself, so any topology that asserts a link and an alias between the same pair of IP addresses must be inaccurate. Thus, by carefully merging three disparate sources of information, the resultant topology is both more accurate and more complete.

We present DisCarte, a novel topology data cross-validation system. DisCarte formulates topology inference and cross-validation as a constraint solving problem using disjunctive logic programming (DLP). DisCarte inputs traces from TR and RR, and, using observed network engineering practices as constraints, outputs a single merged topology. Compared to Rocketfuel-based [138] techniques, topologies produced with DisCarte find 11% more aliases from unresponsive routers, and expose additional topology features such as MPLS, router manufacturer, equal cost multipathing, and hidden and anonymous routers. Compared to Passenger [133], DisCarte correctly aligns 96% of RR and TR addresses, and reduces the false alias rate to approximately 3%. The effect of the improved topology is visually evident: we compare the topology of the popular Abilene network as inferred by Rocketfuel and DisCarte to the actual published topology (Figure 7.1).

In this chapter, we describe the qualitative benefits of DisCarte inferred topologies (Section 7.2) and the difficulties in achieving accurate topologies (Section 7.3). We then discuss the individual elements of the DisCarte system (Section 7.4) and the divide-and-conquer scheme (Section 7.5) we implement to scale DLP to the 1.3 billion facts in our system. We detail our data collection process (Section 7.6), quantify the benefit of DisCarte inferred topologies (Section 7.7), and show DisCarte's effect on bias (Section 7.8). We then conclude how one might redesign record route (Section 7.10) to aid topology discovery and describe our future work (Section 7.11).

7.2 Cross-Validating with DISCARTE

In this section, we describe the benefits of correctly merged traceroute- and RR-inferred topologies. Traceroute (TR) uses TTL-limited probes to generate ICMP time-exceeded responses from each router on a path. The source IP address of each time-exceeded message exposes an IP address for the corresponding router. The record route (RR) IP option is an array in the IP header into which each router on the path inserts an IP address. The array can store at most nine addresses, limited by the size limit of the IP header. Because of how they are implemented (Section 7.3), TR and RR discover *distinct* IP addresses for a given router. We say that a TR-trace and RR-trace have been correctly *address aligned* if each TR-discovered address has been correctly mapped to the RR-discovered address of the same router.

TR and RR can be combined into a single TTL-limited probe with the RR option set. Because an ICMP unreachable error message includes the entire IP header of the failed message, we can recover the RR array from the responses to TTL-limited probes: RR packets need not reach the destination of the probe. Thus, RR does not require the packet destination to return the RR probe, i.e., "ping -R" is not the only means of collecting RR data.

7.2.1 Benefits of Cross-Validation

Cross-validating TR and RR information against observed network engineering practices results in higher quality address alignment. Correct address alignment discovers aliases for routers that do not respond to direct probing, hidden and anonymous routers, and multi-path load balancing.

Alias resolution does not require direct probing In our survey, 193,192 of 602,136 (32.1%) IP addresses do not respond to probes addressed directly to them, preventing both IP-identifier-based matching ("ally" [138]) and source-address matching [109, 61] alias resolution techniques. Six years ago, approximately 10% were unresponsive [138], suggesting that techniques for alias resolution without direct probing will be increasingly important. We further characterize the aliases RR allows us to discover in Section 7.7.

RR exposes hidden and anonymous routers We call routers *hidden* if they do not decrement TTL and do not appear inside a traceroute; some implementations of MPLS [124] cause hidden routers. *Anonymous* routers [157] are routers that decrement TTL but do not send the corresponding ICMP time-exceeded messages: they appear as a '*' in traceroute. The absence of information from these routers is a significant source of error [157]. Out of 100,256 routers observed in our study, RR discovered IP addresses for 2,440 (2.4%) distinct anonymous routers that would have been missed by TR-only techniques. Additionally, we discover 329 (0.3%) distinct hidden routers.

RR discovers multi-path load balancing Internet Service Providers (ISPs) use multiple routes across equal-cost paths to load balance traffic. To prevent out-of-order packet arrival, load balancing routers attempt to map packets in the same flow to the same path. However, due to implementation decisions [37] in some routers, packets with IP options, including RR, break this flow-to-path mapping and traverse multiple equal-cost paths. Thus, probes with RR detect load balancing routers and enumerate additional paths more correctly than TR alone.

RR exposes mid-measurement path instability Mid-measurement path instabilities cause TR to infer incorrect links: TR assumes that sequential probes traverse the same paths. Recent techniques (Paris traceroute [15] and TCP Sidecar [131, 133])

mitigate this concern by preventing a specific class of instabilities: five-tuple load balancing multi-path. Because RR has per-packet path information, we can detect all forms of mid-measurement path changes in the first nine hops. Thus, links discovered via RR exist with higher confidence than links discovered by TR alone.

A trace between Zhengzhou University, China to SUNY Stony Brook, USA (Figure 7.2) is an example of the differences between topology discovery with and without DisCarte. Each box represents a router, and each rectangle within a router represents an interface. Lines between interfaces indicate links. We resolve DNS names of IP addresses when available, and show only the first four hops of the trace then a dotted line to the destination. The trace without RR discovers at most one interface on each router, and fails to discover any interfaces on router 3 (because it is anonymous). Adding RR to the probes and performing address alignment (Section 7.3) discovers many interfaces on each router and exposes many connections between routers, presumably for load balancing. Router labels (S1, R2, etc.) are annotated with their inferred RR implementation type (Section 7.3.1).

7.2.2 Cross Validation Limitations: RR

Many of the benefits of cross-validation rely on the RR option which has two limitations: RR includes only nine hops of data and packets with RR may be dropped or filtered. We describe each in turn.

Because the IP header can hold at most 60 bytes, RR can record only nine IP addresses. We believe the nine-hop limit is why RR has been passed over for topology discovery. Yet, there is reason to revisit this concern. PlanetLab makes available a geographically diverse set of vantage points; these may be within nine hops of much of the network. Further, our experiments use TR probes with and without RR set, so

any information gained from RR strictly increases our understanding of the topology.

Second, routers might choose to drop or filter packets with IP options. Of the 602,136 IP addresses of routers we observed within nine hops of our vantage points (that could have dropped RR), only 8,441 (1%) dropped packets with record route. We mitigate this limitation by running all traces with and without RR set.

7.3 Address Alignment

In order to achieve the benefits of cross-validation (Section 7.2), addresses discovered by TR and RR must be correctly *aligned*. Address alignment is the process of matching the IP addresses discovered by RR to the corresponding addresses discovered by TR. Accurate address alignment requires classifying the RR implementation type of each router in a trace and correctly handling tricky topology features.

7.3.1 Under-Standardized RR Implementations

The record route IP option [116] tells routers to record their IP address into a buffer in a packet's IP header. The interface that is recorded is the first source of implementation variation. Although RFC 791 states that a router should record "its own Internet address as known in the environment into which this data-gram is being forwarded," we have observed that routers record the address corresponding to the incoming, outgoing, or internal interface depending on implementation. The second implementation variation we have observed is whether the address is recorded for an expiring packet, that is, when a packet arrives with TTL=1.

We observe six different RR implementations. We describe each implementation, sorted in order of popularity, along with our best estimate of its manufacturer. An

implementation's popularity is a function of the total number of routers we were able to classify.

Cisco: 61.9% This implementation does not update the RR array for expiring packets: when a TTL=1 packet arrives at a router, the router does not add an address to the RR array. When a packet with TTL>1 passes through the router, the outgoing interface address is recorded. We associate this behavior with Cisco routers due to its popularity and private communications with Cisco engineers.

MPLS: 13.3% This implementation behaves like a Cisco router (above), except for interfaces with MPLS [124] enabled. A packet that exits an MPLS-enabled interface does not modify the RR array (similar to NotImpl, below). We know that these interfaces use MPLS because they also implement the ICMP unreachable MPLS trailers protocol [26] that returns MPLS tunnel identifiers.

NotImpl: 9.1% Some routers disable or do not implement RR. These routers pass RR probes through without modification. We believe that RR may have been previously over looked due to inflated expectation of the number of NotImpl routers.

Juniper: 7.1% In this implementation, the RR array is updated for expiring packets. Some routers record the outgoing interface, while others record the internal loopback interface. Internal loopback addresses can be distinguished from outgoing addresses by hand, for example, if the reverse DNS look-up of the address contains the string "lo-". We believe this RR-type corresponds to Juniper due to its appearance in the Abilene network which uses Juniper routers [4].

Lazy: 5.8% These routers do not decrement TTL for packets with the RR option set, and instead allow the packet to continue to the next hop. This caused significant confusion in our initial experiments using interleaved packets with and without

RR set. Publicly available router configurations at National Lambda Rail (NLR) suggest that Cisco's Carrier line of routers are Lazy. Of all RR implementations we have observed, this is the only one that would seem to violate RFC 791.

Linux: 2.7% Routers that implement Linux-based IP stacks do update the record route array for expiring packets, but instead of adding the address corresponding to the outgoing interface, they use the address of the incoming interface. However, for packets that do not expire, the outgoing interface address is used.

With the exception of the Lazy RR implementation, we believe that these implementation variations correctly implement the RR specification as described in RFC 791. The variations in implementation arise because RR is underspecified, and we recommend additions to the specification (Section 7.11). Also, note that the "Flaky" RR implementation, first identified by Sherwood and Spring [133], does not appear to exist. We believe that Flaky is a combination of the Lazy implementation type above and equal-cost path routing of different hop counts.

7.3.2 Topology Traps

We identify six topology features that complicated accurate topology discovery. In this section, we catalog these features to show the complexity inherent in topology discovery and motivate the need for an automated inference tool.

Hidden routers do not decrement TTL, and are thus are not detected by TTL-limited topology discovery. Hidden routers are caused by certain configurations of multi-protocol label switching [124] (MPLS), and result in missing nodes and incorrect link inferences. As with anonymous routers, the RR IP options can be used to detect hidden routers if supported. Also, the use of MPLS can be

detected by an optional MPLS tag attached as a footer in TTL-exceeded messages [26]. We discovered 329 hidden routers in our experiments.

Non-standard firewall policies introduce varied sources of error. In one case, a firewall in China forges TTL-exceeded messages from the destination [133] for packets with the RR option set. Also, we have observed firewalls that send ICMP source quench, ICMP parameter problem, and ICMP administratively prohibited messages. Each of these behaviors must be identified and removed from the data before processing.

Enabling IP options breaks load-balancing, spreading a single flow across multiple equal-cost paths. Five-tuple load-balancing uses the source and destination IP and port fields, along with the IP protocol to identify a flow, and maps all packets in the same flow to the same path [15]. However, adding IP options to packets with the same five-tuple signature breaks this scheme. We hypothesize that some router implementations fail to account for IP options when calculating the packet offset to the TCP/UDP source and destination port fields when computing the 5-tuple. In other implementations, packets with IP options are routed on arbitrary equal-cost path. Both behaviors add to the complexity of address alignment.

