Exploiting Host Availability in Distributed Systems

by

James W. Mickens

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2008

Doctoral Committee:

Associate Professor Brian D. Noble, Chair
Professor Farnam Jahanian
Professor David L. Neuhoff
Assistant Professor Zhuoqing Mao

ACKNOWLEDGEMENTS

First and foremost, I thank my parents for the support and the encouragement that they provided to me during my time in graduate school.

I thank my advisor, Brian Noble, for having the courage to strive for excellence and accept me as his student. He is a good man and I shall remember our campaign fondly.

I also thank the adminstrative staff who helped me to file travel reports and use inscrutable fax machines. In particular, I want to thank Steve Reger, Bert Wachsman, Kirsten Knecht, and Amanda Brown.

I thank the staff of Espresso Royale Coffee for the free iced tea. Although I've cost your company thousands of dollars in lost revenue, I've repaid it in the form of free lectures on science, philosophy, and the existential importance of Led Zeppelin. Take my wisdom and make it your own.

I also thank the wonderful officemates that I have had over the years. Few bonds are as precious as those forged at 2AM when nobody's paper is finished and the submission deadline is an hour away. Let it be known that I beat impossible odds with the help of these people: Landon Cox, Mark Corner, Richard Hankins, Andrew Nierman, the coughing guy with the huge stack of papers on his desk who failed his quals twice despite having two (2) hardcopies of every computer science paper ever written, Anthony Nicholson, Sam Shah, *the* Xu Chen (aka Simon Chen), Arsalan Tavoliki, Geoffrey Werner-Allen, Karl Chen, Aruna Balasubramanian, Jack Wang,

S.K. Yorenzima, Gong Chen, and Ramya Raghavendra.

Finally, I would like to thank myself, since without me, there would be no James W. Mickens to write this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

CHAPTER I

Introduction

In a distributed system, several networked machines provide a highly available service to remote clients. Well-known distributed systems include network file systems like AFS [52] and NFS [93], and authentication systems like Kerberos [79] and the Lightweight Directory Access Protocol (LDAP) [54]. These systems strive to present a consistent service abstraction to clients scattered across a wide area network.

To ensure the availability and reliability of the service, traditional distributed systems make a clear distinction between clients and servers. Client machines may be poorly administered, cheaply constructed, often offline, and possibly malicious. In contrast, servers are expected to be well-administered and almost always online; they are also expected to have high quality hardware and to respect the distributed protocol. If these lofty expectations for server behavior are satisfied, a well-behaved client will almost always receive a consistent view of the distributed service, despite the presence of other faulty, mischievous, or intermittently online clients.

In such an operating environment, the availability of the distributed service is primarily determined by server-side factors. Such factors include planned downtime, administrative misconfiguration, and the random failure of server hardware. Coping with planned downtime is relatively straightforward; server maintenance can be performed when client demand is known to be low, backup servers can handle load while the original ones are down, and users can be warned of service outages ahead of time. Although misconfiguration is difficult to prevent, server administrators are expected to be better trained than end users—thus, server misconfiguration is expected to be relatively rare. Server hardware failure can be mitigated using well-known techniques like node failover and hardware replication [86]. Thus, in a traditional distributed system, the availability of the servers is extremely high. Vendors often boast of 99.999% machine availability, or less than five minutes of downtime a year [58, 32]. Such statistics are used to bolster a key sales pitch: without highly available server

machines, it is impossible to build robust distributed applications.

1.1 An Increasingly Decentralized World

In recent years, the venerable client/server paradigm has been challenged by the cooperative or "peer-to-peer" framework. Cooperative systems are not composed of expensive custom-built servers, but large numbers of commodity PCs. Each machine acts as both a client and a server, and idle resources that would otherwise be wasted are allocated to the cooperative service. Since most PCs have non-trivial resource surpluses, most machines have substantial opportunities for contribution to the aggregated service abstraction. For example, in a study of 4,801 Windows PCs, Douceur found that the average file system was only 53% full [38]—thus, the typical machine had gigabytes of surplus storage that could be "leased" to a distributed data store. By leveraging such idle resources, peer-to-peer techniques can spread storage, computation, and network transmission burdens across many nodes, providing a foundation for scalable decentralized systems. Examples of these systems include PAST [92], an overlay-based storage system, Pastiche [35], a cooperative backup framework, and Skype [50], a platform for Internet telephony.

The inherent scalability of cooperative systems is very attractive. However, peer-to-peer frameworks invalidate many of the simplifying assumptions from traditional distributed systems design. In particular, servers may be untrusted, independently (and poorly) administered, and constructed from low-quality hardware. Even if all peer-to-peer nodes are trusted, well administered, and have reliable hardware, a larger and inescapable problem still looms: machine uptime patterns are no longer well-behaved. The problem is unavoidable, since peer-to-peer services run at the behest of individual, non-coordinated users who may turn their machines on and off at any moment. Even if their machines are online, users may opt in or out of the cooperative service at any time. The fundamental capriciousness of uptime in peer-to-peer systems makes it very difficult to predict the global performance of the distributed service.

An abundance of empirical evidence suggests that machine availability must be a primary concern in cooperative systems. For example, Bolosky *et al* studied the feasibility of running a serverless file system across the Microsoft corporate network. They found that the primary cause of downtime was users turning off their machines [21]. However, this whim-driven behavior is only partially explained by diurnal workday patterns—a wide variety of uptime behaviors defy such a simple description. Studies of deployed peer-to-peer systems show that application-level availability is very

low and even more capricious [95]. For example, in the Overnet distributed hash table, half of all machines have availabilities below 30% [17]. Such figures stand in stark contrast to traditional server uptimes, for which 99% availability is considered completely unacceptable.

Mobile devices like laptops and PDAs represent yet another platform for new distributed applications. However, these devices also have very intermittent network connectivity. **The key observation is this: when individual node availability is so low and unpredictable, hardware failures and planned downtime no longer drive aggregate metrics for application performance. Instead, the primary factor becomes voluntary entry into and withdrawal from the distributed service.**

1.2 Security and Network Reachability

Availability is also important from the perspective of security. In a distributed system, network reachability is a key component of availability—a host that is powered up but disconnected from its peers cannot provide services to those peers, and it cannot use services delivered by others. Thus, disrupting network access at the IP or overlay level is an extremely pernicious attack. Disruptions may also arise from inadvertent routing misconfigurations or power outages. If distributed systems could explicitly reason about their availability status, it would be easier for them to detect network anomalies in a timely fashion.

Network availability also has a profound impact on the dissemination of malware. Offline hosts are obviously safe from network-based infection vectors. However, scanning worms like Slammer [76] usually contact random IP addresses without regard to whether the associated hosts are online. If malware writers had a better understanding of availability patterns, they could design worms with smarter scanning patterns, or launch random scanning worms at a time when global availability was predicted to be high.

1.3 Availability as a First-class Concern

I argue that accurate models of machine availability are crucial for understanding how modern distributed systems behave. As described in Section VI, previous availability models [16, 20, 40] make conservative assumptions to provide guarantees against worst-case scenarios. However, these pessimistic models provide limited insight into the complex, time-dependent behaviors that are observed in the wild.

For example, in real distributed systems, global availability ebbs and flows with non-trivial deviations from the mean, the minimum, and the maximum [17, 21, 50]. Examining the uptime behavior of individual hosts reveals a wide variety of patterns, but this complexity is imprecisely captured through aggregate-level statistics.

If we want to understand these networks at a deep level, we must devise new availability models. These new models must track per-machine state, explain emergent system-level trends in terms of individual host behavior, and predict future availability, both at the granularity of single machines and clusters of machines. Armed with a fuller understanding of availability, system architects can view fluctuating availability as a manageable (and perhaps exploitable) phenomenon, as opposed to a burden to be coped with via overly pessimistic designs.

1.4 Thesis Statement

It is my thesis that:

> **Using new, more accurate models of machine availability, we can decrease network and storage utilization, improve system security, and generally achieve a better understanding of distributed system dynamics.**

1.5 Overview of the Dissertation

Broadly speaking, the first part of the dissertation provides new analytical techniques for characterizing and predicting availability, describing how these techniques can be applied to improve performance and reliability. The second part of the dissertation focuses on security, showing how availability-introspective systems can detect anomalous (and potentially malicious) behavior. The final part of the dissertation discusses related work and explains how previous research has treated fluctuating host availability as a nuisance instead of an opportunity. Taken as a whole, the dissertation strives to show that availability is a first-class concern in distributed system design.

1.5.1 Exploiting Availability for Performance and Reliability

Chapters II and III represent the first part of the manuscript. Chapter II provides the bulk of the analytical modeling, describing how to encode availability as a binary string or "signal." Given such a representation, we can leverage techniques from signal processing and information theory to analyze the patterns that are contained

within. For example, by examining the Fourier transforms of availability signals, we can create automatic classifiers for various types of host behavior. These basic kinds of behavior form a taxonomy of availability classes, and this taxonomy can be used to succinctly summarize the behavioral differences between two networks.

Chapter II also explains how to predict future availability using techniques like Markov modeling. Chapter III then describes how distributed systems can use these forecasts to improve performance or robustness. For example, the forecasts are useful for cooperative storage systems which seek to minimize the amount of data replicated on lowly available hosts.

Importantly, some hosts may have erratic behavior that is difficult to predict. This is especially likely when availability is defined at the application layer instead of the network layer, and the application allows users to opt in or out as they please. In these scenarios, a host is only available when it is online and its user feels like running a particular program. Availability prediction will obviously be more difficult in these environments. Nevertheless, I show that in most networks, there are a non-trivial number of hosts with predictable behavior, and their regularity can be exploited.

1.5.2 Detecting Network Anomalies with Availability Introspection

By definition, a distributed system requires a network to route messages between hosts. The most common routing framework is the Internet Protocol (IP) [57], which forwards messages across a vast number of routers and physical links scattered across the globe. Applications may find it useful to construct overlay networks [101, 91, 5] atop this physical infrastructure. Overlays act as virtual forwarding systems, allowing applications to define arbitrary routing addresses. They also allow applications to route messages around failures in the physical routing core, since these failures may not affect all of the overlay routes which connect a sender and a recipient.

IP routing and overlay routing are vulnerable to a variety of attacks. For example, IP routers update their forwarding tables using a distributed protocol called BGP [49]. BGP allows each router to announce the address blocks that it handles directly. Routers also use BGP to announce their best paths to all other address blocks. When a router receives a BGP update from a peer, it may start to forward packets through that peer if the peer's path to a destination is shorter than the locally known one. In the absence of malicious parties, epidemic exchanges of routing state help to keep routes short. Unfortunately, BGP messages are not authenticated. Thus, a malicious party can advertise an attractive route to an address block that it does not own. Well-intentioned routers will forward traffic to the attacker-controlled

machine, and the attacker can then drop, inspect, or modify the packets. This type of disruption is called an IP hijacking attack [115].

Overlays also maintain global routing state through local exchanges of forwarding tables. Some overlays authenticate these exchanges, preventing arbitrary parties from influencing the routing process. However, known parties can still be malicious or faulty, and diagnosing routing problems is still difficult. Suppose that overlay peer P_1 wants to send a message to P_5, and that this message will be routed through intermediate peers P_2, P_3, and P_4. If P_5 never receives P_1's message, there may be a failure in the overlay *or* the underlay. In other words, an application-level forwarder may have dropped the message, or the message may have been dropped by a bad IP router or physical link. Even if we "magically" know that the underlay is not to blame, P_1 cannot just blame P_2, since P_3 or P_4 could also be responsible for the drop. Even worse, P_1 cannot even know that P_3 and P_4 follow P_2 in the route, since P_1's routing table only stores immediate next hop information.

In the second part of the dissertation, I describe how availability introspection can detect routing anomalies, both at the IP level and the overlay level. Chapter IV uses *subnet availability fingerprints* to detect IP hijacking. Given a subnet containing N hosts, its fingerprint at time t is simply an N-bit vector representing the availability of each host at t. Fingerprints can be compared for similarity using simple metrics like Hamming distance or bitwise correlation measures. Empirical observation shows that during normal network operation, a subnet's fingerprint is stable over several minutes and globally unique. Thus, an active probing system that constructs fingerprints every ρ seconds can detect routing anomalies with a duration of at least ρ. Chapter IV describes a prototype implementation of such an active probing system and demonstrates that it can detect IP hijacking quickly and accurately.

Chapter V explains how to detect faulty overlay forwarders using availability probing. A misbehaving forwarder is defined as a host that drops messages despite the presence of functional IP paths to its routing peers. To determine the quality of these paths, peers use network tomography [5, 8, 29, 41]. High quality tomography is bandwidth-intensive, but peers only engage in heavyweight path measurements when a routing fault is suspected. Peers publish their tomographic data and their routing tables in a globally-accessible distributed database. When a message sender fails to receive an acknowledgment from the message recipient, the sender uses the database to determine the overlay-level path the message took, as well as the network conditions along the underlying IP paths. If none of the physical links were down, the appropriate overlay peer is blamed for the drop. The overlay can then sanction

the peer in some way, e.g., by adding it to a blacklist of faulty routers.

Suppose that an overlay message is dropped and all of the underlying IP links are good. How do we determine *which* overlay peer is to blame? To answer this question, we use *forwarding commitments* and *recursive fault accusations*. Consider the overlay-level route $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$. P_2 gives P_1 a signed forwarding commitment indicating that it will route the message towards its ultimate destination. Similarly, P_3 will give a commitment to P_2. Now suppose that P_3 maliciously drops the message. P_1 and P_2 will have tomographic data showing that their IP paths are good; thus, each one will blame its downstream peer for the drop. Malicious P_3 can try to avoid blame in two ways. First, it can claim that its path to P_4 was down. This claim will be rebutted by tomographic measurements. Second, P_3 can try to blame P_4 for the drop. However, nobody will believe this claim since P_3, having never forwarded the message, will lack a signed forwarding claim from P_4. Thus, the accusation chain will stop at P_4, and once all of the accusations have been pushed upstream, P_4 will ultimately receive the blame for the message drop.

There are other subtle issues involved in this detection system. For example, malicious peers can submit bad tomographic data in an effort to punish good nodes or absolve bad ones. Peers can also advertise incorrect local routing state in an attempt to funnel traffic through particular nodes. Chapter V discusses way to detect these problems and other types of fraud.

CHAPTER II

Availability Modeling

The original motivation for this dissertation was a simple observation: large-scale distributed services are increasingly implemented atop commodity machines belonging to regular users. Content streaming [83], Internet telephony [50], scientific computing [6], and wide-area storage systems [92, 33, 18] are only a few of the collaborative efforts to which users can volunteer their PCs and laptops. Users are motivated to contribute for three reasons. First, they get something in return, whether it be content, access to a service, or the personal satisfaction of helping to solve a problem. Second, cooperative applications typically run in the background using spare resources, or run in the foreground but only at the behest of the user. Thus, users perceive little penalty for joining the collective. Third, and most relevant to this dissertation, users can opt in and out of the collective as they please. This also lowers the perceived overhead to collaboration. However, constant host turnover makes it difficult to build reliable distributed applications, since at any given time, important service state may reside on disconnected nodes.

From a design perspective, the standard way to cope with churn is to make pessimistic assumptions about the churn rate and then ensure that the system will work if the worst case is the persistent case. For example, consider a cooperative storage system in which each file is replicated on N machines. At any moment, some of the replica sites may be offline; in the worst case, all of the sites may be disconnected at the same time. There are two basic approaches to avoid such a scenario. Whenever peer p_1 goes down, the system can immediately regenerate its data on some other p_2 [101, 91]. This solution essentially posits that the churn rate can be arbitrarily capricious, so every node departure should trigger data regeneration to ensure redundancy targets. An alternate solution is to replicate each object so many times that it is extremely unlikely for all of the replica sites to be offline at once. For example, the TotalRecall system [18] measures global network availability during the night, which

is presumably the time when the largest number of hosts will be off. TotalRecall then "over-replicates" data such that even if this availability level were persistent, redundancy targets would still be met.

Both solutions ensure that redundancy targets are satisfied, but both require extra resources—extra bandwidth for data regeneration in the first case, and extra storage space to hold additional replicas in the second case. However, if the system designer could make less pessimistic assumptions about availability fluctuations, she could reduce some of the resource consumption associated with the "worst case is the common case" paradigm. For example, suppose that the system designer could predict the future availability of a host. In a regenerate-on-departure system, data could then be recreated on a host guaranteed to be online for a long time. This would reduce departure-induced copying, since data would migrate towards hosts that depart less often. Alternatively, in a system like TotalRecall, each object could be replicated across hosts whose future downtime is unlikely to be synchronized. This reduces the amount of over-replication needed to ensure the base redundancy target.

The goal of this chapter is to provide concrete techniques for availability prediction. However, before doing so, we provide a characterization of availability in several networks, providing a taxonomy for common types of availability behavior. By demonstrating that many hosts have identifiable uptime patterns, we accomplish two tasks. First, we learn about the low-level availability dynamics which drive aggregate network availability. Second, we identify sources of regularity that can be exploited for the purpose of prediction.

After establishing the existence of uptime patterns, we introduce techniques for availability prediction and evaluate them using empirically gathered uptime traces. Later, in Chapter III, we describe four different ways that systems can employ these techniques to improve performance and robustness.

2.1 A Note on Long-term Population Dynamics

In this work, we typically study a host's availability between the bookends of its initial entry into the system and its final exit from the system. In most cases, we do not explicitly model the rates of these bookending events. Long-term admission and drop-out phenomena are major issues in collaborative environments [17], but these dynamics are rich enough to merit an independent investigation. When appropriate, we discuss how global entry and exit rates affect availability characterization.

2.2 Do Availability Patterns Exist?

Regularity is the sine qua non of forecasting—a process is predictable only if its behavior contains patterns. Thus, before we attempt to predict host availability, we should establish whether it contains patterns. To do so, we will treat each host's uptime history as binary string where bit b_t is 1 if the host was online at time t, and 0 if it was offline. Given this representation, we can detect regularity using techniques from digital signal processing and information theory. Throughout this section, we will often refer to a host's uptime history as its availability signal.

2.2.1 An Availability Taxonomy

To determine whether uptime patterns exist, we examine two availability traces from two different networks. The first trace followed 51,662 PCs in the Microsoft corporate network [21], and the second captured the behavior of 321 hosts in the PlanetLab distributed testbed [12]. Each machine in the Microsoft trace was pinged hourly. In the PlanetLab traces [102], machines were pinged every 15 minutes, but we sampled every fourth measurement to provide a fair comparison with the Microsoft data. The lifetimes of PlanetLab nodes were long enough that such sampling did not distort the underlying availability patterns. Our PlanetLab data spanned the five week period from July 1 to August 4, 2004. The Microsoft data spanned the five week period from July 6 to August 9, 1999.

In Douceur's study of availability [37], he identified diurnal behavior by applying a Fourier transform to an availability signal and looking for spikes in the daily and weekly frequencies. We can generalize this technique to detect multiple types of uptime classes. We extend the method to include multiple types of spectrum tests. Once an availability signal passes a test, we provide it a label as explained below, and we do not apply any more tests.

For our first test, we classify a host as *always on* or *always off* if its availability signal contains 90% ones or zeros, respectively. Second, machines are subjected to Douceur's test to finding diurnal periodicity. Hosts that pass this test are *work-week periodic*. Third, if the Fourier decomposition resembles the curve $1/f$, the host's uptime pattern is the summation of low frequency sine waves. This means that the machine's online and offline stretches are long running, and we label these nodes as *long stretch*.

A host failing all of these tests is labeled *unstable*. Although it is possible for these hosts contain spectra regularity not covered by the previous tests, we have

Host Type	Microsoft	PlanetLab
Always On	60.66%	15.58%
Always Off	1.22%	5.60%
WW Periodic	9.79%	0.00%
Long Stretch	20.48%	67.60%
Unstable 70-90	2.05%	1.25%
Unstable 50-70	1.67%	1.56%
Unstable 10-50	4.12%	8.41%

Figure 2.1: Uptime Class Categorization

not observed such regularity in empirical availability traces. Indeed, machines which fail all of the previous tests typically have availability spectra that resembles noise, providing little hope of finding exploitable medium-to-long term patterns. However, these hosts might be predictable in the sense that we can always forecast online (offline) for majority online (offline) machines. Thus, we further bin these hosts according to the percentage of time that they are online. This creates the uptime classes *unstable70to90*, *unstable50to70*, and *unstable10to50*.

For availability prediction to be generally useful, most hosts should not have unstable uptime. Always-on and always-off hosts, while uninteresting from the analytical perspective, should be trivially predictable across all lookahead periods. Long stretch hosts lack periodic behavior and thus should be difficult to predict in over large lookaheads; however, they should be very predictable in the short-to-medium term. By definition, diurnal hosts possess rigorously cyclical behavior, so they should be easy to forecast across arbitrary time windows.

2.2.2 Empirical Results

Figure 2.1 describes the constituent uptime classes in the PlanetLab and Microsoft traces. In both networks, unstable hosts comprise a minority, representing 7.84% of Microsoft hosts and 11.2% of PlanetLab machines. Interestingly, over half of the Microsoft hosts are always online, whereas two thirds of the PlanetLab hosts are long stretch. Furthermore, about 10% of Microsoft hosts have diurnal behavior [1], but there are no such machines in the PlanetLab system.

These differences in distribution arise from the unique nature of each network. PlanetLab is a testbed for distributed experiments [12], and it is primarily comprised of machines donated by universities. Institutions must contribute hosts to use the

[1]Note that Douceur reported that 14% of Microsoft hosts have cyclical availability patterns [37], whereas we say that only 9.79% of them are work-week periodic. We report a lower percentage because we use a higher energy cutoff in the daily and weekly spectra for a machine to classify as work-week periodic. From the perspective of evaluating our predictors, this more stringent cutoff is reasonable. For example, a host which is always offline except from noon to 2 PM during the work-week looks more like an always off machine to our predictors. Thus, we categorize it as such.

global testbed, but there are few penalties for poor maintenance of one's contribution. Thus, when PlanetLab hosts break, they are prone to staying broken for an extended period of time. The likelihood of failure is exacerbated by the fact that PlanetLab experiments occasionally run wild and place undue wear on hard disks.

In contrast, each Microsoft hosts is typically administered by a single individual who is likely to quickly notice a failure. Thus, availability in the Microsoft trace is largely driven by the whim-driven behavior of these users. Many individuals, particularly developers, are prone to leaving their machines constantly powered up, explaining the significant population of always-on hosts. Non-technical staff are more likely to turn their computers on at the beginning of the work day and off at the end of it; this population is represented by the 10% of hosts which show diurnal behavior.

The specific classes in our availability taxonomy are somewhat arbitrary, since there are numerous ways that one could categorize uptime behavior. Nevertheless, using our simple classification scheme, we can demonstrate the existence of exploitable availability patterns in real networks.

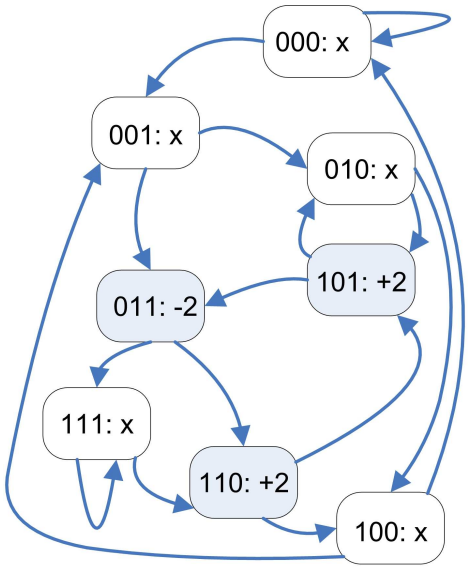
2.3 Forecasting Availability

In this section, we introduce several techniques for availability prediction. We then describe how to combine individual predictors into a "meta-predictor" which dynamically selects the best forecasting method for a particular host and a particular lookahead.

2.3.1 Saturating Counter Predictors

Our first predictor is the *RightNow* predictor. A host's current availability status is used as the value of all predictions for all lookahead periods. RightNow predictors are attractive because they require only one bit of state, and they should work well for machines which are predominantly online or predominantly offline. Unfortunately, they cannot accurately forecast periodic or long stretch behavior beyond the short term.

We can generalize the RightNow predictor to utilize n bits of state. For example, whereas the RightNow predictor represents uptime state using a single bit, the *SatCount-2* predictor uses a 2-bit saturating counter. Such a counter can assume four values (-2,-1,+1,and +2) which correspond to four uptime states (strongly offline, weakly offline, weakly online, and strongly online). During each sampling period, the counter is incremented if the host is online, otherwise it is decremented; increments and decrements cannot move the counter beyond the saturated counts of -2 and +2.



This is the de Bruijn graph for the uptime pattern $\{110\}^*$. Each vertex is labeled as `uptime_state:sat_counter_value`. A counter value of `x` means that the associated vertex has never been visited. Traversing an edge represents a left shift-and-fill of the starting node label.

Figure 2.2: History predictor example

Predictions for all lookahead periods use the current value of the saturating counter, i.e., negative counter values produce "offline" predictions, whereas positive values result in "online" predictions.

By using a few extra bits of storage, SatCount-x predictors are more tolerant than RightNow predictors to occasional deviations from long stretches of uniform uptime behavior. However, like the RightNow predictors, they cannot track more complex availability patterns.

2.3.2 State-Based Predictors

To predict the behavior of hosts with periodic availabilities, we turn to state-based predictors [2]. These predictors explicitly represent a machine's uptime history using a de Bruijn graph. A de Bruijn graph over k bits has a vertex for each binary number in $[0, 2^k\text{-}1]$. A vertex with binary label $b_1b_2...b_k$ has two outgoing edges, one to the vertex labeled $b_2b_3...0$ and the other to the vertex $b_2b_3...1$. In other words, the transition from a parent vertex to a child vertex represents a left shift of the parent's label and an addition of 0 or 1.

Suppose that we represent a host's recent availability as a k-bit binary string, with b_i equal to 0 if the machine was offline during the i^{th} most recent sampling

[2] Readers who are familiar with processor design will notice the similarity between our state-based forecasters and CPU branch predictors [72].

period and 1 if it was online. A k-bit de Bruijn graph will represent each possible transition between availability states. To assist uptime predictions, we attach a 2-bit saturating counter to each vertex. These counters represent the likelihood of traversing a particular outbound edge; negative counter values bias towards the 0 path, whereas positive values bias towards the 1 path. After each uptime sampling, the counter for the vertex representing the previous uptime state is incremented or decremented according to whether the new uptime sample represented an "online" edge or an "offline" edge.

To make an uptime prediction for t time steps into the future, we trace the most likely path of t edges starting from the vertex representing the current uptime state. If the last bit we shift in is a 1, we predict the machine will be online in t time units, otherwise we predict that it will be offline.

Figure 2.2 depicts the state maintained by such a *History* predictor. In this example, the host's availability has a periodicity of 3 samples and the repeated uptime string is 110. The machine does not deviate from this pattern, so only the three shaded vertices represent observable uptime states.

2.3.3 Tolerating Noise in the State Space

Suppose that a host has a fundamentally cyclical uptime pattern, but the pattern is "noisy." For example, a machine might be online 80% of the time between midnight and noon and always offline at other times. If the punctuated downtime between midnight and noon is randomly scattered, the de Bruijn graph will accumulate infrequently visited vertices whose labels contain mostly 1's but differ in a small number of bit positions. As the length of time that we observe the host grows, noisy downtime will generate increasingly more vertices whose labels are within a few bit-flips of each other. Probabilistically speaking, we should always predict that the host will be online from midnight to noon. However, the many vertices representing this time interval are infrequently visited and thus infrequently updated. Their counters may have weak saturations (-1 or +1) that poorly capture the underlying cyclic availability.

For hosts like this, we can nudge predictions towards the probabilistically favored ones by considering *superpositions* of multiple uptime states. Given a vertex v representing the current uptime history, we make a prediction by considering v's counter and the counters of all observed vertices whose labels differ from v's by at most d bits. For example, suppose that k=3 and d=1, and that each of the $2^k = 8$ possible vertices corresponds to an actually observed uptime history. To make a prediction

for the next time step when the current vertex has the label 111, we average the counter values belonging to vertices 111, 110, 101, and 011. If the average is greater than 0, we predict "online," otherwise we predict "offline."