Different-length equal-cost paths can create false links. Equal-cost paths may have different hop-count lengths. This results in multiple sets of probes, offset in TTL, for the same routers along the path. We use RR to partition probes by the path they traversed, and only compare probes that take the same path. By partitioning probes by the paths that they traverse, we remove one source of self-loops common to traceroute-inferred topologies [15]. Traces from Cornell University to PlanetLab nodes in Amsterdam have this behavior (Figure 7.3).

RR fills. The address alignment algorithm monitors hop-by-hop increases in the size

of the RR array to classify each router's RR type (Section 7.4). Because a given hop may add more than one entry into the RR array when the RR array fills up—reaches nine entries, the information about the true number of RR entries for this hop is lost. For example, a packet with eight RR entries that transitions from a Cisco RR-type router to a Juniper RR-type router, would normally receive two new RR entries. However, since there is only space for one more IP address, the second entry is lost. The address alignment algorithm has to consider more possibilities when the RR array fills. DisCarte's DLP code base doubles in size to handle this seemingly simple case.

Persistent Routing Loops can prevent naive trace collection from terminating. Our data collection scripts had to be rewritten to detect loops. We revisited the looping paths three weeks later and found that approximately half still persisted. We further characterize the routing loops discovered in Section 7.6.3.

7.3.3 Ambiguity in classification

The variety of RR implementations make router classification ambiguous. Because different topologies and router implementations can generate the same trace (Figure 7.4), a router may be misclassified, leading to mismatched addresses and aliases. Thus, in the same trace, the IP X might be an alias for IP A or B depending on the RR implementation. Further, the third probe discovers two new RR addresses (Y, Z) and it is ambiguous whether IP address Y belongs to a Hidden router. We depict two of 15 possible interpretations of the trace.

A single mismatched pair of addresses causes cascading errors as each subsequent RR address in a trace is misaligned. However, using observed network engineering practices it is possible to correctly match RR and traceroute discovered addresses (Sec-

tion 7.4.4). For example, network engineers tend to allocate IP addresses on either end of a link out of a /30 or /31 network [65, 92], so the topology that best matches this pattern is most likely correct.

7.4 DISCARTE

Large-scale cross-validation and address alignment is difficult and error-prone, not only because of the need to infer the different RR implementations of routers, but also due to complex network topology features (Section 7.3).

Our system, DisCarte, uses disjunctive logic programming (DLP) [32, 123, 86], a constraint solving technique that, to the best of our knowledge, has not been used for topology discovery. DLP has the ability to describe a low-level set of inter-dependent interactions while simultaneously shaping the solution to match high-level constraints. For example, we instruct DLP to find the set of RR implementations such that the link and alias assignments do not cause routers to have self-loops. The DisCarte process (Figure 7.5) consists fact generation (Section 7.4.2) and fact processing (Section 7.4.3 – 7.4.5) phases. We describe our data collection phase in Section 7.6. In this section, we provide a brief description of the DLP technique, describe how we transform raw topology data into DLP facts, and present the DisCarte address alignment algorithm and its corresponding cost function.

7.4.1 DLP Introduction

DLP is a formalism representing indefinite information. Superficially similar to Prolog, language statements consist of facts, inference rules, and weak and strong con-

straints. Inference rules are *disjunctive*—they are of the form:

$$\text{fact}_1 \textbf{ or } \text{fact}_2 \textbf{ or } \ldots \textbf{ or } \text{fact}_n \Leftarrow \text{fact}_0 \tag{7.1}$$

indicating that fact_0 implies exactly one fact in the set of facts $\text{fact}_1 \ldots \text{fact}_n$. Because each inference rule can potentially imply many different facts, a disjunctive logic program has many possible solutions, or *models*. Potential models are then pruned by strong and weak *constraints*. Any model that violates a strong constraint is removed from the solution set, and the remaining models are assigned a numeric cost based on the weak constraints they violate. The output from a DLP is the lowest cost model of inferred facts generated from input facts and inference rules.

The specific DLP implementation we use is DLV [86]. The language restricts how constraints are specified, to preserve the *monotonicity* property of the cost function: that adding new facts can only increase that cost of a model. DLV uses this property to prune high cost sub-trees from the solution space. DLP can efficiently represent complex problems, for example, the formulation for the graph 3-color-ability problem [59] is two lines long [45].

7.4.2 Data Pre-processing

Raw trace data must be converted into facts for DLP. These facts consist of both straightforward parsing of the data, deriving facts more easily computed without DLP, and *probe pairs*. Here, we describe the facts computed in the pre-processing step.

Some network topology features can be identified statically without DLP. Routers with the Linux RR implementation have a simple signature: the response to a TTL-limited probe comes from router X, and the last entry in the RR array is also X. Similarly, we declare a router X to be Lazy if all non-RR probes with TTL=t return

ICMP time-exceeded responses from X, all RR probes with TTL=t return responses from a different router Y, and all non-RR probes with TTL=$t+1$ return from router Y. Responses to RR probes from non-standard firewalls have the return address set to the probe's destination, instead of the router's interface. Once these network features have been detected, we correct for them as we identify probe pairs.

Two TTL-limited probes form a "probe pair" if one probe expires at router X, and the other probe goes through X and expires at the next TTL. Each probe pair fact is of the following form: "probePair(p_1,p_2,$delta$)", where p_1 and p_2 are unique probe identifiers, and $delta$ is the difference between the size of the two RR arrays. By convention, p_2 is the probe that went one TTL farther. Identifying probe pairs in non-RR (traceroute-only) data is trivial but error prone: mid-measurement path instabilities (Section 7.2) can cause sequential probes to take different paths. When adding RR to probes, probe pair identification becomes more accurate—RR probes record the traversed path—but more complicated. Lazy RR implementations and multi-path routing with different length paths complicate probe pair identification. For example, after passing through a Lazy router, TTL-limited probes with RR set go one hop farther then intended. Before we can try to identify probe pairs in the presence of a Lazy router, all probes with RR that pass through that router must be re-normalized as if they were sent from the subsequent TTL. Also, if there is evidence of multi-path routing with different length paths, we must be careful to only compare probes that took the same length path. Last, if a trace has both Lazy routers and different length paths, we can only identify Lazy routers on the path taken by the non-RR probes, so information on the other path must be discarded.

7.4.3 Address Alignment with DLP

Though an exotic choice, DLP lends itself well to the address alignment problem. For each trace, the pre-processor will output a set of potentially over-lapping probe pairs such as "probePair(X,Y,$delta_1$)" and "probePair(Y,Z,$delta_2$)". The job of the DLP is to infer the most likely RR implementation type assignments that are globally consistent: router Y must have the same RR type in all of its probe pairs. Then, based on the type assignments, DLP outputs link and alias facts that form a topology.

To constrain the assignment of implementation types to those consistent with the probe pair facts, we express DLP inference rules that describe each possible transition from router to router for each $delta$ in a probe pair. The $delta$ is the number of additional RR entries in the second probe of the pair, and may be any number from 0 to 9, though we have not observed a delta greater than 4. An RR entry will be added if (a) leaving a Cisco router, (b) arriving at a Juniper or Linux router, or (c) traversing a hidden router. Traversing a NotImpl router (or entering an MPLS tunnel, which has the same effect) does not add to the delta. The inference rule is the list of possible RR-type transitions that would result in a probe pair with the same $delta$. For example:

$$\begin{aligned} transition(X, Y, Cisco, Cisco) \quad & \textbf{or} \\ transition(X, Y, Juniper, Juniper) \quad & \textbf{or} \\ transition(X, Y, Cisco, NotImpl) \quad & \textbf{or} \\ transition(X, Y, NotImpl, Juniper) \quad & \textbf{or} \\ transition(X, Y, NotImpl, Hidden, Cisco) \quad & \textbf{or} \\ transition(X, Y, NotImpl, Hidden, NotImpl) \quad & \textbf{or} \\ \Leftarrow probePair(X, Y, delta), & \\ delta = 1. & \end{aligned} \tag{7.2}$$

indicates that if we find a probe pair with $delta = 1$, then the transition between the router corresponding to the first probe (X) and the router corresponding to the second probe (Y) is a transition from a Cisco-type RR router to another Cisco-type RR router *or* from a Juniper-type RR type router to another Juniper-type RR router, etc. We must include more atypical transitions, such as from a router that does not implement RR (NotImpl), through a router that does not show up in traceroute but implements RR (Hidden), to another router that does not implement RR (NotImpl). We wrote DLP inference rules for delta=$0 \ldots 4$, and show the possible transitions from 0 through 2 in Table 7.1. We also implement a duplicate set of all rules where the RR array is full (Section 7.3.2). Thus for a probe with full RR array and $delta = X$, all possible transitions for $delta \geq X$ must be considered.

Multiple possible transitions per probe pair and independent computation of probe pairs imply that there are potentially exponentially many models relative to the number of probe pairs. We discuss the cost function for intelligently pruning this set to produce the best model (Section 7.4.4) and our divide-and-conquer technique for scaling this algorithm (Section 7.5).

7.4.4 Engineering Practices and Cost Function

Recall from Section 7.4.1 that DLP supports strong and weak constraints: models that violate strong constraints are removed and the rest are ordered by degree of weak constraints violated. DLP outputs the lowest cost model.

The only strong constraint in the DisCarte system is that a router's RR implementation must be consistent across all its interfaces. In other words, it is never the case that the same router uses the Cisco RR behavior for one interface and Juniper RR behavior for another interface. A potential issue with this rule is the MPLS RR type, where

individual interfaces might appear to be of RR type Cisco or NotImpl. The strong constraints are carefully written to handle this exception.

Weak constraints are chosen based on observed patterns which we believe correspond to network engineering practices. Each practice should hold as a general rule of thumb, but may be violated in an individual solution. Thus the model that violates the fewest practices is likely to be the closest approximation of reality. Here we list weak constraints in order of importance.

1. There should be no self-loops. We believe a correctly-implemented router would never route packets directly back to itself. Avoiding this condition prevents situations where two distinct routers are merged by a single bad alias, and conversely when a link is incorrectly added between interfaces on the same router [15].
2. Many IP addresses on either end of a link are adjacent in IP space: they are "off-by-one." We expect that network architects try to conserve address space by using the smallest network blocks available, either /30 or /31. The implication is that models where the IP addresses of links are off-by-one should be preferred over those without. Gunes *et al.* use this technique to infer aliases directly [65]. Figures 7.2 and 7.3 show this behavior.
3. Aliases inferred by direct probing (ally [138]) are often correct. The validity of direct probing techniques [142, 61] has been independently demonstrated, so that information should be used when available. However, due to temporal changes in topology or potential for inaccuracies in the technique, information from direct probing is not guaranteed, thus it remains a weak constraint.
4. Hidden routers are rare, so of two equally-likely models, the solution with the fewest hidden routers should be preferred. We derive this rule from observation of out-of-band data, such as DNS naming conventions and /30 and /31 IP

addressing in links.