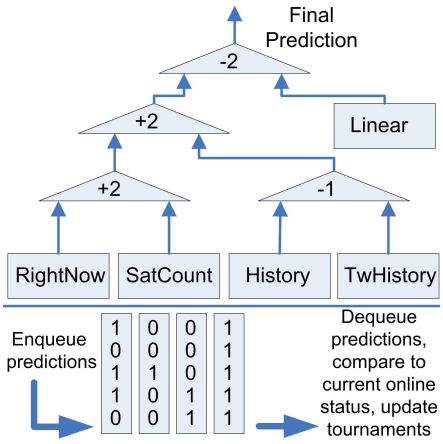
We call such a predictor a *TwiddledHistory* predictor, since it considers the current vertex and all "twiddled" vertices whose labels differ by up to d bits. The hope is that by averaging the counters of similar uptime states, we remove noise and discover stable underlying availability patterns. The TwiddledHistory strategy will perform worse than the regular History strategy when vertices within d bits of each other correspond to truly distinct uptime patterns. In these situations, superposition amalgamates state belonging to unrelated availability behavior, reducing prediction accuracy.

2.3.4 Linear Predictors

Linear prediction [51] is a common technique from digital signal processing and statistical time series analysis. It uses a linear combination of the last k signal points to predict future points. The k coefficients are chosen to reduce the magnitude of an error signal, which is assumed to be uncorrelated with the underlying "pure" signal. To make availability predictions for t time steps into the future, we iteratively evaluate the linear combination using the k most recent availability samples, shifting out the oldest data point and shifting in the predicted data point. Linear prediction produces good estimates for signals that are stable in the short term but oscillatory in the medium to long term [88]. We would expect this technique to work well with machines having diurnal uptimes, e.g., machines that are online during the work day and offline otherwise.

2.3.5 Hybrid Predictor

A machine can transition between multiple availability patterns during its lifetime. Furthermore, some availability patterns are best modeled using different predictors for different lookahead periods. To dynamically select the best predictor for a given uptime pattern and lookahead interval, we employ a *Hybrid* predictor. Our approach is similar in spirit to the "mixture of experts" strategy of the Network Weather Service [111], but closer in design to hybrid branch predictors [72]. For each lookahead period of interest, the Hybrid predictor maintains tournament counters. These saturating counters determine the best predictor to use for that lookahead period. For example, Figure 2.3 depicts a three-level tournament. Negative counter values select the left input, whereas positive values select the right. In this example, the SatCount



This figure depicts the tournament counters and update queue of a Hybrid predictor. The five bits in each queue entry represent the previous predictions of the five sub-predictors.

Figure 2.3: Hybrid predictor example

predictor is currently more accurate than the RightNow predictor. Similarly, the History strategy is outperforming the TwiddledHistory strategy. The best history-based approach is beating the best "simple" approach, and the Linear predictor performs worse than the best of the other four predictors. Thus, the final output of the Hybrid predictor is the History prediction.

Consider a Hybrid predictor making forecasts for an t-sample lookahead period. At the beginning of each time unit, the Hybrid predictor samples the current uptime state of its node. Its five sub-predictors are updated with this state, and each sub-predictor makes a prediction for n time units into the future. The final output of the Hybrid predictor is selected via tournaments as shown in Figure 2.3, and the individual sub-predictions are placed in a queue and timestamped with `curr_time` + t. If the head of the queue contains an entry whose timestamp matches the current time, the entry is dequeued and the tournament counters are updated using the dequeued predictions. A tournament counter remains unchanged if both of the relevant dequeued predictions match the current uptime state or both do not match. Otherwise, one prediction was right and the other was wrong, and the tournament counter is incremented or decremented appropriately. In the last stage of the update, the `curr_time` value is incremented.

If a Hybrid predictor is responsible for multiple lookahead periods, it keeps a separate update queue and tournament counter set for each period. Each queue and counter set is maintained using the algorithm described above.

2.3.6 The Predictability of Microsoft and PlanetLab Hosts

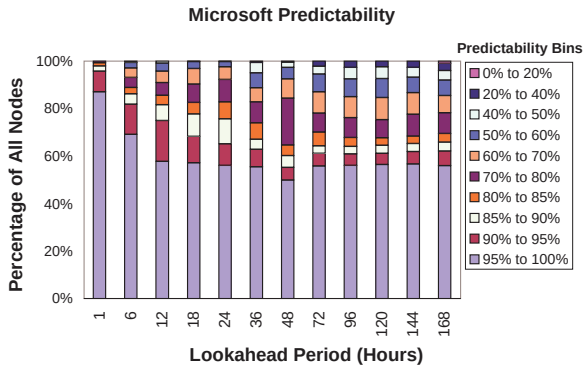
In Section 2.2.2, we demonstrated the existence of regular availability patterns in the Microsoft and PlanetLab systems. To determine whether our prediction techniques could detect this regularity, we associated a unique Hybrid predictor with each host. We used the first two weeks of uptime data to train each predictor; two weeks was a reasonable training length because it gave a predictor two chances to observe uptime patterns with a periodicity of a full week. After training the predictors, we evaluated their accuracy by comparing their predictions to the remaining three weeks of availability data. During each hour in the evaluation period, the predictors made forecasts for multiple lookahead intervals and were updated with uptime samples from that hour. We say that a node is p-predictable for a certain lookahead period if we predicted its uptime behavior with at least accuracy p.

Each Hybrid predictor used 3-bit saturating tournament counters. The organization of the tournaments resembled the structure shown in Figure 2.3. However, instead of a single Linear predictor, two Linear predictors competed against each other—one tracked 168 bits of history and the other tracked 336 bits. Also, the History and TwiddledHistory predictors were replaced with two two-level tournaments, allowing competitions between the same predictor type with different k values. The single History predictor was replaced with a two-level tournament comparing k values of 6, 24, 48, and 56; the same k values competed in the TwiddledHistory multi-level tournament. In all experiments, the SatCount predictors used 2 bits of uptime state, and the TwiddledHistory predictors used a d of 1.

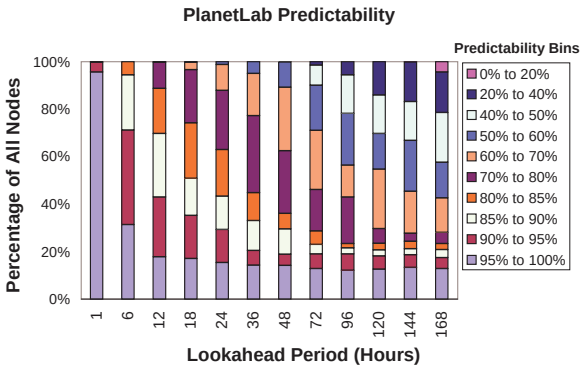
Figure 2.4 bins the predictability of individual Microsoft and PlanetLab hosts for several lookahead intervals. For a 1-hour lookahead period, 95.6% of PlanetLab machines can be predicted with greater than 95% accuracy, as compared to only 87.0% of Microsoft hosts. However, as the lookahead period increases, the percentage of PlanetLab hosts that are 95%-predictable quickly drops below 20%. In contrast, slightly over half of the Microsoft hosts are 95%-predictable across all lookahead intervals.

As the lookahead period increases, the predictability decay of the Microsoft machines is more graceful than that of the PlanetLab hosts. For example, with a 144 hour lookahead period, 72.5% of the PlanetLab machines have worse than 70% predictability; in the Microsoft data set, only 22.6% of the hosts are this bad. In fact, for all of the studied lookahead periods, no more than a fourth of the Microsoft nodes are ever worse than 70%-predictable.

In the context of Figure 2.1, these differences make sense. Roughly 60% of Mi-



(a) A stable core of Microsoft nodes remains highly predictable across all lookahead periods.



(b) PlanetLab nodes start out highly predictable, but their predictability quickly degrades. Overall, these nodes are less predictable than the Microsoft set.

Figure 2.4: Microsoft and PlanetLab predictability

crosoft hosts are always on, and these hosts are trivially predictable for arbitrary lookahead periods. In contrast, PlanetLab is dominated by long stretch hosts. These machines are highly predictable in the short term—given the current uptime state of a long stretch hosts, we can confidently predict that it will remain in that state for the next few hours. However, these machines are increasingly unpredictable for larger lookahead intervals because their uptime regimes lack periodicity. Once such a host changes uptime state, it will keep that state for many hours, but the arrival of these changes are random. Thus, the predictability of the PlanetLab system as a whole degrades for long lookahead periods, even though it is more predictable than the Microsoft network in the short-term.

As indicated by Figure 2.1, diurnal behavior is unknown amongst PlanetLab hosts. However, roughly 10% of Microsoft hosts have daily periodicity. These machines are often highly predictable, although this is not always true. For example, some machines are only work-week periodic to the extent that they are offline during non-work day hours; during the actual work day, they may have erratic availability

that is difficult to predict. Also, some diurnal machines are occasionally left online for multiple work days, or left online during the weekend. Such aperiodic behavior is difficult to forecast, and it may introduce "useless" history into the state-based predictors.

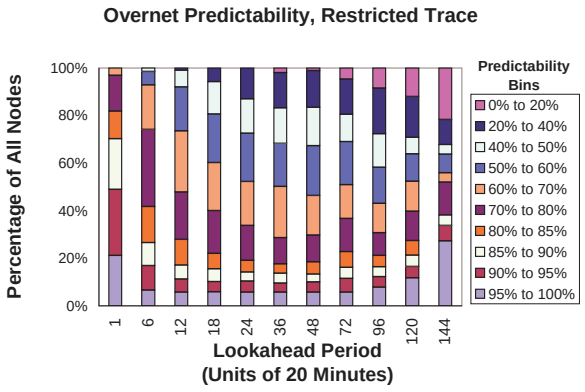
2.3.7 The Predictability of Overnet Hosts

Up to this point, we have defined availability in terms of network reachability—a a host is available if it responds to a ping. This notion of availability is reasonable for distributed applications which run as core system services. For example, in the FARSITE cooperative file system [1], users can turn their machines on and off as they please. However, hosts are expected to run a FARSITE server whenever they are online. Thus, from the perspective of the distributed protocol, a host that responds to pings is one that is available to the FARSITE collective.

Using this definition of availability, Microsoft and PlanetLab machines have fairly long session times; a host that has just come online will likely stay online for multiple consecutive hours. Availability is more capricious when we define it in terms of user-level applications that are intermittently active. In these scenarios, a host is online only if it is network accessible and its user has decided to run the program of interest. In application-level cooperative systems like Gnutella and Napster, churn rates are much higher, with typical session times on the order of tens of minutes [17, 95]. A natural question is whether these uptime patterns are amenable to our prediction techniques.

Unfortunately, it is difficult to collect longitudinal data about the peer-to-peer applications that are actually deployed. Many protocols are proprietary (e.g., [50, 112]) and obfuscated to protect user information and impede outside analysis. Studying open protocols may place a research institution in legal danger, since content owners may demand access to the data to prove allegations of illegal content sharing. Due to these issues, many empirical traces of peer-to-peer activity are of short duration. Nevertheless, we will mine the Bhagwan Overnet trace [17] for preliminary insights into uptime prediction in user-level cooperative applications.

The Bhagwan trace spans the seven day stretch between January 15 and January 21, 2003. It measures the availability of 1,468 randomly selected peers using a sampling period of 20 minutes. Given a seven day observation period, this means that each host has 504 data points. The Overnet trace is thus much smaller than the Microsoft and PlanetLab ones, both in terms of "wall time" duration and the number of data points per host.



Our availability predictions are less accurate for the Overnet trace than for the Microsoft and PlanetLab data sets.

Figure 2.5: Overnet predictability

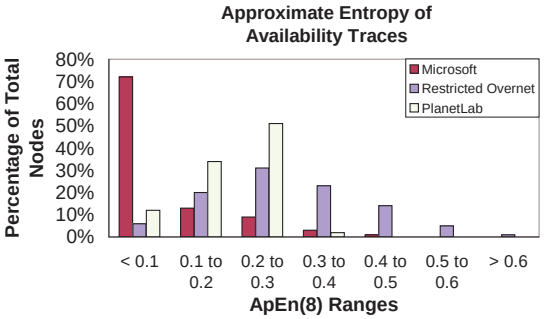
To evaluate availability forecasting in the Microsoft and PlanetLab networks, we trained our predictors on two weeks of data. Since this data had a measurement granularity of an hour, this resulted in 336 training samples and 504 evaluation samples. For a fair comparison, we also trained our Overnet predictors on 336 samples; this left only 168 samples for evaluation purposes [3]. However, we did filter out nodes that were not online at least once in the first 100 samples and the last 100 samples. This created a more challenging prediction environment, since many Overnet nodes were almost always offline and thus easy to predict. Bhagwan also estimated a non-trivial attrition rate of 32 hosts per day in the full node set [17]. This thesis does not focus long-term attrition effects, so the smaller trace set (haphazardly) filters out some of these "permanently" lost nodes.

In the full, unfiltered Overnet trace, about a third of all peers are 95% predictable for all lookahead periods; however, these are nodes which are almost always off. Prediction is much more difficult in the filtered trace. As depicted in Figure 2.5, less than 10% of the peers in the restricted trace are 95%-predictable for an arbitrary lookahead period. Indeed, the overall predictability of the restricted Overnet trace is much worse than that of the Microsoft or PlanetLab system. An obvious question arises: why is the behavior of Overnet hosts so difficult to forecast?

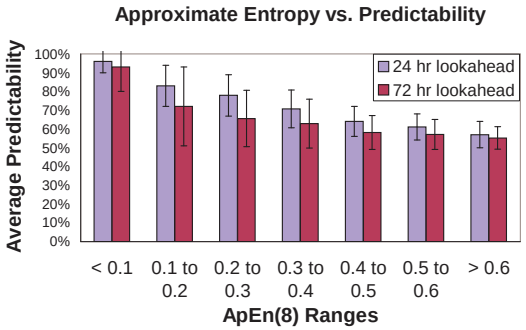
2.3.8 Entropy and Predictability

We must interpret the Overnet results from the previous section with caution, since we have fewer evaluation samples than in the PlanetLab/Microsoft experiments

[3] We did not selectively pick hourly samples as we did for the PlanetLab trace because a probing granularity of 20 minutes is appropriate for a network with high churn rates.