5. Routers supporting RR are more common than those that do not (NotImpl). We verify this empirically by observing that with each new TTL, subsequent probes in a trace typically record new RR entries.

The cost for a model is assigned based on the number of practices violated, weighted by the importance of the practice. We experimented with different weight assignments, but as long as the relative importance of practices remained as above, the weight assignment did not affect the final solution. Also, it is possible for DLP to output multiple equal-cost models, if there is insufficient information to make an alignment, or no model at all, if there is an error in the data or flaw in our model. We next address both points further.

7.5 Scaling and Conflicts

DLP alone does not scale to Internet-sized topologies, as the number of possible RR implementation assignments grows exponentially with the number of probe pairs. Our top-level approach is to process the data in pieces large enough to provide the correct solutions, yet small enough that they are solved quickly—divide and conquer. Merging processed pieces back together can expose *conflicts*: that the same pair of IP addresses are believed to be both aliased and linked. In this section, we describe a data partitioning method that reduces conflicts and engineer a technique to resolve conflicts once they occur.

7.5.1 Divide and Conquer

Dividing the data is easy; dividing the data while preserving enough information for DLP to produce meaningful results is difficult. Our first approaches at partitioning the data produced a scalable execution—one trace per run, or many traces from the same source—but they resulted in many incorrect inferences. Because each run interpreted only the data from probes leaving the source, the DLP solver missed potentially conflicting data from measuring the return path.

To provide a core of correct, reliable address alignments and router implementation inferences, we start by computing all two-cliques—the trace from site X to Y with the trace from Y to X—as shown in Figure 7.6, left.

Atop this core, we process triangle-like subsets of all traces between pairs of sources and a destination (Figure 7.6, right). The insight is that the path between the source pair has already been computed and found to be free of conflicts, so it is reliable. By using this approach, we reduce the number of unresolved conflicts—those conflicting inferences that remained after all processing—from 1,547 to 28 in the PlanetLab data-set.

We hoped to process all possible triangle subsets for maximum overlap and thus maximum cross-validation, but with 379 sources and 376,408 destinations, this task is intractable. Instead, we processed the 71 million non-overlapping triangle subsets on a 341 processor (heterogeneous) Condor [91] cluster. Triangle subsets typically take a second to process, though the execution time is highly variable. The Condor scheduler estimates that we have used 96,225 hours or approximately 11 CPU years on this project (including time spent debugging).

7.5.2 Unions and Conflicts

We extract the facts in the models produced by the divide and conquer phase and search for contradictions. A contradiction appears when two addresses that are thought to be aliases are seen to be linked in a subset of facts. (Two IP addresses can be assigned to the same router if they are aliases of aliases, so the alias inference can result from several sets of facts; a link cannot be synthesized from different traces–see Section 7.4).

To resolve conflicts, we pick an arbitrary model from each faction (those indicating link and those indicating alias) and run both input subsets together through DLP. If the result contains exactly one model, then the conflict is resolved, and we record whether the IP addresses are linked or aliased as a *hint*. Once the hint is recorded, all affected models are recomputed via DLP.

The conflict resolver can fail to resolve a conflict if the DLP outputs multiple models with both link and alias facts asserted, or no model at all. Having multiple models indicates that we have insufficient information to resolve this conflict, whereas producing no models indicates an error in the input or a potentially new RR behavior. In any case, if the conflict resolver cannot resolve the conflict, then all facts associated with the two IP addresses are removed from the model. In our experiments, 12,731 of 9,793,309 (0.13%) of subsets produce no valid model, and 22,095 of 1,021,027 (3.7%) of facts remain unresolved. It is the subject of our future work to characterize the unresolvable traces and improve the conflict resolution process.

7.6 Data Collection

We collect two sets of topology data to validate DisCarte: one between PlanetLab nodes and the other from PlanetLab nodes to all advertised BGP prefixes. For both, we perform TTL-limited traceroute-like probes with and without the RR option set. For the BGP prefixes data set, we also use the "stoplist" technique to avoid probing destinations in a way that might appear abusive. We conclude by reporting on the distribution of stable routing loops that we discover in our experiments.

7.6.1 Data Sets

The PlanetLab [114] data set is an all-pairs trace, from all PlanetLab nodes to all other PlanetLab nodes. This repeats the Passenger [133] study. Because some PlanetLab nodes were unavailable, we were able to collect data from only 387 nodes.

In the BGP data set, we probe 376,408 destinations. To generate the destinations, we divide each advertised BGP prefix [101] into a /24 sub-prefix, choose a representative address from each by setting the last bit, and then remove unresponsive IP addresses. This IP generation strategy is similar to iPlane [92], except that we disaggregate larger prefixes down to /24-sized sub-nets.

We probe using traceroute's increasing TTL, alternating probes with and without the RR option set, three times with each. We stop probing a destination after probes for six sequential TTLs have been dropped. Due to firewalls that drop probes and source nodes rebooting, we do not have data for all sources to all destinations, but we do collect approximately 1.3 billion probe responses.

7.6.2 Stoplist Probing

We believe that RR probes are more likely to generate abuse reports then other topology discovery techniques. The RR option is rare and intrusion detection systems target anomalous events. However, we note that network mapping need not probe destination hosts often: careful measurement coordination can avoid reports of abuse. Our insight is that we can avoid probing destination networks from every source by noticing when the path from a new source merges with an already-observed path.

The goal of our "stoplist" technique is to give each destination a *red zone*: a region close to the destination that will not be probed from machines outside our control. A stoplist is a per-destination list of the last k IP addresses on a trace to the destination. We generate the stoplist from a single host under our administrative control, so potential abuse reports can be handled locally without involving PlanetLab support. To generate the stoplist, we run a reverse traceroute to each destination and record the last $k = 3$ hops. A reverse traceroute works by guessing the TTL distance to the end host, sending a probe, and then searching with larger TTL if the destination was not reached, or with smaller TTL if it was. Once the TTL distance to the end host is known, the last three hops are determined by decrementing the TTL.

The stoplist is then distributed among sources, and as each node traces towards a destination it stops when an IP on the stoplist is discovered. Using this technique, we received no abuse reports either in generating or using the stoplist.

7.6.3 Routing Loops

Routing loops are a symptom of network misconfiguration and can frustrate topology inference. DisCarte found a surprisingly high number of routing loops: 8,550 source-destination pairs contained a persistent routing loop and prevented packets from reach-

ing the destination. These pairs were re-tested three weeks later. We were not able to retest 2,071 (24.1%) of these pairs, because the configuration of the source nodes had changed. At the later date, 4,501 (52.6%) of the loops were resolved, while 1,976 (23.1%) remained.

Of the routing loops that persisted through both tests, 689 unique routers appear 4,544 times in some part of a loop. China Railway Internet (CRNET, AS 9394) has more of these routers (61) than any other AS. Korea Telecom follows with 47 routers, and Level 3 with 35. When weighted by the number of traces that traverse these loops, almost 10% of the routers again belong to CRNET, almost twice as many as the next-most frequent location, Frontier Communications of America (AS 5650).

7.7 Validation

In this section, we validate the output of DisCarte in terms of accuracy and completeness. We first compare the aliases produced by DisCarte to those produced by Rocketfuel's ally tool [138]. Then, we compare the routers, links, and degree distribution of topologies inferred by DisCarte and the Rocketfuel and Passenger techniques against four published topologies.

Of course, any active IP-probing methodology will suffer from inherent shortcomings: that backup paths and link-layer redundancy are visible, and that multiple-access network links are not differentiated from point-to-point links. DisCarte does not address these problems, so we do not consider them further.

7.7.1 RR Aliases

We use the IP-identifier [92, 138] and source-address matching [109, 61] alias resolution techniques to verify the aliases inferred by DisCarte. DisCarte over the BGP-prefixes data set found 374,337 aliases, 42,284 (11.2%) of which were not found by direct probe-based techniques in Rocketfuel's ally.

We then re-applied ally to confirm the aliases asserted by DisCarte: 88.3% were confirmed to be correct, 3.8% were claimed to not be aliases by the IP-identifier technique, and the remaining 7.8% were from unresponsive routers and could not be confirmed. 91.2% of the aliases found by DisCarte involved IP addresses discovered only by adding the RR option.

7.7.2 Comparison to Published Topologies

Research networks including Abilene[1], Géant[2], National LambdaRail (NLR)[3], and Canarie (CANET)[4] publish the configuration files of their routers, which makes determining a "correct" topology possible. We compare DisCarte's inferred map to these true topologies as well as the topologies produced by the Rocketfuel [138] and Passenger [133] techniques. To build the correct topology, we parse the "show interfaces" information available for each router from each network's web site. We use publicly available software to generate Rocketfuel[5] and Passenger[6] topologies. In each network,

[1] `http://vn.grnoc.iu.edu/xml/abilene/show_interfaces.xml`

[2] `http://stats.geant2.net/lg/process.jsp`

[3] `http://routerproxy.grnoc.iu.edu`

[4] `http://dooka.canet4.net`

[5] `http://www.cs.washington.edu/research/networking/rocketfuel/`

[6] `http://www.cs.umd.edu/projects/sidecar`

we consider the number and accuracy of discovered routers, the degree distribution, and completeness of the discovered links.

For each network, we classify each inferred router into one of four accuracy categories: good, merged, split, and missed.

Good: There is a one-to-one mapping between this inferred router and a router in the correct topology. All of this router's discovered interfaces are correctly aliased. In a perfectly inferred topology, all routers would be good.

Merged: There is a one-to-many mapping between this inferred router and routers in the correct topology. In this case, multiple real-world routers are incorrectly inferred as a single router. Merged routers result from inaccurate alias resolution, artificially deflate the router count, and inflate the node degree distribution.

Split: There is a many-to-one mapping between routers in the inferred topology and a single router in the correct topology. In this case, a single router from the correct topology appears split into multiple routers in the inferred topology. Split routers result from incomplete alias information, inflate the router count, and deflate the node degree distribution.

Missed: This router was not found: none of the router's interfaces were discovered by the inferred topology. Missing routers result from insufficient vantage points or from data discarded due to unresolved conflicts (Section 7.5.2). Missing routers deflate the router count and bias the node degree distribution towards observed routers.

Classifying the number of inferred routers by accuracy (Figure 7.7) illustrates three interesting characteristics. First, although all three inference schemes tend to have substantial numbers of "split" routers, Rocketfuel has so many split routers that it incorrectly over-estimates the router count by as much as seven times the true value.

This is a result of routers that are unresponsive to direct alias probing, so no aliases are found (recall Figure 7.1). So, although aliases from DisCarte result in a more accurate topology, more complete alias resolution techniques are still required. Second, for each topology, DisCarte has more "good" nodes than other techniques, except for Passenger in the Géant network. In this exception, Passenger finds two more "good" nodes then DisCarte, at the cost of four incorrectly merged nodes. We demonstrate below that the presence of merged routers alters the topology's degree distribution. Third, DisCarte-inferred topologies have no merged routers and fewer split routers than Rocketfuel.

Next, we consider the degree distribution of inferred topologies. Degree distribution affects the accuracy of Internet-modeling [93] and path diversity studies [149], and has been studied in its own right [47]. We plot the degree distribution of the topologies inferred by Rocketfuel, Passenger, and DisCarte along with the actual degree distribution for each published topology (Figure 7.8). In all networks, the DisCarte inferred topology most closely tracks the actual degree distribution relative to the other two techniques. Also, the effect of merged routers on the degree distribution is apparent: Passenger deviates significantly from reality in the Géant data set due to the four merged routers it infers.

Of four published topologies, our inferred topology has *no false links* (Table 7.2), and discovers at least 63% of existing links. We believe the only way to improve the completeness of the link coverage is to increase the number of measurement vantage points and their network diversity.

Comparison to research networks at first does not appear inherently challenging: their openness, homogeneity, and proximity to most PlanetLab vantage points make them relatively easy validation cases. However, each research network is distant from several vantage points, which are often behind interesting configurations (specifically

those sites in China and Israel) that can introduce false links and aliases. Further, routers of specific research networks (Abilene,NLR,CANET) do not respond to alias resolution probes, which confounds topology inference.

7.8 Topology Analysis

In this section, we consider the degree distribution and sampling bias apparent in our DisCarte-inferred topology. We chose these properties because they could be affected by the missing or erroneous aliases.

Lakhina *et al.* [84] introduce a method for evaluating measured network topologies to see sampling bias in the degree distributions of routers. The fundamental assumption is that high-degree routers are equally likely to be anywhere in the topology, and specifically, are no more likely to be near to the sources than farther away. A biased sample would tend to see many of the links incident to nearby routers, because the shortest path tree from a source is more likely to include the links of nearby routers, and less likely to include more than two links on distant routers.

We repeat the analysis of Lakhina *et al.* and find sampling bias in both DisCarte- and Rocketfuel-inferred topologies. In Figures 7.9 and 7.10, we show the complementary cumulative distribution of the router out-degrees of routers in the near set (those within the median distance from a vantage point), in the far set (those of median or greater distance), and overall. That the near set has somewhat higher degree demonstrates sampling bias in the topology. This suggests that more data rather than higher-quality topologies are required to remove bias.

We expect sampling bias to be present in the topology we measure. The best approach to eliminating such a bias is most likely to wildly increase the number of van-

tage points relative to the destinations as performed in Rocketfuel [138]. Even when doing so, sampling bias is not eliminated: Lakhina's test found bias in all studied topologies.

7.9 Related Work

We classify related work into network mapping techniques, measurement-based inferences, error-avoidance in traceroute, and error characterization in network maps.

7.9.1 Internet Mapping

Techniques for Internet mapping present various techniques for selecting traceroute measurements and resolving aliases. In 1995, Pansiot and Grad [109] pioneered network mapping by tracerouting to approximately 3000 destinations and introduced alias resolution by source address. Govindan and Tangmunarunkit's Mercator [61] revolutionized mapping through source-routed probes, alias discovery by source routing, and validation against real-world networks CalREN and Los Nettos. Rocketfuel [138] sought fidelity of ten ISP maps by exploiting traceroute servers and added alias resolution by IP address. Skitter [38] and iPlane [92] apply many of these techniques continuously, making a current reference topology available to researchers.

DisCarte is comparable to these projects in that it introduces a new and more complete technique for improving the correctness of the network mapping and a novel method for alias resolution. We approached correctness in the measured topology by measuring each path using two methods (RR and TR) so that we can detect and remove disputed conclusions. These features are crucial to continued network instrumentation because (a) security concerns cause administrators to filter traffic destined for routers

and (b) the scale of the network demands such a large scale measurement that some collected traces are certain to have errors, and unlike in the natural sciences, these errors are not averaged out by further measurements.

7.9.2 Learning and Inference Techniques

Techniques to interpret raw network measurements are sometimes required; these often involve learning techniques to manage the scale of the problem. Padmanabhan used Gibbs sampling and Bayesian learning to discover lossy links [107]. Mahajan *et al.* [94] used linear constraints to model intra-domain link weights: a study that could imply a means of detecting and removing false links (those that have too high a cost to be used). Yao *et al.* [157] present a technique for merging anonymous routers—those that do not respond to traceroute that might otherwise be ignored in a topology (potentially partitioning the network) or thought unique on each observation (wildly inflating the path diversity). Finally, Gunes and Sarac [65] use the addressing structure of the network to deduce the prefixes to which interface addresses belong and infer aliases.

7.9.3 Traceroute Error Avoidance

Although we apply the paired measurement of TR and RR to bolster uncertain measurements, an alternate, but complementary, approach is to reduce the likelihood of error in the first place. Augustin *et al.* [15] observe that router load balancing is typically flow based: to restrict a traceroute to a single path requires only re-designing the tool to preserve the five-tuple of protocol with source and destination address and port.

Sherwood and Spring showed that RR had the potential to detect route changes and could be applied toward a more reliable traceroute [133], but found that simple (deterministic) methods became intractable and was unable to process almost 40% of

their data.

7.9.4 Network Map Errors

Some errors in network maps may be avoided through improved techniques. Teixeira *et al.* [149] noted a lack of fidelity in Rocketfuel maps between incomplete measurement (backup links were missing) with incomplete or erroneous alias resolution (some addresses were split and merged); in estimating path diversity, these factors canceled each other somewhat, but the result was not reliable.

7.10 Record Route Redesign

If the record route IP option were designed today, it would benefit from more precise standardization and the ability to sample paths longer than 9 hops.

Address alignment would be trivial if record route implementations were standardized (and such standards were adhered to). We believe the implementation diversity in record route (Section 7.3) is there because RFC 791 does not specify how to treat options on expiring packets. For topology discovery, the most appealing RR implementation is that of Juniper, where addresses are recorded for expiring packets. If this scheme were universal, an alias could be discovered with a single packet.

A more powerful record-route option would include the ability to "skip" a configurable number of addresses before starting to record. In this way, successive RR probes could record 9 hop subsections of a path, giving complete RR information from end to end, as opposed to the current 9 hop limit. Implementation is simple: routers need only increment the RR array index pointer even if the RR array is full, allowing the index to wrap. Thus, the sender sets the initial RR pointer value to $4 - (4 \times k) \mod 256$ to

skip k hops before starting to record the route. Recall that a router along the path only records the route if the pointer value p is in the range $4 \leq p < l$ where l is the length of the RR option in the IP header.

7.11 Conclusion

Internet topology measurement faces a continuing problem of scale: more nodes and links are added; measurement platforms like PlanetLab grow; and filtering policies and implementations remain diverse. To capture this topology requires not simply the ability to collect, store, and query against the 1.3 billion response packets in our data-set, but also the ability to filter this data to discern which observations and interpretations are valid. Toward this goal, we adopted disjunctive logic programming to merge our expectations of network engineering practice—common vendor choice and address prefix assignment—to interpret and merge our topology data.

DisCarte provides a novel cross-validation tool for network topology discovery—it finds aliases that increasingly cannot be detected by active probing (30% of addresses we found could not be probed), it finds routers that do not decrement TTL (329) or generate ICMP errors (2,440), it verifies that probed paths are consistent during a measurement—but extracting this information requires significant effort. Expectations of network engineering practice provide the hints required to interpret this data accurately, and a divide-and-conquer approach allows the flexible interpretation to take place quickly over subsets of the data and resolve contradictions.

Our effort owes its inspiration to Vern Paxson's *Strategies for Sound Internet Measurement* [113]. Our approach that led to DisCarte is to measure the same path and topology using two different methods so that their consistency can ensure an accurate

result (one of Paxson's "calibration" strategies). Along the way, we adopted many of his hints: study small components first (the PlanetLab topology before the Internet; small cliques before larger ones), invest in visualization (we used neato and dot [62] to compare topologies), build test suites (our regression tests include 77 difficult-to-interpret traces and groups of traces), and we will publish our data and analysis scripts.

Our future work is to develop two related components: an application-specific version of our (inefficient but general-purpose) DLP-based solver, and a more efficient measurement interpretation scheduler that would choose to study related measurements together to reduce the computational requirements of the analysis. In this first application of record route in topology measurement, getting the right answer took precedence over performance; making the measurements and analysis efficient enough to be repeated will take engineering.

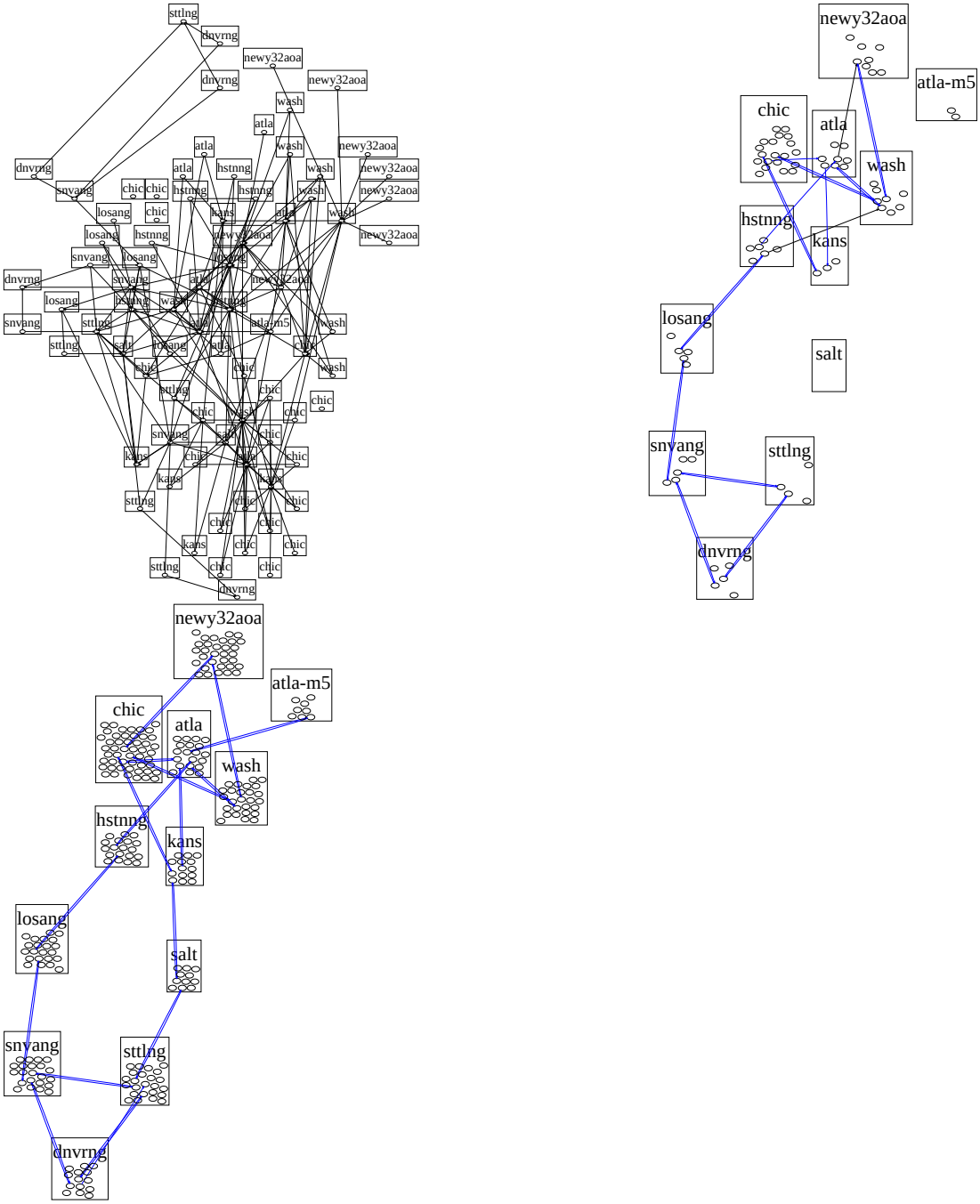


Figure 7.1: Abilene topology: inferred by Rocketfuel (left, routers unresponsive to direct alias resolution), DisCarte (middle), and actual topology (right). Rectangles are routers with interior ovals representing interfaces.