(a) This chart categorizes the approximate entropies of the availability strings in each trace set.



(b) This graph shows the relationship between the entropy of a Microsoft availability string and its predictability. Each y-bar represents a standard deviation.

Figure 2.6: Approximate Entropy Results

and we cannot fully control for long term node attrition. We also do not have enough samples to confidently apply our availability taxonomy from Section 2.2.1. However, manual inspection of the three availability traces reveals qualitative differences in uptime patterns. Overnet nodes appear to have more long stretch downtime than Microsoft or PlanetLab nodes, but the online stretches of Overnet nodes seem more randomly punctuated by bursts of downtime. This implies that Overnet nodes with non-trivial amounts of uptime should be more difficult to predict than Microsoft or PlanetLab nodes with similar uptime percentages.

To quantify this intuition, we use the information theoretic concept of approximate entropy [87], denoted `ApEn(x)`. Given an integer `m` and a bit string s whose length is much larger than `m`, `ApEn(m)` represents the additional information provided by the last bit of an `m`-bit substring, given that we already know the first `m-1` bits. `ApEn(m)` is highest when all `m`-bit substrings within s have equal frequencies. When approximate entropy is low, s has repeated patterns and is non-random. As a simple example, consider the string $\{10\}^*$. Knowing the first bit of a two bit substring allows perfect prediction of the following bit. Thus, `ApEn(2)` is close to 0.

Figure 2.6(a) bins the approximate entropies of the uptime strings in the Microsoft, PlanetLab, and restricted Overnet traces. This figure validates our intuition that Overnet hosts have less regular availability patterns. In the Microsoft and PlanetLab networks, the vast majority of `ApEn(8)` values are smaller than 0.3, but 42% of Overnet nodes have an `ApEn(8)` greater than 0.3. Figure 2.6(b) plots `ApEn(8)` versus predictability for the Microsoft data set, confirming that higher entropy values are indeed correlated with lower predictability.

Also note that PlanetLab availability has higher entropy than Microsoft availabil-

ity. Referring to Figure 2.1, we see that the PlanetLab network has twice as many unstable10to50 nodes, which we would expect to be quite random. More importantly, PlanetLab is dominated by long stretch nodes instead of always on nodes. This also increases system entropy, since long stretch nodes have less regularity than always on machines.

Nodes which display erratic behavior when considered in isolation may show emergent periodic behavior when considered in aggregate. This notion is supported by the Overnet trace, which shows diurnal periodicity at the global level. By expanding the notion of superposition to include clusters of machines, we may be able to diminish the impact of entropy upon prediction accuracy for single nodes. Devising such techniques is an important avenue for future research.

2.4 Discussion

The Microsoft, PlanetLab, and Overnet traces were collected using a centralized probing infrastructure. Centralized collection makes analysis easy since there is no need to collate information scattered across the wide area. However, centralized measurement may be unattractive for reasons of scalability and trust. Additionally, probing from a single vantage point may not produce completely generalizable results, since availability measurements can be affected by one's location in the network. Even if a host is up and running, routing failures may make that host unreachable from certain parts of the network. Centralized probing cannot capture these viewpoint-dependent disruptions in availability.

Decentralized probing can alleviate some of these issues by collecting measurements from multiple vantage points. However, decentralized data collection also raises issues of trust, since some hosts may not trust the measurements of others. Indeed, even in a centralized probing system, malicious hosts can ignore ping requests that were successfully received, or try to respond to pings that were sent to offline hosts. We revisit some of these issues in Chapter V. However, developing threat models for availability-aware systems remains an important topic for future work.

The Microsoft and PlanetLab traces used pings to determine availability. If machines receive IP addresses in a non-static way, e.g., from DHCP or NAT, and the probing infrastructure is unaware of such dynamic assignments, then ping-based probing can lead to an overestimation of the number of hosts and an underestimation of host availability [17]. Fortunately, IP aliasing should be rare in both of these traces. The Microsoft study performed name lookups before each round of pinging so that availability strings could be assigned to specific machines instead of specific

IP addresses. Furthermore, as stated in the instruction manual for PlanetLab administrators, the primary IP address for each PlanetLab node should be a static one.

Aliasing is not a problem if local system logs are used to track availability [98]. Such an approach also allows one to measure uptime exactly, as opposed to estimating it via sampling. However, as mentioned above, network reachability is a key component of availability in distributed systems. Since locally measured uptime is not always synonymous with "online and reachable" time, some applications may prefer to measure availability using network probes. Also note that the finer temporal resolution of system logs is not necessarily useful for availability prediction. This is because very brief downtime is generally noise which should be ignored. For example, downtime due to software upgrades or rejuvenation rebooting is usually aperiodic (making it difficult to predict) and fairly brief (so that it has little impact on a host's overall availability profile). Thus, such events should be omitted from the history that is used to make predictions, since they will only obscure the fundamental availability trends. If such events are substantial causes of downtime, then the sampling interval can be decreased. The sampling interval should also be selected with an eye towards typical session times. For example, session times are shorter in Overnet than in PlanetLab, so Overnet nodes should be sampled more frequently than PlanetLab ones.

In Section 2.3, we described two basic methods for predicting availability: Markov modeling and linear prediction. Many other forecasting techniques exist, such as fractal prediction [89],"universal" prediction [73], and the simplex projection method [104]. We investigated the applicability of these three techniques to our availability traces, but we found that none of them were significantly better than a Hybrid predictor containing Markov forecasters and linear predictors. We also found no advantage to creating Hybrid predictors which contained these additional forecasting techniques as sub-predictors.

2.5 Conclusions

By definition, distributed systems contain hosts which are scattered across the wide area. Historically, these hosts have been divided into two categories. Servers were responsible for coordinating access to key system state, and clients interacted with the servers on behalf of end users. Given the privileged role of servers, they were expected to be well-administered and highly available. The availability of clients was interesting only to the extent that client interactions generated load on the servers.

As Internet-enabled machines became ubiquitous, the potential client population exploded. In many cases, implementing services atop a small number of centrally-maintained servers became unattractive, both for reasons of scalability and trust. In reaction to these trends, software architects devised serverless or peer-to-peer designs; in these systems, each participating machine is both a client and a server who is responsible for maintaining a fraction of the global system state. These frameworks can potentially harness a vast amount of aggregate computing power, but they introduce several new problems. In particular, it is difficult to implement a robust service atop a server set whose composition is constantly changing.

To deal with this problem, designers have traditionally made pessimistic assumptions about the churn rate and over-engineered their systems to function in the worst imaginable environments. This approach implicitly assumes that availability fluctuation is an inscrutable phenomenon. However, as we have shown in this chapter, churn is not a random process. Many hosts behave in regular ways. Even more importantly, this behavior is often predictable. In the next chapter, we explain how to leverage availability prediction to improve performance in a wide variety of distributed systems.

CHAPTER III

Exploiting Availability Prediction

The previous chapter demonstrated that networks contain identifiable availability patterns. It also introduced methods for predicting the availability of individual hosts. In this chapter, we describe several applications of availability prediction which improve system performance or robustness. Although networks often contain erratic hosts whose behavior is difficult to model, our optimizations only require that *some* hosts have exploitable regularity. In the unlikely event that no such regularity exists, our availability-aware heuristics do no harm.

This chapter describes four distinct applications of availability prediction. The first is a distributed hash table (DHT) which preferentially stores data on highly available nodes, reducing regeneration bandwidth and increasing data availability. In the second application, an overlay uses availability prediction to modulate the rate at which routing peers are probed for liveness. The result is a dramatic reduction in probing bandwidth with minimal degradation in routing table freshness. The third application is a delay-tolerant network which uses availability forecasts to create routes with low delivery latencies. The final application introduces availability-aware models for malware propagation. These models are more accurate than availability-agnostic ones, leading to better infection forecasts for both white hats and black hats. In fact, the latter group can use these forecasts to pick the most damaging time to launch a worm.

3.1 Availability-aware Replica Placement

In a cooperative storage system, each host maintains a fraction of the global file set. Each data object is replicated across multiple hosts, both to improve short-term availability and long-term persistency. When a replica site goes offline, its objects are typically copied from other replicas and regenerated at a new site [91, 101]. This

policy ensures that redundancy targets are always satisfied. However, each object copy necessitates a wide-area data transfer. If there are many objects, or objects are large, or host churn is high, this regeneration policy will be quite expensive [20].

The TotalRecall system [18] provides an alternative to what it calls "eager" replication policies. TotalRecall estimates global availability at night (when it is presumed to be lowest), and then replicates data beyond the required redundancy target such that the target is always met, even at night. For example, suppose that the minimally acceptable replication factor is 2, and that 50% of hosts are offline at night. TotalRecall will initially replicate an object 4 times, but it will not perform eager regeneration unless it actually observes an object's replication level to drop below 2. This should be rare if the worst-case estimate of global availability is pessimistic enough.

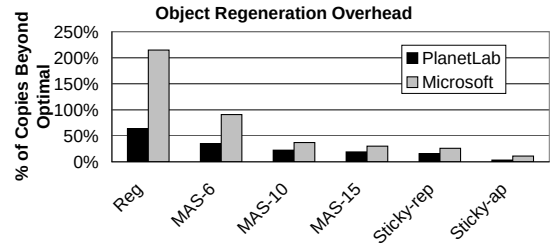
Using this "lazy" replication policy, TotalRecall requires less network bandwidth than the eager strategy. However, lazy replication requires an increase in the aggregate storage burden. Availability prediction allows for a third approach. When a replica site goes down, we can copy its objects onto a host that is predicted to be online for a long time. If our predictions are correct, then biasing data towards such hosts will reduce the number of regeneration copies without increasing the global storage requirements.

3.1.1 Availability-aware Data Placement

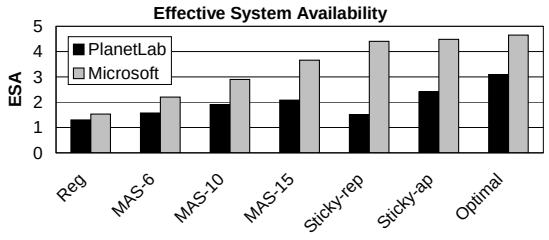
In the text below, we describe availability-aware replication within the context of an overlay-based storage system. For the sake of concreteness, we use the Chord overlay, which uses a ring-based routing geometry [101]. However, our results are easily extended to other topologies.

Each Chord peer has a 160-bit overlay identifier, typically the hash of its IP address. The 2^{160} possible identifiers form a circular address space; the *successor* of a node is the first online machine with a larger identifier `mod` 2^{160}, and the *predecessor* is defined similarly. Each node tracks s immediate successors as well as several routing table peers. Through clever selection of routing table entries, nodes only need to maintain $O(logN)$ entries to provide $O(logN)$ route length. We refer readers to [101] for more details on routing table maintenance.

In a Chord-based storage system, an object is stored on the first node whose identifier is larger than the hash of the object. If replication is desired, the k replicas are stored on the first k nodes with larger identifiers. The node immediately preceding the replica sites for an object is that object's *replica manager*. Queries for that object



(a) Availability-guided data placement reduces the copy overhead incurred when replica sites depart.



(b) By biasing objects towards nodes which will be online for several consecutive hours, data availability also increases.

Figure 3.1: DHT simulation results

are routed to the replica manager, who responds to the initiator with the IP addresses of the replica sites.

We investigate five replication strategies. The first two do not use availability prediction. In the *regular* replication method described above, objects must be regenerated whenever a replica site leaves or a node join causes the first k successors of an object id to change. In the *sticky replica* strategy, a newly entering node N places replicas on its first k join-time successors. N continues to use a replica site until that site leaves the overlay, at which point it is replaced with the first successor of N that is not already a replica site for N. The sticky replica strategy requires fewer object copies than the standard scheme since only node leaves cause replicas to be transferred. Unfortunately, the overlay identifiers for an object's replica sites are no longer a simple function of that object's id. If a replica manager goes down unexpectedly, the pointers to its replica nodes are lost, and the associated data cannot be quickly rediscovered [28]. To guard against this, each manager backs up its replica site pointers on its first k successors. When a manager leaves the overlay, its predecessor can find the replica sites for the new objects it manages and copy the necessary data to its replica sites.

The next two replication strategies use availability prediction to guide replica placement. Each node has a Hybrid predictor that it updates every hour. After each update, the node estimates the remaining number of hours that it will remain

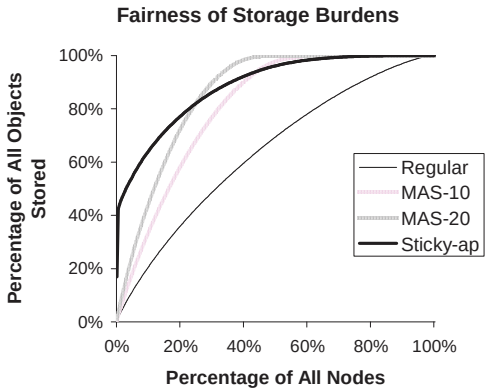


Figure 3.2: Storage skew in the Microsoft DHT

online, making iterative availability predictions for 1 hour into the future up to some maximum lookahead period. During each iteration, the estimate is incremented by 1 if the Hybrid predictor outputs "online" *and* the observed accuracy of predictions for that lookahead period surpasses a minimum threshold; if either condition is false, iteration terminates. Nodes periodically exchange their estimated availabilities with the other peers in their routing tables. These values are piggybacked atop standard routing stabilization messages [101]. In the simulation results given later, the maximum lookahead period was 15 hours and the minimum confidence level was 90%.

In the *most-available successors* replication strategy, abbreviated MAS-j, a replica manager places objects on the k most available of its first j successors, where $k \leq j \leq s$. A node's replica site is sticky as long as it remains one of the first j successors. The *sticky replicas with availability prediction* scheme, abbreviated sticky-ap, features unconstrained attachment to replica sites as in the regular sticky strategy. Additionally, when a node picks a new replica site, it picks the most available of its immediate s successors that is not already a replica site.

For comparison purposes, we also study the *optimal* placement strategy. This strategy is like sticky-ap, but the availability predictor is an oracle. When a new replica site must be picked, the optimal scheme selects the node in the first s successors that will definitely be online for the longest consecutive period.

3.1.2 Evaluation

Figure 3.1 shows DHT performance when availability is driven by the Microsoft or the PlanetLab trace. The results were produced using a derivative of the well-known Chord simulator [101]. Leaves and joins that happened in the same hour in an availability trace were uniformly and randomly distributed across the corresponding

hour in the simulation. All simulations ran for 504 virtual hours. The simulated Microsoft DHT contained 1000 nodes and the simulated PlanetLab one contained 321 nodes. 2% of DHT operations were writes and 98% were reads. In aggregate, the Microsoft DHT issued about 660 requests each minute according to a Poisson distribution. The PlanetLab DHT used the same per-node request rate, but due to its smaller size, it only issued about 210 aggregate requests per minute. The replication factor was 4 in both DHTs, and each node's routing table tracked 20 immediate successors. Each replication strategy was tested five times. The ith run for each strategy used the same set of nodes, but this set was changed for each value of i. Standard deviations were less than 3% amongst the trials for a particular replication strategy.

Figure 3.1(a) shows that availability-guided replication results in many fewer copies due to replica regeneration. The savings are greatest in the Microsoft DHT, with the regular replication strategy requiring 215% more copies than optimal and the sticky-ap strategy requiring only 11% more copies than optimal. The gains are smaller in the PlanetLab system, with the regular replication strategy requiring 65% more copies than optimal and the sticky-ap strategy requiring 3.4% copies beyond optimal. The discrepancy in savings is primarily explained by the fact that the Microsoft system had more hosts that were always online. Biasing data towards these peers will reduce overhead more than biasing data towards long stretch nodes that will be online for several consecutive hours, but will still eventually go offline and require object copying. Also, the Microsoft DHT had work-week periodic nodes, unlike the PlanetLab one. Although work-week periodic nodes may often be offline, we can still take advantage of phase-shifted diurnal patterns to reduce object copying (see the example in Section 6.3).

We should distinguish between the savings derived from having sticky replica sites and the savings produced by clever choice of these sites. For example, in the Microsoft DHT, if we compare the sticky-rep strategy with sticky-ap, we see that sticky-ap required roughly 7.3 million copies, whereas the sticky replicas strategy required about 8.3 million copies. This reduction of a million copies represents the savings from quickly identifying highly available nodes, as opposed to hoping that your first k successors are highly available and having to regenerate their replicas if you are wrong.

Figure 3.1(b) describes the effective system availability (ESA) of the two DHTs using each replication strategy. ESA expresses global object availability in units of "nines." For example, if an arbitrary object is accessible via some replica site 99%

of the time, the system-wide object availability is 0.99 or 2 nines of availability. We defer a more detailed discussion of ESA to Douceur's work [39].

Looking at Figure 3.1(b), we see that ESA improves as the DHT has more freedom to bias data towards highly available nodes. Once again, the improvement is greater in the Microsoft system. For example, in the Microsoft DHT, MAS-15 more than doubles the baseline ESA, adding 2.17 nines. In the PlanetLab DHT, MAS-15 improves ESA from 1.31 to 2.08.

If a node uses a replication strategy with unconstrained stickiness, it may place data on peers that move beyond its first s successors. In these scenarios, the node will have to issue extra pings to ensure that these replica sites are online. If objects are very small or network bandwidth is very constrained, the relative cost of additional heartbeat messages may necessitate schemes like as MAS-10 which place data on peers that would already be pinged.

Also note that there is a tension between reducing data regeneration and maintaining equitable storage loads. This tension is depicted in Figure 3.2, which shows the cumulative distribution of storage burdens with respect to the number of nodes in the Microsoft DHT. The line $y = x$ represents a perfectly equitable storage load, i.e., $X\%$ of the total objects would be stored on exactly $X\%$ of the nodes. The regular replication strategy was close to this line, although its curve was slightly convex since real-life storage burdens will never be perfectly uniform. The distributions for the availability-guided replication strategies were much less equitable. For example, in the regular replication scheme, 1.4% of nodes stored less than 100 objects per online hour. With MAS-20, 57.0% of peers stored less than 100 objects per online hour. The skew was even more dramatic for the sticky-ap scenario, where 10% of nodes stored 64% of the objects.

The system designer must balance competing requirements for high ESA, low bandwidth usage, and equitable storage burdens. The threat model must also be considered. If nodes are untrusted, storing most data on a few nodes may amplify an attacker's ability to delete or modify objects. Studying these trade-offs is an important area for future work.

3.2 Availability-modulated Overlay Probing

In a peer-to-peer overlay, hosts opt in and out at will. Such membership churn requires nodes to periodically test the availability of their peers. Ideally, the probe rate would be frequent enough to detect peer disconnections quickly, but no faster, since unnecessary probe messages are pure overhead. Most overlays use a static

ping rate which is pessimistically chosen to provide lower bounds on routing table consistency [91, 101]. However, schemes which probe each peer at the same rate are vulnerable to two problems. If the churn rate is underestimated, then bandwidth will be wasted on unnecessary probes. If churn rates are overestimated, then nodes will be slow to detect the exits of their routing peers, and routing tables will become less consistent.

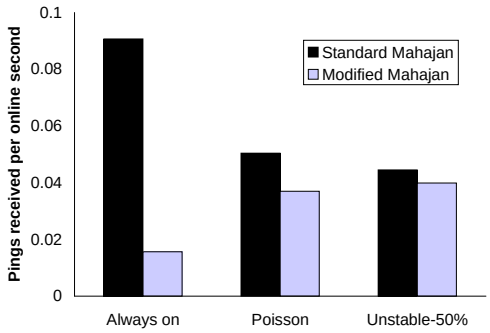
Mahajan's overlay allows each node to dynamically select the rate at which it probes its peers [70]. Hosts introspect the session times of their routing peers and estimate the average churn rate in the system. Routing peers are then probed at the minimal rate which maintains routing table consistency, assuming the average churn rate is a constant churn rate [70]. This approach provides a large reduction in probe traffic. However, it assumes that the average session time can predict the behavior of a particular node. This assumption will be false if availability is sufficiently heterogeneous. In these cases, some peers will be probed too often, which wastes bandwidth, and other peers will be probed too infrequently, which decreases routing table consistency.

Using *per-host* availability predictions, we can tailor ping rates for each targeted host, probing nodes that are about to disconnect more frequently than nodes which will be online for a while. In networks containing both highly available and erratically available nodes, availability-modulated probing (AMP) should require much less bandwidth than Mahajan's scheme but provide similar levels of routing table consistency.

3.2.1 Design

In AMP, each node tracks its own uptime history and maintains an availability predictor for lookaheads of one time unit up to some maximum m. For a given node, let c_τ be the historical likelihood that predictions for a τ-sample lookahead were correct—in other words, c_τ represents our empirical confidence in such predictions. We define a node's predicted remaining uptime t_{rem} as the number of consecutive sampling periods starting from the current time t for which the node is both predicted to be online at $t+\tau$ and c_τ is greater than some minimum confidence c_{min}; the largest possible t_{rem} is m. Nodes predict their own t_{rem} values and piggyback them upon preexisting overlay control messages.

We use t_{rem} values to modulate the heartbeat frequency provided by Mahajan's self-tuning algorithm [70]. Given the Mahajan period ω with which a host would ping all the peers in its routing table, we define a per-peer period $max(t_{rem}, 1) \cdot \omega$.



The more predictable a host's uptime is, the more that AMP can lower probe rates towards that host.

Figure 3.3: Ping distribution in the synthetic trace

In other words, the further a node is from its expected disconnection time, the less frequently it is probed. When disconnection is predicted to be imminent, the ping rate collapses to the standard Mahajan rate.

If a system contains many highly predictable nodes, we expect AMP to offer large reductions in probing bandwidth. If most nodes are unpredictable, we hope that AMP will perform similarly to an availability-agnostic scheme. This will happen if our predictors have little confidence in their forecasts and set t_{rem} to zero.

3.2.2 Evaluation

To determine whether AMP can take advantage of uptime heterogeneity, we created a custom network simulator for the Pastry overlay [91]. Each simulated Pastry node was assigned an availability predictor with the parameterizations described in Section 2.3.6. Host availability was driven by an external uptime database which could contain synthetic or empirically gathered availability data. In all experiments, predictors were trained on 336 hours (two weeks) of availability data before being evaluated. By training our predictors for two weeks, we gave them two chances to observe uptime patterns with a periodicity of a full week. In all experiments, the standard and modified probing schemes had a target loss rate of 2%, i.e., no more than 2% of messages forwarded by a node should be forwarded to a disconnected peer. The maximum lookahead period for availability prediction was five hours, and c_{min} was 90%. For each availability trace, joins and leaves observed in the same hour were randomly and uniformly scattered throughout the corresponding time unit in the simulation.

Detecting Heterogeneous Availability

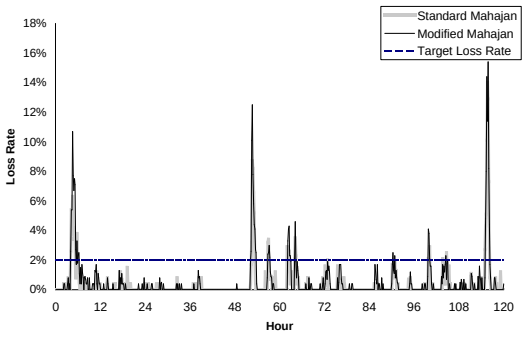
The success of AMP hinges on its ability to identify predictable availability patterns and lower probe rates towards the corresponding hosts. As a basic test of AMP's ability to perform such differentiation, we generated a 1000 node synthetic availability trace which contained three types of uptime patterns. A third of the nodes were always online. Another third had uptimes and downtimes governed by a Poisson process with a mean of ten hours. The last third randomly participated in the network with 50% likelihood each hour. Hosts in the first set should be perfectly predictable. Hosts in the second set should be somewhat predictable, since a host that has just joined will likely stay online for a few hours. Hosts in the last set are very unpredictable and should be pinged often.

Figure 3.3 shows the average number of probes per second that each node type received, as averaged over twenty simulation runs of 168 virtual hours. Interestingly, in the standard Mahajan scheme, always on nodes received twice as many probes as the unstable nodes! This may seem counterintuitive, but consider what happens when a group of unstable nodes leaves the network during a short time period. The clustered exits trigger an increase in the probing rate, but the exited nodes cannot receive these pings because they are offline. The always on nodes *can* receive these probes. Thus, they become the unnecessary targets of ping surges when a flurry of nodes leave the network.

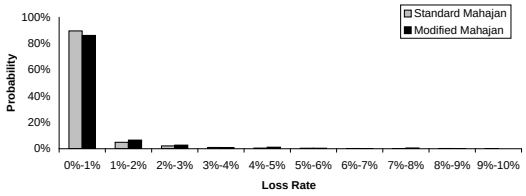
As shown in Figure 3.3, AMP successfully identified always on hosts and probed them far less frequently than other hosts. The ping rate towards Poisson nodes decreased by a quarter—once an availability predictor observed several sessions lasting multiple hours, it forecasted that when a Poisson node came online, it would stay online for a while. The probe rate towards unstable nodes *decreased* slightly, which was not desirable. This occurred because a node with random uptime will occasionally display long stretches of contiguous uptime. This may temporarily convince its uptime predictor that it is more available than it really is.

3.2.3 Real-life Availability Traces

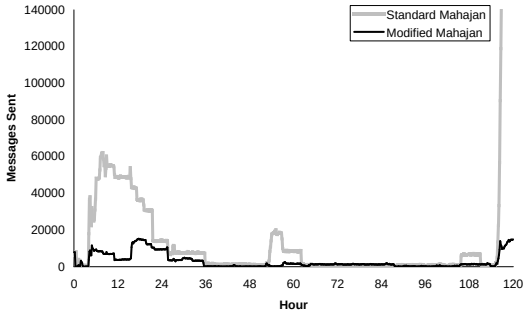
For our next test of AMP, we simulated two different overlays using empirically gathered availability traces. The first overlay was driven by a trace from the Microsoft corporate network [21], and the second was driven by availability data from the PlanetLab distributed test bed [102]. In the Microsoft network, over 60% of all nodes were almost always online, 21% of all nodes had long-stretch availability, and 10% of nodes had diurnal uptime. In contrast, 16% of all PlanetLab hosts were



(a) The standard Mahajan scheme had an average loss rate of 0.5%, whereas AMP had an average of 0.6%. Each data point represents the loss rate over a ten minute interval.



(b) Both schemes had similar pdfs for loss rates. The worst observed loss rate for the standard scheme was 17.2%. The maximum loss rate for AMP was 23.8%. These results represent the average of 100 simulation runs of 120 virtual hours each.



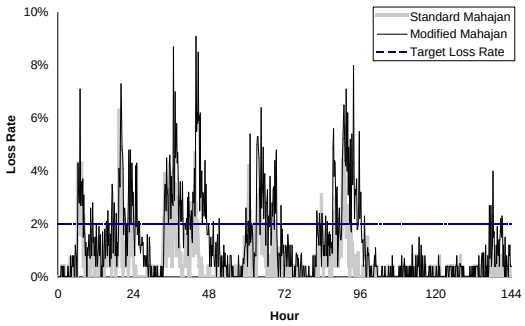
(c) AMP sent many fewer probes during punctuated, aperiodic drops in global availability. Each data point represents the number of probes sent over a ten minute interval.

Figure 3.4: PlanetLab simulations (321 nodes)

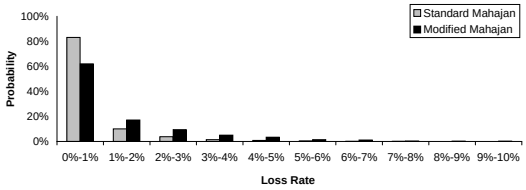
always online, 68% were long stretch, and none had diurnal availability [74].