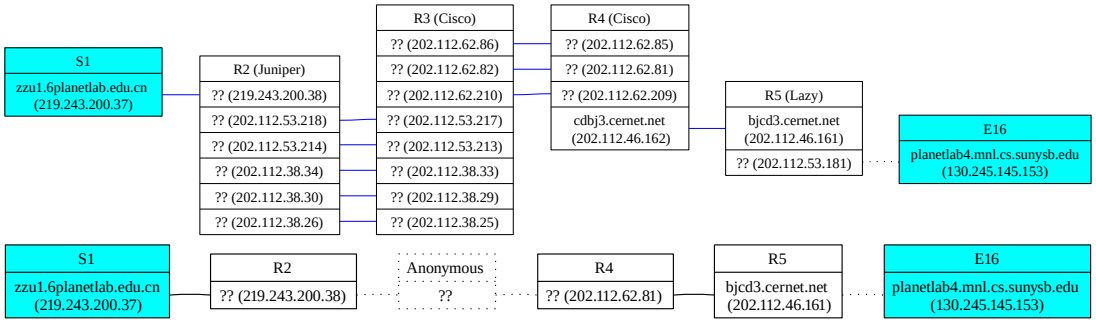


Figure 7.2: Partial Trace from Zhengzhou University, China to SUNY Stony Brook, USA; inferred by DisCarte (top) and Rocketfuel-techniques (bottom). DisCarte finds many load-balanced paths through an anonymous router (R3) and helps determine the implementation class of each device along the path.

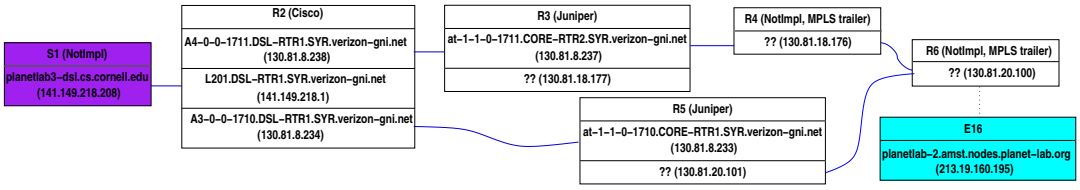


Figure 7.3: Partial trace from Cornell to Amsterdam where probes that take different-length paths: bottom path is one hop shorter then top.

Probe ttl	ICMP source IP	RR Array
1	A	-
2	B	X
3	C	X,Y,Z

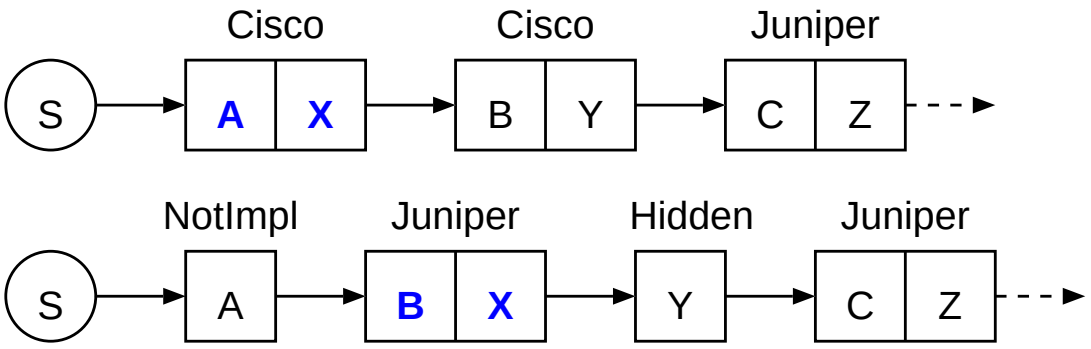


Figure 7.4: Varied RR implementations create ambiguous alignments between IP addresses discovered by TR probes (A, B, C) and those discovered by RR (X, Y, Z). We show two of fifteen possible topologies inferred from a partial hypothetical trace from source S: rectangles represent routers and letters are IP interfaces. RR *delta* is the number of new RR entries since the previous TTL.

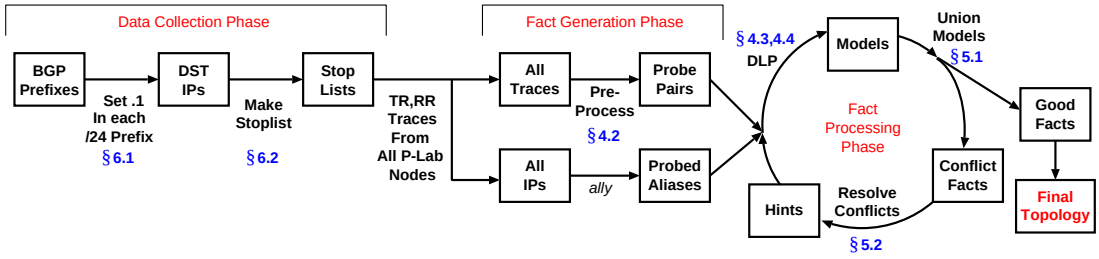


Figure 7.5: Overview of the DisCarte Topology Inference System.

delta=0	delta=1	delta=2
NotImpl → NotImpl	NotImpl → Hidden → NotImpl	NotImpl → Hidden → Hidden → NotImpl
NotImpl → Cisco	NotImpl → Hidden → Cisco	NotImpl → Hidden → Hidden → Cisco
Juniper or Linux → NotImpl	Juniper or Linux → Hidden → NotImpl	Juniper or Linux → Hidden → Hidden → NotImpl
Juniper or Linux → Cisco	Juniper or Linux → Hidden → Cisco	Juniper or Linux → Hidden → Hidden → Cisco
	NotImpl → Juniper or Linux	NotImpl → Hidden → Juniper or Linux
	Cisco → NotImpl	Cisco → Hidden → NotImpl
	Cisco → Cisco	Cisco → Hidden → Cisco
	Juniper or Linux → Juniper or Linux	Juniper or Linux → Hidden → Juniper or Linux
		Cisco → Juniper or Linux

Table 7.1: Possible router RR implementation transitions arranged by RR delta; deltas 3 and 4 are not shown. Juniper and Linux are written together to save space.

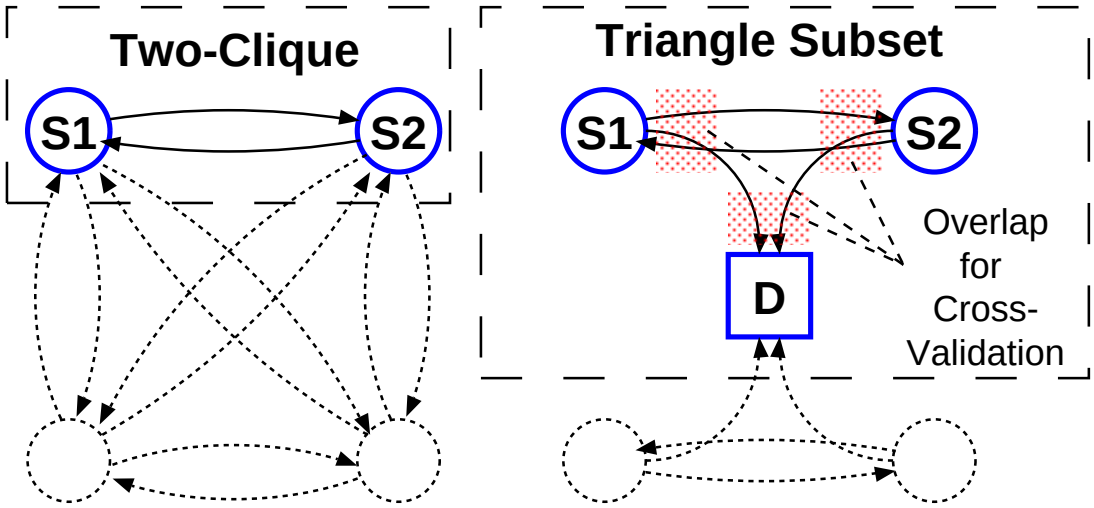


Figure 7.6: We first align addresses in two-cliques (left) between all sources and then subset triangles (right) to all destinations increasing overlap and decreasing errors.