Figure 3.4(a) depicts typical simulation runs for AMP and the modified Mahajan scheme in the PlanetLab overlay. Both approaches kept average loss rates below the desired ceiling, although loss rates displayed aperiodic spikes caused by short-lived depressions in global availability. These infrequent anomalies were likely caused by runaway experiments across the distributed testbed. For example, network congestion caused by chatty programs can lead to availability probes being dropped, making hosts which are actually online appear to be disconnected.

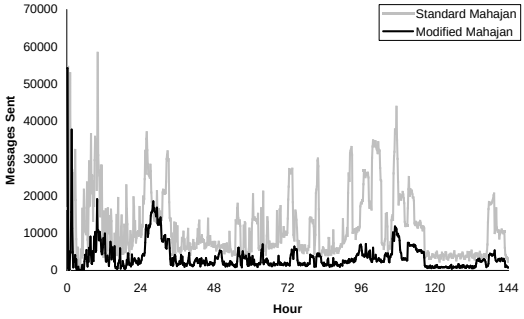
Figure 3.4(b) shows pdfs for loss rates under the two probing methods. The standard Mahajan scheme satisfied the loss target 95% of the time, whereas AMP met the target 93% of the time. This small reduction in compliance was offset by large savings in probe traffic, as shown in Figure 3.4(c). When global churn rates were low, AMP was only slightly better than Mahajan's scheme; however, during aperiodic high churn events, AMP avoided the massive probing waves launched by the standard Mahajan approach. During the five simulated days, AMP reduced overall



(a) The standard Mahajan scheme had an average loss rate of 0.5%. AMP's average loss rate was 1.2%.



(b) The maximum observed loss rate for the standard Mahajan scheme was 7.9%. The worst case loss rate for AMP was 9.1%.



(c) AMP results in bandwidth savings of 70%.

Figure 3.5: Microsoft simulations (5000 nodes)

probing traffic by over 80%. The reason is that AMP's predictors viewed rare, aperiodic events as "noisy samples" in fundamentally regular availability patterns. Thus, aperiodic events did not radically alter AMP's probing rates. In contrast, Mahajan's estimators lacked the historical perspective needed to implement hysteresis. Thus, punctuated drops in global availability always led to massive probing waves.

For overlays with rigid constraints on routing table consistency, the standard Mahajan reaction may be desirable. However, we believe that many overlays will gladly trade slightly decreased routing consistency for massive decreases in probing traffic. This trade-off will be particularly attractive to overlays which must scale to millions of machines, since the maintenance bandwidth in such overlays is daunting [20].

Figure 3.5 shows simulation results for the Microsoft availability trace. AMP reduced probing bandwidth by 70% while keeping average loss rates below the target loss rate. However, when compared to its performance in the PlanetLab network, AMP exceeded the loss target more often. Over all ten minute intervals, the standard Mahajan scheme satisfied the loss goal 93% of the time, whereas AMP satisfied the goal only 79% of the time. This performance degradation arose because the Microsoft trace contained hosts with diurnal uptime patterns. Some of these hosts were less

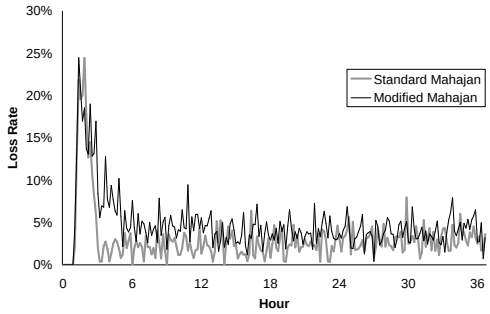


Figure 3.6: Loss rates when all nodes are unstable

predictable than one might expect. For example, some of them were occasionally left online during the weekend or in-between work days; others were infrequently left offline during the work day. Such behavior was aperiodic, but unlike the global availability depressions in the PlanetLab network, it was not short-lived, often spanning half a day or more. Aperiodic behavior of this duration will *not* be ignored by our availability predictors, even though such behavior is actually still noise. Using this noise to predict future availability leads to inaccurate forecasts and thus less consistent routing tables.

Since diurnal nodes are particularly susceptible to aperiodic, medium-duration uptime anomalies, AMP should modulate their probe rate using a smaller window size or a higher accuracy threshold. AMP could automatically detect whether a node was diurnal by applying Fourier decomposition to its availability history (see Section 2.2.1). Exploring such adaptive techniques is an important area for future research.

3.2.4 AMP in Unpredictable Networks

AMP takes advantage of predictable nodes to reduce probing bandwidth. However, some networks may not contain many predictable nodes. These scenarios are more likely if one defines availability at the application level (is the overlay client running on the local host?) instead of at the OS level (is the machine powered up and connected to the network?). This is because application-level availability is a subset of, and more capricious than, OS-level uptime. Both the PlanetLab and the Microsoft traces measured OS-level availability.

To evaluate AMP's performance in extremely chaotic environments, we simulated a 1000 host network consisting entirely of nodes with random uptimes. Figure 3.6 depicts loss rates in this environment. After roughly two days, the availability predictors realized that their forecasts were inaccurate, and they set t_{rem} to zero. Sub-

sequently, loss rates using AMP were within 2% of those in the standard Mahajan scheme. The slight yet persistent penalty arose from the fact that even erratic nodes occasionally display regular behavior, providing false confidence to their availability predictors.

These results suggest that AMP is inappropriate for networks in which most nodes have highly capricious uptimes. In these environments, AMP should be completely disabled, and probe rates should be set using the standard pessimistic approaches.

3.3 Availability-aware Routing in Delay-tolerant Networks

In the traditional wired Internet, machines often have the luxury of persistent, high quality connections to their peers. In contrast, *delay-tolerant networks* (DTNs) [42] are composed of heterogeneous devices with vastly differing networking capabilities. Delay-tolerant networks must deliver messages in spite of intermittent device connectivity and differences in link bandwidth and latency that may span orders of magnitude. We provide two examples of DTNs later in this section.

3.3.1 Design

The simplest DTN routing algorithm does not rely on global routing state—at each hop, a message is sent along the first available link that provides forward progress. Given the heterogeneous link qualities and intermittent connectivities that characterize a DTN, such a naive scheme provides end-to-end delivery latencies that are far from optimal. Jain *et al* describe how to use *resource oracles* to decrease delivery times [59]. For example, they define a contact oracle which has perfect knowledge of link availabilities. Given two devices and an arbitrary point in the future, the contact oracle can output whether a link will exist between the two machines. Using our availability prediction techniques, we can approximate such an oracle and plug it into Jain's routing algorithm [59].

3.3.2 Evaluation

We evaluate our contact oracle by simulating the behavior of two DTNs. The first one is reminiscent of an example provided by Jain [59]. Imagine that a remote village without wired Internet access wishes to fetch web pages. Further suppose that the village is willing to tolerate asynchronous delivery latencies on the order of a day; such latencies are quite reasonable if, for example, the web pages are used to teach a class whose syllabus is known in advance. The remote village has a much larger sister city with a wired Internet connection, but this city is several hours away

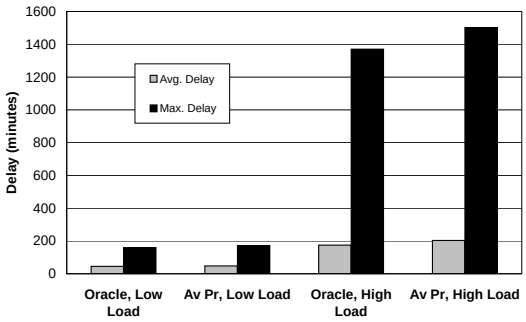
by ground transportation. Luckily, the buses that travel between the two cities can act as data mules, with outbound vehicles from the village carrying web requests and inbound vehicles carrying web data to be downloaded in the city. We assume that there are three round trips between the village and the city each day. The time required for each one way trip is chosen uniformly from the range [100 minutes, 140 minutes]. The first bus leaves the village at 8 AM, and the last bus is expected to return to the village at 8 PM. Each bus has a data capacity of 128 MB (equivalent to a small USB memory card), and each bus stays in network contact at the village or the city for five minutes. We assume that the bandwidth of the USB device is 1 Mbps.

The village has two additional means of communication. First, the village and the city are periodically connected by a satellite link. The satellite is close enough to both locations to form a direct link every six hours, and the link persists for ten minutes. The satellite bandwidth is 10 kilobits per second and the latency is 3 seconds. Second, the village has access to a slow dial-up modem which, for reasons of expense, is only accessible from 11 PM to 6 AM. Due to an unreliable telephone infrastructure, this link is offline for 10% of its ostensibly available period, with the unexpected disconnections scattered uniformly between 11 PM and 6 AM.

Figure 3.7 shows simulation results for the DTN described above. Web requests were 1KB on average and web responses were 10KB on average, as suggested by empirical studies of web traffic [94]. Predictors were trained on two weeks of synthetic availability data with a sampling granularity of 20 minutes. Web requests were then generated at randomly chosen times for 5 simulated days; each simulation terminated once all messages had been delivered. Routing was reactive, i.e., when a message arrived at a node to be forwarded, the node used the most recently observed availability data to calculate the route for that message.

In the low load scenario of Figure 3.7, the village and the city exchanged 200 messages a day. In other words, 200 web requests were sent to the city and 200 web fetches were sent to the village). Using our availability predictors led to average message latencies that were only 6.7% worse than those incurred by infallible oracles. The worse-case delay was only 7.5% worse.

In the high load scenario, the village and the city exchanged 1000 messages a day. Uptime mispredictions resulted in greater penalties in this scenario, since a single poor prediction could result in many messages being routed through an erratically online node. However, average message delays in our predictor system were still within 16% of those in a system with infallible contact oracles. Worse case delays



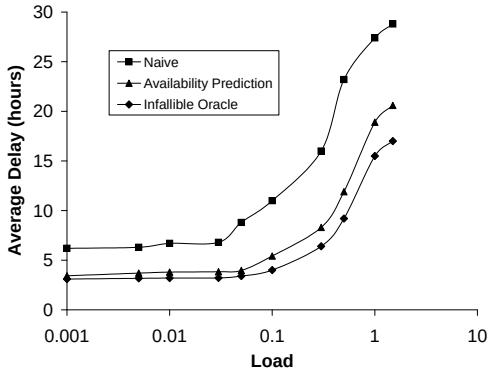
Using our availability predictors as contact oracles, delivery latencies are close to those generated by an infallible contact oracle. These results represent the outcomes of twenty simulation runs.

Figure 3.7: Message delays in the village DTN.

were within 9.6% of optimal.

Our second evaluation DTN represents a collaborative sensor node system. We imagine that each user in the system possesses a laptop, a desktop PC, and a set of trusted laptops and desktops belonging to friends. When a user moves to a new location and begins to work on her laptop, the laptop may "notice" something interesting about the surrounding environment, e.g., the availability of a new wireless access point. The user wants to share this information in an effort-free (and secure) method with her friends. Thus, her laptop acts as a store-and-forward node for the interesting piece of information. If the laptop is online when a friend's device is online, the laptop can directly transmit the information to the friendly device. Otherwise, the laptop tries to forward the data across a path of trusted machines.

In our simulation of this DTN, each PC's uptime was driven by a trace from the Microsoft corporate network. Each laptop's availability was driven by a trace from Kim *et al*'s wireless availability study [64]. We filtered out laptops and PCs which were not online at least once during the first 20% and the last 20% of their respective trace period. Laptop availability was sampled every 30 minutes and PC availability was sampled every hour. Each user had a trusted collection of ten laptops and ten PCs. Message sizes varied uniformly between 1KB and 20KB, and messages were randomly generated by each laptop using a Poisson distribution with a λ of 0.5 messages per online hour. When a laptop generated a message, it first delivered it to all trusted nodes which were also online. If the laptop predicted that an offline friend would come online before it left the network, it would wait to deliver the data directly. Otherwise, it would instruct one or more of its online buddies to forward the data to the remaining friendly devices. If, for a particular message, several destination devices shared a common next hop from the current machine,



In low to medium load situations, our availability prediction scheme is within 15% of optimal. The performance gap widens for loads greater than 0.6, with our scheme 40% worse than optimal under a load of 1.5. However, we still perform much better than a naive routing scheme which simply waits for the sender and the receiver to come online at the same time.

Figure 3.8: Message delays in a collaborative sensor DTN.

only one copy of the data was forwarded to the next hop. Laptops communicated with desktops using 11 Mbps wireless links, and desktops communicated with each other using 100 Mbps Ethernet connections. We assumed that a path (i.e., link) existed between two machines if they were online at the same time, and all routing was reactive.

Figure 3.8 shows simulation results for the collaborative sensor DTN. Each data point represents the average of 20 trials, and during each trial, the DTN was comprised of a random subset of 300 laptops and 300 PCs from the relevant availability traces. Each trial ran for 504 simulated hours, and predictors were pre-trained on 336 hours of data. We used Jain's definition of load [59], such that the load over the duration of a simulation was the sum of the traffic demand divided by the sum of the available transmission bandwidth during this time. Note that some of this bandwidth could lie idle if a node had nothing to transmit at a particular moment.

As in the village DTN, when loads became high, the delay differences between the availability prediction system and the infallible oracle system grew. This difference was at worst 40% for a load of 1.5, but was closer to 10-15% for more reasonable loads. Additionally, our availability prediction system always had better performance than a naive scheme in which nodes never forwarded data using intermediate nodes and always waited for the destination machine to come online. Thus, we believe that our availability predictors can provide a meaningful improvement in DTN performance.

DTNs are a fairly new concept, so there is no consensus on the best way to maintain distributed routing state. One could imagine that devices engage in an epidemic protocol to exchange local link tables. Each table contains a list of known

peers and the availability histories that these peers advertise. When two devices establish a connection, they exchange link tables and update local state with any new information received in the exchange. If one device has data to forward, it consults its local availability database and determines whether data should be routed through its current peer, or whether the transmission should be deferred until another device arrives.

3.4 Availability-aware Malware Analysis

Traditional analytic models of viral propagation [63, 85, 108] assume that machines are always online. This simplifying assumption makes mathematical modeling easier. However, as shown in Chapter II, real networks have non-trivial fluctuations in host availability. This means that at any moment during a viral outbreak, some infected machines will be offline. These hosts are effectively non-contagious, but they are also incapable of receiving a patch. Similarly, healthy offline machines cannot be infected, but they cannot be patched either, leaving them susceptible to infection when they come back online. The net result is a quantitative and qualitative impact on the viral dynamic, an impact which has been observed in the real world. For example, an empirical study of the CodeRed worm discovered strong diurnal patterns in behavior, with the number of active diseased hosts spiking at the start of the workday and ebbing as some people applied patches and many people turned off their workplace computers and left for their homes [77].