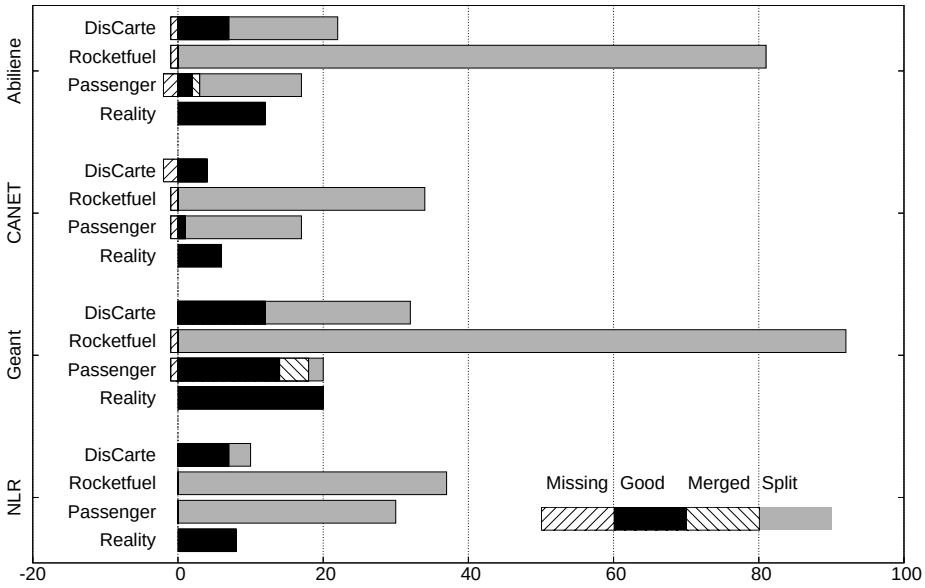
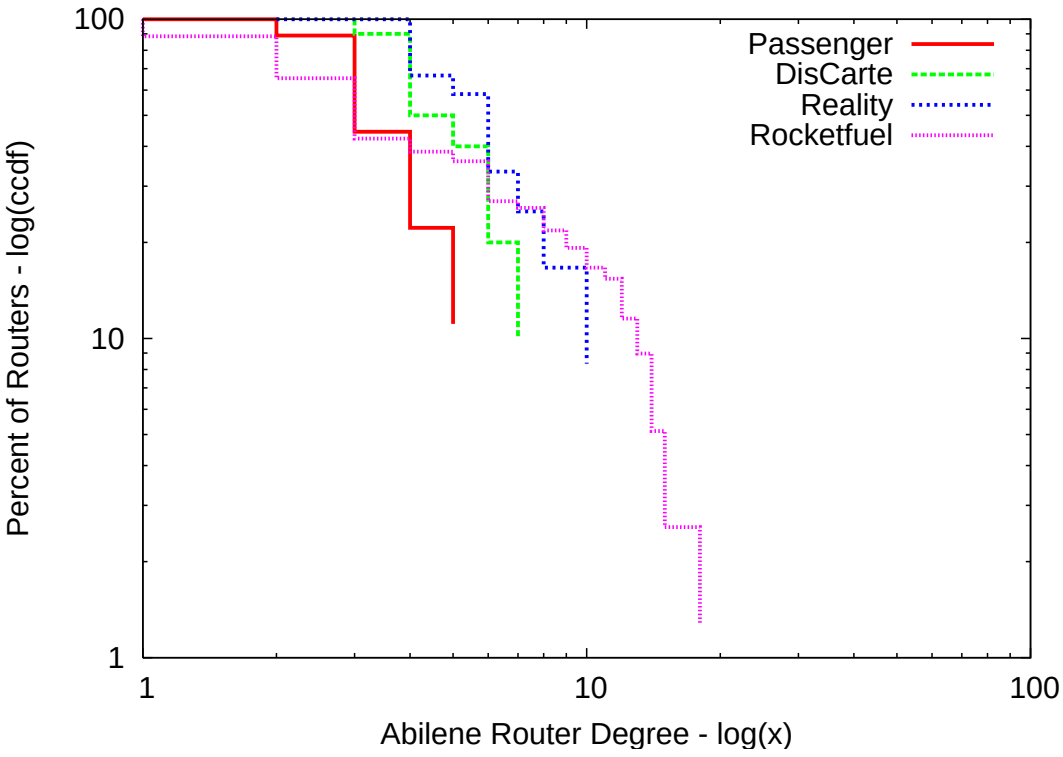
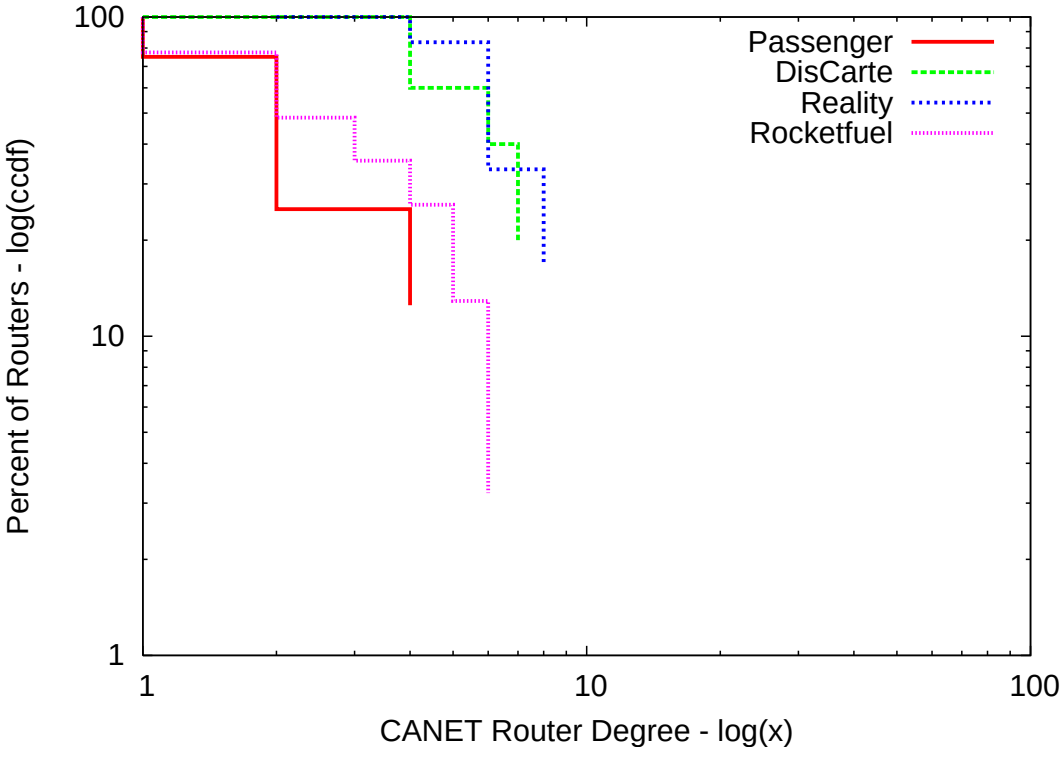
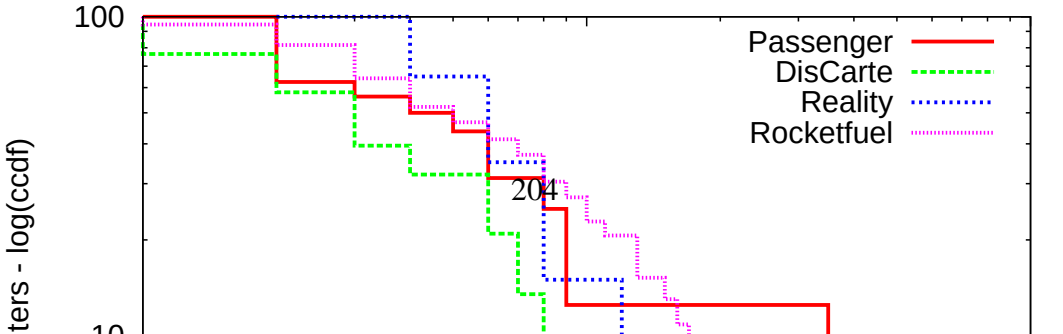


Figure 7.7: Number of discovered routers (partitioned by accuracy classification) compared to published topologies.


100
10
1
1
10
100
Percent of Routers - log(ccdf)
Abilene Router Degree - log(x)
Passenger
DisCarte
Reality
Rocketfuel


100
10
1
1
10
100
Percent of Routers - log(ccdf)
CANET Router Degree - log(x)
Passenger
DisCarte
Reality
Rocketfuel


100
Passenger
DisCarte
Reality
Rocketfuel

	Abilene	CANET4	Géant	NLR
Links: Found	21	11	45	21
Total	33	16	62	22
(%)	63%	69%	72%	95%
False Links	0	0	0	0

Table 7.2: Completeness of DisCarte-inferred links.

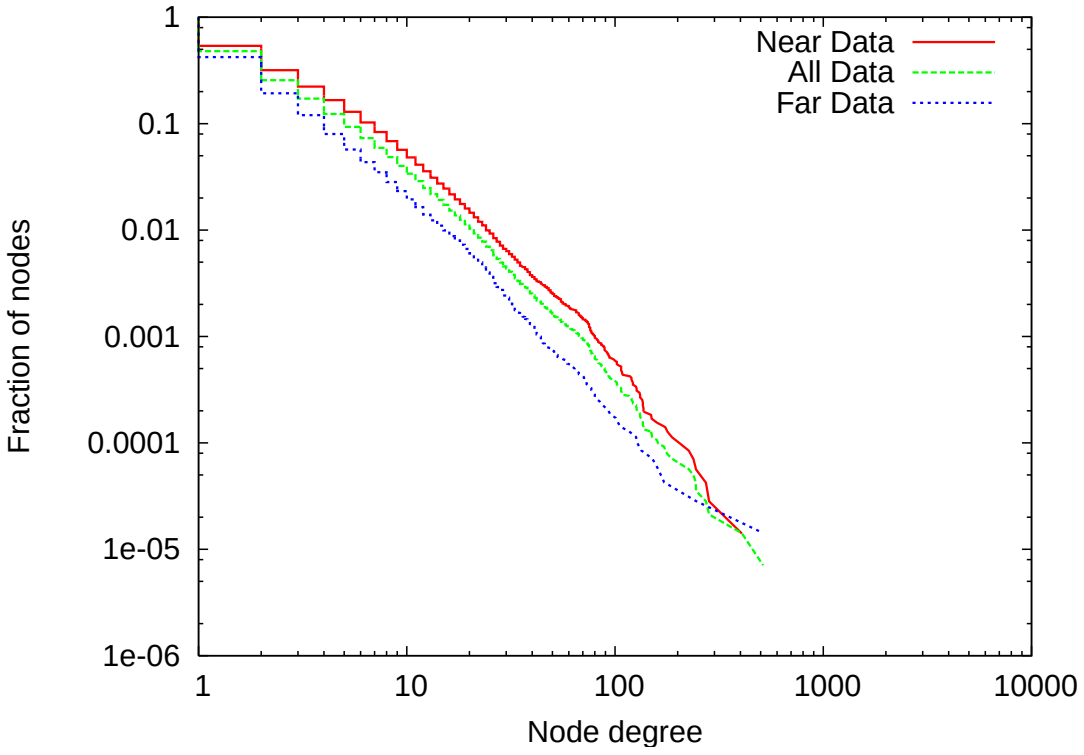


Figure 7.9: Bias in DisCarte-computed topology.

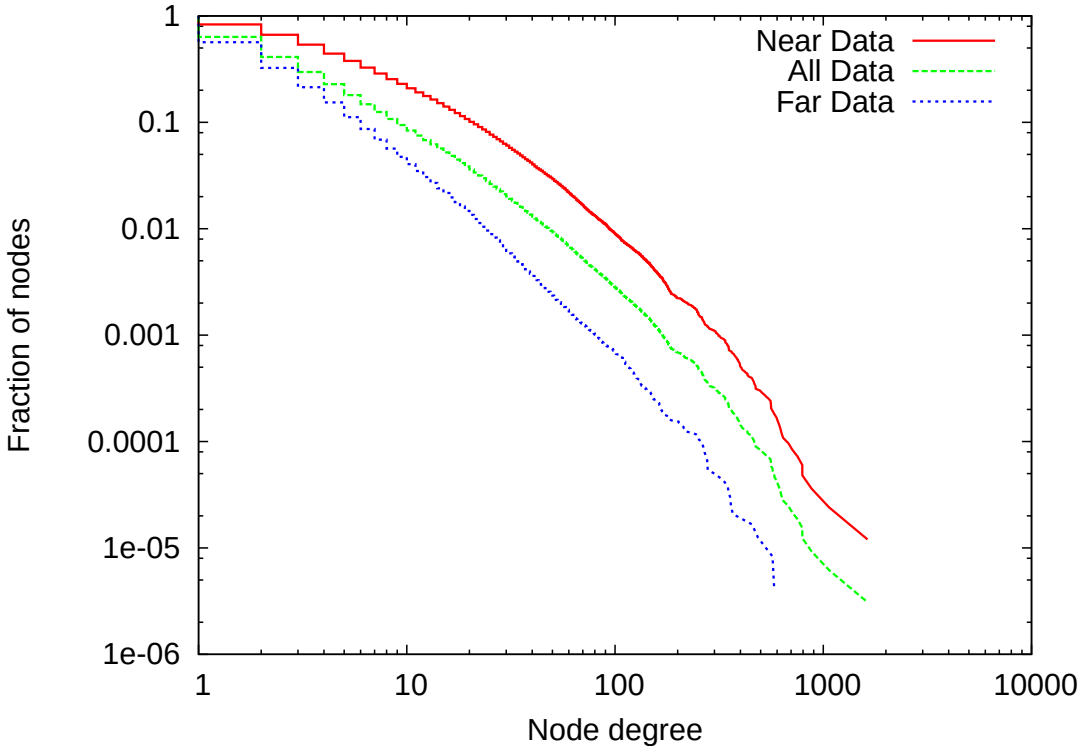


Figure 7.10: Bias in Rocketfuel-computer topology.

BIBLIOGRAPHY

[1] A. Abdul-Rahman and S. Hailes. Supporting Trust in Virtual Communities. In *Proceedings Hawaii International Conference on System Sciences 33*, 2000.

[2] K. Aberer and Z. Despotovic. Managing trust in a Peer-2-Peer information system. In H. Paques, L. Liu, and D. Grossman, editors, *Proceedings of the Tenth International Conference on Information and Knowledge Management (CIKM-01)*, pages 310–317, New York, Nov. 5–10 2001. ACM Press.

[3] Abilene. `http://abilene.internet2.edu`.

[4] Abilene router configurations. `http://http://pea.grnoc.iu.edu/Abilene`.

[5] E. Adar and B. Huberman. Free riding on gnutella. Technical report, Xerox PARC, August 2000.

[6] S. Ailleret. Larbin: Multi-purpose web crawler. `http://larbin.sourceforge.net/`.

[7] See `www.akamai.com`.

[8] `http://www.akamai.com/en/html/technology/overview.html`.

[9] A. Akella, S. Seshan, and A. Shaikh. An empirical evaluation of wide-area internet bottlenecks. In *Internet Measurement Conference*. ACM SIGCOMM/USENIX, 2003.

[10] T. H. P. R. Alliance. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`.

[11] M. Allman and V. Paxson. On Estimating End-to-End Network Path Properties. In *SIGCOMM*, pages 263–274, 1999.

[12] K. C. Almeroth, M. H. Ammar, and Z. Fei. Scalable delivery of web pages using cyclic best-effort multicast. In *Proceedings of INFOCOM*, pages 1214–1221, 1998.

[13] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, pages 131–145, Banff, Alberta, Canada, Oct. 2001.

[14] D. Applegate and E. Cohen. Making intra-domain routing robust to changing and uncertain traffic demands: Understanding fundamental tradeoffs. In *ACM SIGCOMM*, pages 313–324, Karlsruhe, Germany, Aug. 2003.

[15] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeir. Avoiding traceroute anomalies with Paris traceroute. In *IMC*, 2006.

[16] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz. TCP behavior of a busy Internet server: Analysis and improvements. In *INFOCOM*, San Francisco, CA, Mar. 1998.

[17] S. Banerjee, B. Bhattacharjee, and C. Kommreddy. Scalable Application Layer Multicast. In *Proceedings of ACM SIGCOMM*, 2002.

[18] S. Banerjee, B. Bhattacharjee, and S. Parthasarathy. A Protocol for Scalable Application Layer Multicast. CS-TR 4278, Department of Computer Science, University of Maryland, College Park, 2001.

[19] S. Banerjee, C. Kommareddy, K. Kar, B. Bhattacharjee, and S. Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *Proc. IEEE Infocom*, June 2003.

[20] S. Banerjee, S. Lee, B. Bhattacharjee, and A. Srinivasan. Resilient multicast using overlays. *ACM Sigmetrics*, June 2003.

[21] S. Bhattacharjee, K. Calvert, and E. Zegura. Self-organizing wide-area network caches. In *IEEE Infocom'98*, 1998.

[22] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[23] B. Bollobas. *Random Graphs*. Academic Press, 1985.

[24] R. P. Bonica and D.-H. Gan. ICMP extensions for multiprotocol label switching. Internet Draft (work in progress): draft-ietf-mpls-icmp-05, Mar. 2006.

[25] R. P. Bonica, D.-H. Gan, P. Nikander, D. C. Tappan, and C. Pignataro. Modifying ICMP to support multi-part messages. Internet Draft (work in progress): draft-bonica-internet-icmp-08, Aug. 2006.

[26] R. P. Bonica, D.-H. Gan, and D. C. Tappan. Icmp extensions for multiprotocol label switching. Internet Draft (work in progress): draft-ietf-mpls-icmp-05, Mar. 2006.

[27] B. Braden, D. Clark, and S. Shenker. Rfc1633: Integrated Services in the Internet Architecture: an Overview.

[28] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, Oct. 1995.

[29] See `www.bittorrent.com`.

[30] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In J. Wroclawski, editor, *Proceedings of the ACM SIGCOMM 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM-02)*, volume 32, 4 of *Computer Communication Review*, pages 47–60, New York, Aug. 19–23 2002. ACM Press.

[31] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 56–67. ACM Press, 1998.

[32] F. Calimeri, W. Faber, N. Leone, and G. Pfeifer. Pruning operators for disjunctive logic programming systems. *Fundamenta Informaticae*, 71(2-3):183–214, 2006.

[33] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. In *INFOCOM*, Kobe, Japan, Apr. 1997.

[34] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in communications (JSAC)*, 2002.

[35] W. chang Feng, D. Kandlur, D. Saha, and K. Shin. Stochastic Fair Blue: A Queue Management Algorithm for Enforcing Fairness. In *INFOCOM*, pages 1520–1529, Anchorage, AK, Apr. 2001.

[36] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *Proceedings of ACM SIGMETRICS*, June 2000.

[37] Personal e-mail from Cisco engineers.

[38] k. claffy, T. E. Monk, and D. McRobb. Internet tomography. *Nature, Web Matters*, Jan. 1999. `http://www.nature.com/nature/webmatters/tomog/tomog.html`.

[39] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *ACM SIGCOMM*, pages 106–114, Stanford, CA, Aug. 1988.

[40] `http://codeen.cs.princeton.edu/talks/iris_pl`.

[41] B. Cohen. Incentives build robustness in bittorrent. In *P2P Economics Workshop*, 2003.

[42] Computer Science and Telecommunications Board, National Research Council. *Looking Over the Fence at Networks: A Neighbor's View of Networking Research*. The National Academies Press, 2001.

[43] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *SOSP*, Bolton Landing, NY, Oct. 2003.

[44] S. Deering and D. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. In *ACM Transactions on Computer Systems*, May 1990.

[45] `http://www.dbai.tuwien.ac.at/proj/dlv/examples/3col`.

[46] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *ACM SIGMETRICS*, Banff, Canada, June 2005.

[47] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the Internet topology. In *ACM SIGCOMM*, pages 251–262, Cambridge, MA, Sept. 1999.

[48] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *Computer Communications Review (Proceedings of SIGOCOMM'98)*, 28(4):254–265, Sept. 1998.

[49] W.-C. Feng, D. Kandlur, D. Saha, and K. G. Shin. BLUE: An Alternative Approach to Active Queue Management. In *NOSSDAV '01: Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*, pages 41–50, New York, NY, USA, 2001. ACM.

[50] S. Floyd. TCP and explicit congestion notification. *ACM CCR*, 24(5):10–23, Oct. 1994.

[51] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[52] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[53] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. Reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of SIGCOMM*, Cambridge, Massachusetts, Sept. 1995.

[54] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (SACK) option for TCP. IETF RFC-2883, July 2000.

[55] R. Fonseca, G. M. Porter, R. H. Katz, S. Skenker, and I. Stoica. IP Options are not an option. Technical Report UCB/EECS-2005-24, EECS Department, University of California, Berkeley, Dec. 2005.

[56] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, San Francisco, CA, Mar. 2004.

[57] E. Friedman and P. Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10(2):173–199, 1998.

[58] L. Gao and F. Wang. The extent of AS path inflation by routing policies. In *IEEE Global Telecommunications Conference (GLOBECOM) Global Internet Symposium*, volume 3, pages 2180–2184, Taipei, Taiwan, Nov. 2002.

[59] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[60] Gnutella Home Page. See `http://gnutella.wego.com`.

[61] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *INFOCOM*, pages 1371–1380, Tel Aviv, Israel, Mar. 2000.

[62] Graphviz. `http://www.graphviz.org`.

[63] K. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *SOSP*, Bolton Landing, NY, Oct. 2003.

[64] M. Gunes and K. Sarac. Analytical IP alias resolution. In *IEEE Int'l Conference on Communication*, June 2006.

[65] M. H. Gunes and K. Sarac. Analytical IP alias resolution. In *IEEE International Conference on Communications (ICC)*, pages 11–15, 2006.

[66] N. Hu, L. E. Li, Z. M. Mao, P. Steenkiste, and J. Wang. Locating internet bottlenecks: algorithms, measurements, and implications. In *ACM SIGCOMM*, pages 41–54, Portland, OR, Aug. 2004.

[67] N. Hu, O. Spatscheck, J. Wang, and P. Steenkiste. Locating Internet bottlenecks: Algorithms, measurements, and implications. In *ACM SIGCOMM*, Portland, OR, Aug. 2004.

[68] M. Huang. VNET: PlanetLab virtualized network access. `http://www.planet-lab.org/doc/vnet.php`, May 2005.

[69] V. Jacobson. Pathchar. `ftp://ftp.ee.lbl.gov/pathchar/`.

[70] V. Jacobson. Traceroute. `ftp://ftp.ee.lbl.gov/traceroute.tar.Z`.