This section provides availability-sensitive models for malware propagation. In previous work [74], I introduced a simple availability-aware framework that I discuss briefly for pedagogical reasons. I then focus on a more sophisticated model introduced by Dagon *et al* [36]. Dagon's model represents uptime using a "diurnal shaping function" that is inferred from backscatter traffic [78]. Using my availability prediction techniques, I show how to improve the fidelity of Dagon's model. I also show how black hats can use the improved Dagon model to pick the most damaging time to launch a worm.

3.4.1 The Kephart-White Model

The classic Kephart-White model [63] uses a differential equation to describe malware propagation. It assumes a susceptible-infected-susceptible (SIS) environment—a machine enters the system in a healthy state, and it can catch and subsequently

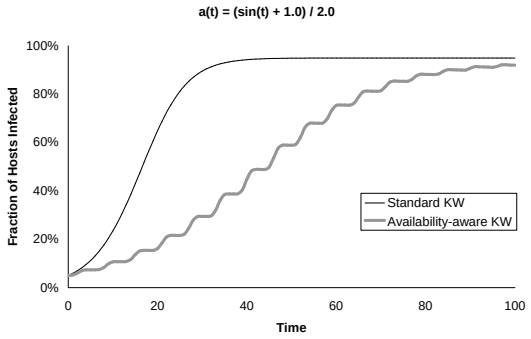


Figure 3.9: Comparing the two Kephart-White models ($\beta = 0.002, \langle k \rangle = 100, \delta = 0.01$)

be cured of the infection an infinite number of times [1]. The model assumes a homogeneous network topology in which all nodes have similar levels of connectivity or "out-degree." In such a network, the fraction I of infected nodes is given by:

$$\frac{dI}{dt} = \beta \langle k \rangle I(1 - I) - \delta I \tag{3.1}$$

where t is time, β is the viral birth rate along every edge from an infected node, $\langle k \rangle$ is the average connectivity of a node, and δ is the cure rate at each infected node. β, δ, and $\langle k \rangle$ are assumed to be constant.

The right-hand side of Equation 3.1 consists of a growth term ($\beta \langle k \rangle I(1 - I)$) and a loss term ($-\delta I$). The growth in infected hosts is governed by the probability that a sick node "sends infection" across a link (β), the number of neighbors that a sick node has ($\langle k \rangle$), the fraction of nodes that are sick (I), and the fraction of healthy nodes which are capable of being infected ($1 - I$). The loss of infected hosts is driven purely by the cure rate δ and the current number of infected machines I.

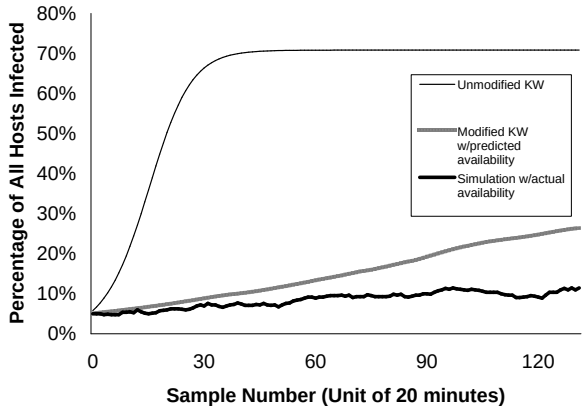
To capture the notion of fluctuating availability, we introduce a time-varying *availability fraction* denoted a. Like I, a can assume values between 0 and 1. At time t, $a(t)$ represents the fraction of machines that are currently online. Incorporating the availability fraction into the Kephart-White model results in the following equation:

$$\frac{dI}{dt} = \beta \langle k \rangle (Ia)[(1 - I)a] - \delta Ia. \tag{3.2}$$

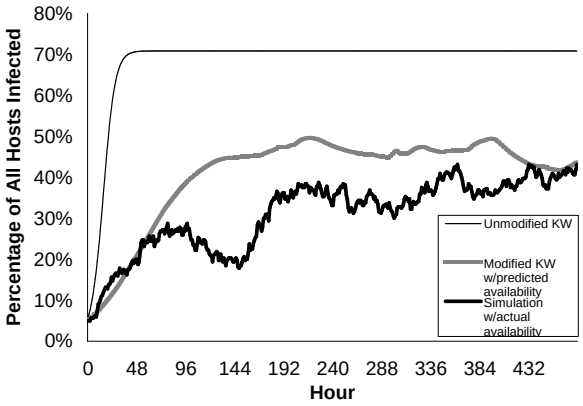
With respect to Equation 3.1, Equation 3.2 replaces the infected fraction I with Ia and the healthy fraction $(1 - I)$ with $(1 - I)a$. These new terms represent the fact that nodes must be online to transmit or receive the infection.

Figure 3.9 provides a simple illustration of how availability can affect the viral dynamic. This example uses a synthetic availability signal described by the periodic

[1] The simple SIS model has no conception of permanent immunity, so it only crudely models the deployment of remedies like software patches. The Dagon model described in the next section will capture the disinfection process.



(a) Overnet nodes are infrequently and erratically online. Ignoring their chronically low availability leads to dramatic overestimates of infection intensity.



(b) The PlanetLab network is prone to occasional, aperiodic, and massive dips in global connectivity. Such unpredictable events reduce the accuracy of the availability-aware KW model, but it is still much more accurate than the standard one.

Figure 3.10: Accuracy of viral models for the PlanetLab and Overnet systems (β=0.24,δ=0.07)

function `(sin(t) + 1) / 2`. The growth in the infected population exhibits accelerations and deaccelerations that are synchronous with the increases and decreases in host availability.

Figure 3.10 provides a more realistic depiction of the relationship between availability and malware propagation. In this figure, we depict the spread of a virus with β=0.24 in the Overnet network and the PlanetLab network. The graphs show the predictions of the standard Kephart-White model, a discrete-time simulation using actual availability data, and an availability-aware Kephart-White model using *predicted* availability with a 24 hour lookahead. In both networks, the availability-agnostic models produced far different outcomes than those observed in the availability-aware frameworks.

3.4.2 Dagon's Model

To analyze the impact of diurnal effects upon worm propagation, Dagon *et al* [36] introduced a modified version of the Kermack-McKendrick model [44]. Dagon's model incorporates a "diurnal shaping function" $\alpha(t)$ which represents the fraction of all hosts online at time t. Dagon estimates $\alpha(t)$ by observing changes in the volume of scan traffic collected by distributed sinkholes. The sinkhole data is split into 24 hour chunks, and each chunk is normalized so that the maximum traffic volume is one. The chunks are then averaged together to estimate $\alpha(t)$.

Like the Kephart-White model, Dagon's model uses differential equations to describe the infection process. However, Dagon's model explicitly represents the susceptible population S, and it adds a removed population R to model hosts which have received a patch and are permanently immune to future infections. Mathematically, we describe the population dynamics as follows:

$$\frac{dS}{dt} = -\beta(I\alpha)(S\alpha) \tag{3.3}$$

$$\frac{dI}{dt} = \beta(I\alpha)(S\alpha) - \gamma(I\alpha) \tag{3.4}$$

$$\frac{dR}{dt} = \gamma(I\alpha). \tag{3.5}$$

The new parameter γ is the rate at which infected nodes are patched, and α is the fraction of all hosts online. Like the Kephart-White model, Dagon's framework assumes homogeneous connectivity. However, the average node connectivity $\langle k \rangle$ is hidden within the viral birth rate β. An infected node can scan any computer on the Internet, so $\langle k \rangle$ is assumed to be 2^{32}, i.e., the total number of four byte IP addresses. The "true" scan rate is thus $\beta/2^{32}$. This model of infection dispersal corresponds the random uniform scans of worms like CodeRed [77].

3.4.3 The Optimal Timing Attack

Dagon describes how white hat researchers can estimate $\alpha(t)$ using distributed monitoring systems like DShield or NetBait [30]. Armed with estimates of $\alpha(t)$, they can forecast the progression of multiple worms and prioritize the allocation of defense resources using the projected virulences.

If black hats could calculate $\alpha(t)$, they could determine the release times that would result in the fastest infection spread. Dagon argues that estimating $\alpha(t)$

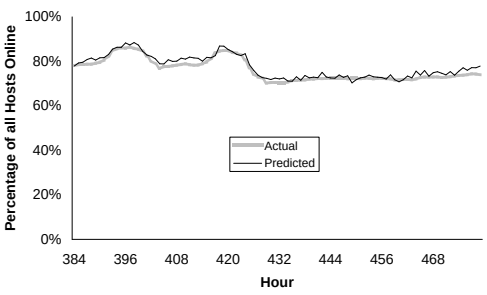
is difficult for botmasters because they would have to collect data on the scale of white hat monitoring projects. However, botmasters already have access to a large-scale distributed sensor network—the bots themselves! Each bot can track its own availability without generating probe traffic. Once the bot's availability history has reached a certain size, it can send the data back to the botmaster; since an availability history is just a bit string, it will easily fit into a single UDP packet. If a botmaster can subvert a small but representative portion of a large network, he can synthesize $\alpha(t)$ by averaging the availability histories of the bots. Armed with the historical $\alpha(t)$, the botmaster can apply linear prediction to forecast future values of $\alpha(t)$ and determine the best time to unleash the worm.

To estimate $\alpha(t)$, Dagon averages 24 hour segments of backscatter traffic and normalizes them so that the maximum possible availability is one. In contrast, our availability function is extrapolated from historical availability data, not induced by preexisting worm traffic. This difference has important ramifications. First, the attacker does not have to wait for a prior worm outbreak to generate enough backscatter to synthesize $\alpha(t)$. Instead, the attacker can calculate the function himself. Relying on backscatter traffic is also problematic because a worm's backscatter traffic will steadily decrease at some point due to patching activity. This means that Dagon's averaged $\alpha(t)$ will have a larger $\alpha(0)$ than $\alpha(24)$. To fix this malformation, Dagon uses an unspecified heuristic to ensure that $\alpha(t)$ is "not distorted much" [36]. Our approach removes the dependence between patching activity and availability measurement, eliminating the need for corrective heuristics. From the modeling perspective, we also believe that it is cleaner to have distinct parameters for host availability and patch activity.

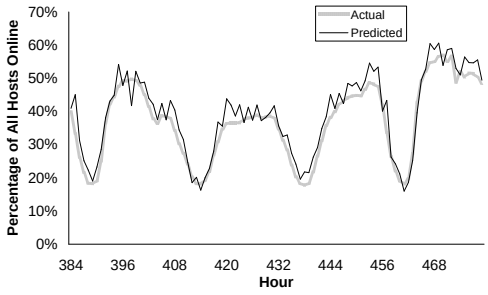
Note that our predictors do not assume a particular period length for availability activity. Dagon's assumption of a 24 hour period cannot capture the weekday/weekend dynamics observed in systems like the Microsoft network. These dynamics can be important with respect to choosing the best launch time for a worm, since host availability (and thus effective infection rates) will be lower on Saturday and Sunday.

3.4.4 Evaluation of Feasibility

To determine the feasibility of the timing attack, we sought answers for two questions. First, can a botmaster accurately predict global availability if he knows the availability of a smaller number of bots? Second, can a botmaster use these predictions to create more virulent worms?

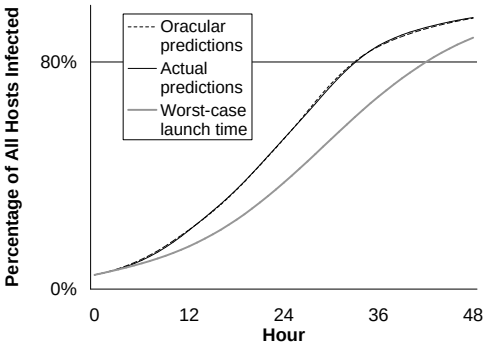


(a) Sampling 10% of all network hosts, a botmaster can predict global availability in the Microsoft dataset with a root mean square error of 1.7%.

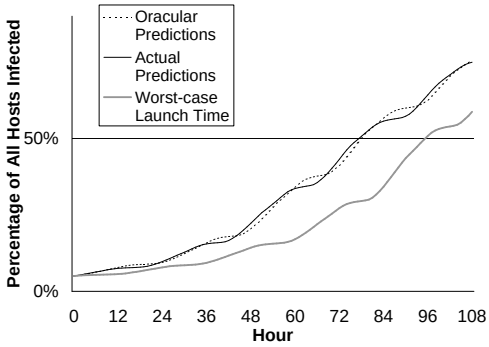


(b) Using the same technique, a botmaster could predict availability in the laptop trace with a RMSE of 3.8%. The error increases to 5.6% if only 3% of machines are randomly sampled.

Figure 3.11: Accuracy of predicted $\alpha(t)$



(a) For the Microsoft trace, our availability predictors select the same launch time as an oracular predictor would. The average speed to the infection target across all launch times in the window was 38 hours.



(b) In the laptop trace, our prediction scheme also results in the optimal speed to the target saturation level. The average time to the infection target was 83 hours.

Figure 3.12: Launch times vs. virulences (β=0.2)

In Figure 3.11(a), we compare actual global availability in the Microsoft trace to the predicted availability generated by a hypothetical botmaster. To generate the predicted signal, we randomly selected 10% of all hosts to be "subverted." We aggregated two weeks of their availability data to approximate overall availability during that time period. We then trained a linear predictor on the approximated $\alpha(t)$ and created an availability forecast for the next five days. As shown in Figure 3.11, the predicted signal is quite accurate, with a root mean square error of 1.7%. If the number of subverted machines is decreased to 3%, the RMSE only increases to 1.9%.