[71] V. Jacobson. Congestion Avoidance and Control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[72] V. Jacobson. Congestion Avoidance and Control. In *Proceedings, SIGCOMM '88 Workshop*, pages 314–329. ACM SIGCOMM, ACM Press, Aug. 1988. Stanford, CA.

[73] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. IETF RFC-1323, May 1992.

[74] R. Jain and K. K. Ramakrishnan. Congestion avoidance in computer networks with a connectionless network layer: Concepts,. *Proceedings of the Computer Networking Symposium; IEEE; Washington, DC*, pages 134–143, 1988.

[75] C. Jin, Q. Chen, and S. Jamin. Inet: Internet topology generator. Technical Report CSE-TR-433-00, University of Michigan, EECS dept., 2000. `http://topology.eecs.umich.edu/inet/inet-2.0.pdf`.

[76] D. Kaminsky. Paratrace. `http://www.doxpara.com/read.php/docs/paratrace.html`, Nov 2002.

[77] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proceedings of the Twelfth International World Wide Web Conference (WWW)*, 2003.

[78] E. Katz-Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe. Towards ip geolocation using delay and topology measurements. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 71–84. ACM, 2006.

[79] See `www.kazaa.com`.

[80] K. Keys. iffinder. `http://www.caida.org/tools/measurement/iffinder/`, Mar. 2006.

[81] A. Kuzmanovic and E. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks. In *ACM SIGCOMM*, pages 75–86, Portland, OR, Aug. 2004.

[82] K. Lai and M. Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of SIGCOMM*, pages 283–294, 2000.

[83] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USITS*, pages 123–134, San Francisco, CA, Mar. 2001.

[84] A. Lakhina, J. Byers, M. Crovella, and P. Xie. Sampling biases in IP topology measurements. In *INFOCOM*, pages 332–341, San Francisco, CA, Apr. 2003.

[85] N. Leibowitz, M. Ripeaunu, and A. Wierzbicki. Deconstructing the kazaa network. In *3rd IEEE Workshop on Internet Application*. IEEE, 2003.

[86] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Computational Logic*, 7(3):499–562, 2006.

[87] J. Liang, R. Kumar, and K. W. Ross. The KaZaA Overlay: A Measurement Study. `http://cis.poly.edu/~ross/papers/KazaaOverlay.pdf`.

[88] `http://www.tcpdump.org`.

[89] M. Lichtenberg and J. Curless. DECnet Transport Architecture. *Digital Technical Journal*, 4(1), 1992.

[90] `http://www.limewire.com/english/content/netsize.shtml`.

[91] M. Litzkow, M. Livny, and M. Mutka. Condor: A hunter of idle workstations. In *ICDCS*, 1988.

[92] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*, Seattle, WA, Nov. 2006.

[93] P. Mahadevan, D. Kriokov, K. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. In *SIGCOMM'06*, 2006.

[94] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *IMW*, pages 231–236, Marseille, France, Nov. 2002.

[95] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *SOSP*, pages 106–119, Bolton Landing, NY, Oct. 2003.

[96] Z. M. Mao, J. Rexford, J. Wang, and R. Katz. Towards an accurate AS-level traceroute tool. In *ACM SIGCOMM*, pages 365–378, Karlsruhe, Germany, Aug. 2003.

[97] S. Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Sterling, 1994.

[98] M. Mathis, J. Mahdavi, S. Floyd, and A. Ramanov. RFC2018: TCP Selective Acknowledgment Options, October 1996.

[99] A. Medina, M. Allman, and S. Floyd. Measuring the evolution of transport protocols in the Internet. *ACM CCR*, Apr. 2005.

[100] A. Medina, I. Matta, and J. Byers. BRITE: A flexible generator of Internet toplogies. Technical Report BU-CS-TR-2000-005, Boston University, 2000.

[101] D. Meyer. University of Oregon Route Views project. `http://www.routeviews.org/`.

[102] S. B. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay measurements. In *INFOCOM*, New York, NY, Mar. 1999.

[103] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. See `http://www.caida.org/outreach/papers/2003/sapphire2/`.

[104] R. Morselli, B. Bhattacharjee, A. Srinivasan, and M. A. Marsh. Efficient lookup on unstructured topologies. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 77–86, New York, NY, USA, 2005.

[105] A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *ACM SIGCOMM*, pages 11–18, Karlsruhe, Germany, Aug. 2003.

[106] Nanog email: Dos? `http://www.merit.edu/mail.archives/nanog/2003-01/msg00594.html`.

[107] V. N. Padmanabhan, L. Qiu, and H. J. Wang. Passive network tomography using bayesian inference. In *IMW*, pages 93–94, Marseille, France, Nov. 2002.

[108] V. N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *IPTPS*, 2002.

[109] J.-J. Pansiot and D. Grad. On routes and multicast trees in the Internet. *ACM CCR*, 28(1):41–50, Jan. 1998.

[110] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997.

[111] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31:2435–2463, December 1999.

[112] V. Paxson. An Analysis of Using Reflectors for Distributed Denial-of-service attacks. *ACM Computer Communications Review (CCR)*, 31(3), July 2001.

[113] V. Paxson. Strategies for sound Internet measurement. In *IMC*, pages 263–271, Taormina, Sicily, Italy, Oct. 2004.

[114] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *HotNets*, pages 59–64, Princeton, NJ, Oct. 2002.

[115] `http://pdos.csail.mit.edu/~strib/pl_app`.

[116] J. Postel, editor. Internet protocol specification. IETF RFC-791, Sept. 1981.

[117] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *In Proceedings of the ACM SIGCOMM 2001 Technical Conference*, 2001.

[118] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proceedings of 3rd International Workshop on Networked Group Communications*, Nov. 2001.

[119] D. Reed. "Small TCP Packets == very large overhead == DoS?", July 2001. See `http://www.securityfocus.com/archive/1/195457`.

[120] X. Rex Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient multicast support for continuous-media applications. In *Proceedings of NOSSDAV*, St. Louis, Missouri, May 1997.

[121] Transmission control protocol specification. IETF RFC-793, 1981. ARPA Working Group Requests for Comment DDN Network Information Center, SRI International, Menlo Park, CA.

[122] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. In *Proceedings of ACM SIGCOMM 2005*, August 2005.

[123] F. Ricca, W. Faber, and N. Leone. A backjumping technique for disjunctive logic programming. *The European Journal on Artificial Intelligence*, 19(2):155–172, 2006.

[124] E. C. Rosen, A. Viswanathan, and R. Callon. Multiprotocol label switching architecture. IETF RFC-3031, Jan. 2001.

[125] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[126] S. Savage. Sting: a TCP-based network measurement tool. In *USITS*, pages 71–79, Boulder, CO, Oct. 1999.

[127] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP Congestion Control with a Misbehaving Receiver. *Computer Communication Review*, 29(5), 1999.

[128] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The end-to-end effects of Internet path selection. In *ACM SIGCOMM*, pages 289–299, Cambridge, MA, Sept. 1999.

[129] N. Shankar, C. Komareddy, and B. Bhattacharjee. Finding Close Friends over the Internet. In *Proceedings of International Conference on Network Protocols*, Nov. 2001.

[130] R. Sherwood and N. Spring. A Platform for Unobtrusive Measurements on PlanetLab. In *USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Nov. 2006.

[131] R. Sherwood and N. Spring. A platform for unobtrusive measurement on PlanetLab. In *USENIX Workshop on Real, Large Distributed Systems (WORLDS)*, Seattle, WA, Nov. 2006.

[132] R. Sherwood and N. Spring. Touring the Internet in a TCP Sidecar. In *ACM Internet Measurement Conference (IMC)*. ACM Press, 2006.

[133] R. Sherwood and N. Spring. Touring the Internet in a TCP sidecar. In *IMC*, pages 339–344, Rio de Janeiro, Brazil, Oct. 2006.

[134] http://slurpie.sourceforge.net.

[135] Smurf Attack: See `http://www.cert.org/advisories/CA-1998-01.html`.

[136] N. Spring, M. Dotcheva, M. Rodrig, and D. Wetherall. How to resolve IP aliases. Technical Report 04–05–04, University of Washington Dept. CSE, May 2004.

[137] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *ACM SIGCOMM*, pages 113–124, Karlsruhe, Germany, Aug. 2003.

[138] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM*, pages 133–146, Pittsburgh, PA, Aug. 2002.

[139] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1):2–16, Feb. 2004.

[140] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for network research: Myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, Jan. 2006.

[141] N. Spring, D. Wetherall, and T. Anderson. Reverse-engineering the Internet. In *HotNets*, pages 3–8, Cambridge, MA, Nov. 2003.

[142] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *USITS*, pages 225–238, Seattle, WA, Mar. 2003.

[143] Squid Web proxy cache. `http://www.squid-cache.org/`.

[144] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *USENIX Annual Technical Conference*, Monterey, CA, June 2002.

[145] W. R. Stevens. RFC2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997.

[146] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[147] J. Strauss, D. Kitabi, and F. Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *IMC*, 2003.

[148] H. Tangmunarunkit, R. Govindan, and S. Shenker. Internet path inflation due to policy routing. In *SPIE ITCOM Workshop on Scalability and Traffic Control in IP Networks*, volume 4526, pages 188–195, Denver, CO, Aug. 2001.

[149] R. Teixeira, K. Marzullo, S. Savage, and G. Voelker. In search of path diversity in ISP networks. In *IMC*, pages 313–318, Miami, FL, Oct. 2003.

[150] `http://www.cisco.com/web/strategy/government/wsca/states/price_list.html`.

[151] `http://www.ebay.com`.

[152] L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[153] X. Wang and M. K. Reiter. Mitigating bandwidth-exhaustion attacks using congestion puzzles. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 257–267, New York, NY, USA, 2004. ACM Press.

[154] P. Watson. Slipping In The Window: TCP Reset Attacks. See `http://www.osvdb.org/reference/SlippingInTheWindow_v1.0.doc`.

[155] See `http://www.gnu.org/software/wget/wget.html`.

[156] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[157] B. Yao, R. Viswanathan, F. Chang, and D. Waddington. Topology inference in the presence of anonymous routers. In *INFOCOM*, pages 353–363, San Francisco, CA, Apr. 2003.

[158] B. Yu and M. P. Singh. A social mechanism of reputation management in electronic communities. In *Proceedings of Fourth International Workshop on Cooperative Information Agents*, 2000.

[159] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM*, San Francisco, CA, Sept. 1996.

[160] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of Internet flow rates. In *ACM SIGCOMM*, Pittsburgh, PA, Aug. 2002.

[161] Y. Zhang, M. Roughan, C. Lund, and D. Donoho. An information-theoretic approach to traffic matrix estimation. In *ACM SIGCOMM*, pages 301–312, Karlsruhe, Germany, Aug. 2003.

[162] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area dat a dissemination. In *Eleventh International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2001)*, 2001.

[163] P. Zimmermann. Pretty good privacy user's guide. Distributed with PGP software, June 1993.