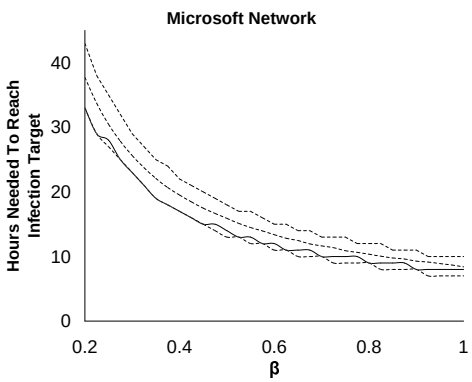
Figure 3.11(b) shows that our aggregation/prediction algorithm is slightly less accurate for a trace of laptop availability [64]. Rising and falling edges in the availability curve are forecasted tightly, but prediction errors of up to 6% occur during peak availability hours. During these periods, variance in overall availability is greatest, making accurate prediction more difficult.

Using a forecasted $\alpha(t)$ and Equation 3.4, a botmaster can simulate the spread of his worm and calculate the release time which provides maximum impact. For example, suppose that a botmaster in the Microsoft network wants to find the release time which results in the quickest infection of 80% of all hosts[2]. As shown in Figure 3.12(a), given the approximated $\alpha(t)$ from above and a five day launch window, our availability prediction scheme reaches the target infection level at the same time as a scheme that uses an oracle for prediction. In fact, our scheme selects the same launch time (hour 391) that an oracle would select. Looking at Figure 3.11, we see that hour 391 is the beginning of the first diurnal peak in the launch window. Launching the worm at this moment allows it to infect the surge of computers that are just entering the network. The worst-case launch time was hour 424, which represented the beginning of the weekend and its two day availability trough. Launching a worm at this moment resulted in a time to target infection level that was 9 hours later than optimal.

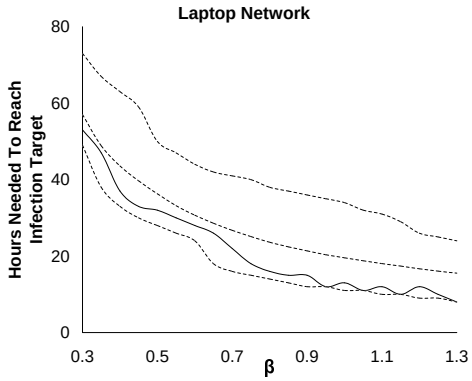
In Figure 3.12(b), we show the spread of a worm in the laptop environment. Since overall availability is lower in this network than in the Microsoft one, we chose a lower infection target of 50%. In the laptop environment, our predictor did not pick the same launch time as the oracular one, but both worms reached the target infection level with the same speed. The oracular attacker launched its worm at hour 419, which was the beginning of a diurnal surge in availability. The attacker which used our availability predictors launched its worm four hours later, at the beginning of a midday plateau in availability. The worst-case worm missed the optimal time to the infection goal by 18 hours; it launched during the availability decline of a Friday evening. In both networks, the average time to the infection target was roughly halfway between the optimal time and the worst-case one.

To further quantify the effectiveness of the timing attack, we performed simulations for a wide variety of β values. The results, shown in Figure 3.13, indicate that availability-guided worms almost always saturate quicker than randomly launched ones. However, the extent of the improvement and the β values which demonstrate the greatest gains are network-dependent. For example, in the laptop network (Figure 3.13(b)), the rising and trailing edges of availability are easily predicted, but the fluctuations during plateaus are difficult to forecast. Thus, our availability-guided scheme provides less benefit to the very slowest viruses, which are the most sensitive to plateau fluctuations. For β values larger than 0.8, our availability-guided scheme reduces saturation times by 32%-48% relative to the saturation speed of a randomly

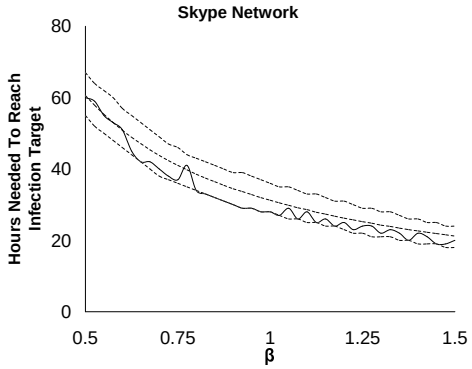
[2]To focus on the impact of launch times, we set γ to 0.



(a) For β between 0.2 and 0.45, the availability-guided scheme is near optimal.



(b) In the laptop network, availability-guided saturation times are within 25% of optimal for $\beta > 0.8$.



(c) Global availability in the Skype network is the most difficult to predict. Thus, our forecasting scheme provides less benefit.

The three dotted lines represent worst case, average case, and best case times to target saturation levels. The solid line represents the saturation time obtained using availability prediction. The target saturation level was 80% in the Microsoft network. The target was 50% in the other networks due to their lower aggregate availability.

Figure 3.13: Worm saturation speeds

launched worm.

Of the three networks, availability in the Skype system [50] is the most difficult to predict. This is because the system exhibited multiple deep, aperiodic drops in availability. Thus, our availability-guided scheme is less effective than in the other networks.

Interestingly, in all three networks, as β grows, the saturation time using availability prediction begins to show oscillatory behavior. For example, in the Skype network, the oscillation begins for β larger than 1, and in the Microsoft network it starts for β larger than 0.45. To explain this phenomenon, we note that as β increases, the average and best case saturation times converge. This means that even minor availability mispredictions can lead to the loss of the benefits provided by our

new scheme. The oscillations occur because only certain points in each availability signal are difficult to accurately predict. As β grows, the temporal positions of these points slide in phase relative to the beginning of the worm outbreak (when minor availability fluctuations make a difference) and the time at which the worm has almost completely saturated (when minor availability fluctuations are not very important). The phase movement causes corresponding fluctuations in the saturation times provided by our availability-guided scheme.

3.4.5 Is Availability Prediction Really Needed?

Our experiments validate the intuitive notion that worms are best launched at the beginning of an availability surge. One might conclude that since these surges take place at the same time during each work day, availability prediction is not necessary—botmasters should always launch their worms at 10 AM. However, this begs the question of which 10 AM the botmaster should choose. In the availability traces used above, all of the laptop hosts and most of the Microsoft hosts resided in a one time zone. In these networks, diurnal variations in availability are easily defined with respect to a single reference time zone. However, when networks span many time zones, phase relationships between each zone's diurnal patterns can lead to complicated interactions at the global level [36]. In these situations, an attacker cannot simply declare that his worm should be launched during peak diurnal availability, since this peak will be defined in different ways for different subsets of hosts. Using our zombie sampling/extrapolation technique, an attacker need not possess a deep understanding of these interactions. Instead, he merely needs to ensure that his zombies provide a representative sample of the entire network. We discuss the issue of sampling bias in more detail below.

Recent random scanning worms like Slammer [76] and Blaster [11] have spread as quickly as possible, with infected hosts aggressively probing the address space to find susceptible peers. However, our evaluation of the timing attack has focused on relatively slow worms with saturation times of a few tens of hours. A natural question is whether the timing attack is relevant, given the current popularity of extremely virulent worms.

There are two ways to answer this question. First, we note that launch time is still an important parameter for aggressive worms with near-immediate saturations. This is because the flash saturation level is a function of how many hosts are available at the initial release time, and instantaneous global availability can vary widely over time. For example, in the Microsoft network, global availability varies by roughly

15%, and in the laptop network, it varies by more than 30%. As we describe below, worms which spread aggressively are more likely to be detected quickly. So, for a worm which wishes to maximize damage over a short period of time, it is useful to be able to predict when aggregate network availability will be highest.

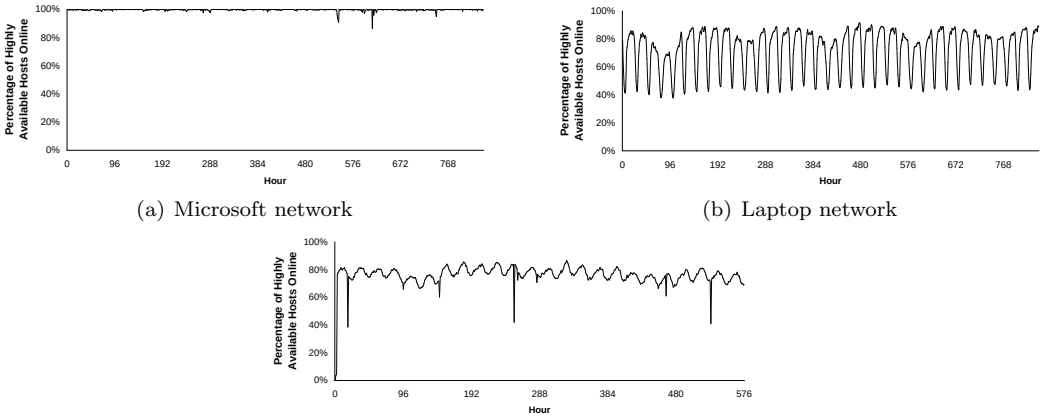
Interestingly, it may not be in a worm's best interest to spread too quickly. Aggressive address scanning is network-intensive. Hosts afflicted with worms like Slammer often generate orders of magnitude more traffic than they normally would. Many worm detection systems leverage this fact and use thresholding schemes to identify infected hosts, nominating those which send too much data or open too many TCP connections [2, 109, 110]. Thus, we believe that slower spreading worms will become more prevalent as attackers seek to avoid detection by thresholding schemes. Using the terminology of Agosta's work in adaptive thresholding [2], newer worms will attempt to hide in the "shadow" of the anomaly detector, sending more traffic than normal but less traffic than would cross an alert threshold. In this context, availability prediction allows an attacker to pick the best launch time for a worm, given the constraint of a smaller scanning budget that will not arouse suspicion.

3.4.6 Countermeasures

Launching an optimal timing attack requires three steps. First, the botmaster must recruit zombies. Second, the zombies must send their availability data back to the botmaster. Third, the botmaster must accurately extrapolate global availability trends from those of the zombies. Defenders can prevent the timing attack by disrupting one or more of these steps.

Interfering with the first two steps has been the goal of much ongoing research (e.g., [19, 34]). But despite these efforts to stop host subversion and disable preexisting zombie networks, botnets are becoming more common, not less [105].

If the botmaster only subverts machines whose uptime does not mirror that of the larger network, his predictions of global availability will be inaccurate. Thus, one could disrupt the timing attack by forcing the botmaster to sample a non-representative set of hosts. However, without *a priori* knowledge of the network traffic that belongs to the attacker, there is no way to selectively guide that traffic to a biased subset of nodes. A botmaster might unintentionally bias towards highly available machines if he recruits zombies over a time span that is much shorter than the natural period of global availability. For example, in the Microsoft trace, the natural period is seven days and half of machines are always online [37, 74]. If the botmaster recruits all of his zombies during a few hours on a Saturday or Sunday,



(a) Microsoft network

(b) Laptop network

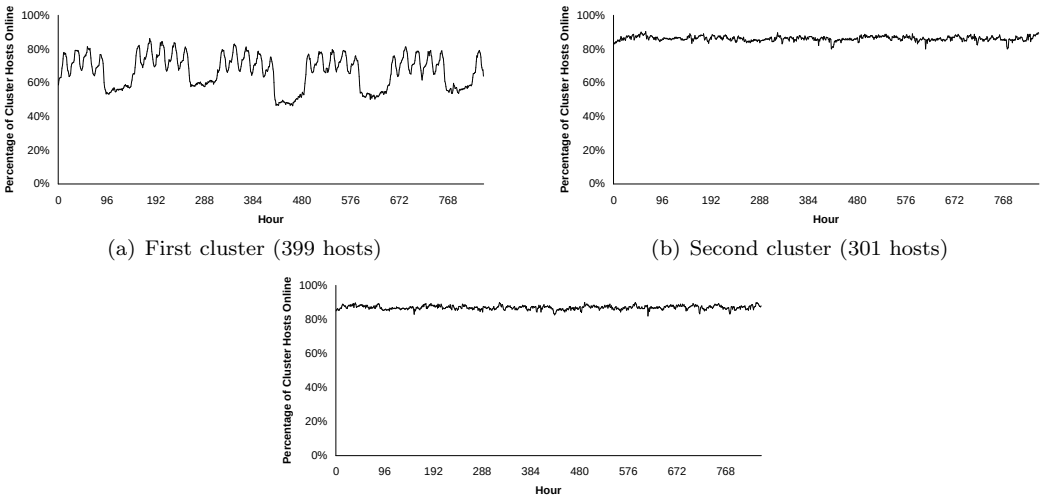
(c) Skype network

Figure 3.14: Aggregate uptime of the most available hosts

most of them will be highly available hosts without diurnal behavior. The aggregate availability signal of the zombies will be flat, preventing the botmaster from extrapolating the diurnal peaks and valleys seen in the real global signal. Thus, a successful timing attack in the Microsoft network requires a zombie recruitment phase of at least a few days.

In networks with aggregate diurnal patterns but low overall availability, even the most available hosts are likely to display diurnal fluctuations. Figure 3.14 depicts the aggregate uptime of the most available third of hosts in the Microsoft, laptop, and Skype networks. In the Microsoft system, the trend line is essentially flat, since the most available third of hosts are always online. Few such hosts exist in the laptop and Skype networks, so even if the botmaster's sampling is biased towards the most available nodes, he will still be able to extrapolate the diurnal patterns in global availability. The extrapolated signal may have a different offset from the x-axis than the real one. However, to launch an effective timing attack, only the shape and the phase of the extrapolated signal need to resemble that of the real signal. The former governs the fluctuations in the spreading rate of the worm, and the latter determines the real world "wall time" at which the fluctuations occur.

N-way programming and other forms of software heterogeneity can reduce the impact of worms by limiting the number of hosts with a particular vulnerability [9, 43, 61]. However, if the number of program versions is small, then random assignment of versions to hosts still leaves each version cluster vulnerable to timing attacks. This is because a random, unbiased sample of hosts will still mirror behavioral patterns displayed by the global network. One might hope to avoid this phenomenon through clever version assignment, such that the aggregate availability signal of each cluster



(a) First cluster (399 hosts)

(b) Second cluster (301 hosts)

(c) Third cluster (300 hosts)

Using simulated annealing, we can distribute N software versions to N host clusters such that N-1 of the clusters lack diurnal availability fluctuations (and thus are immune to timing attacks). In this example, the minimal cluster size was 300.

Figure 3.15: Availability-aware N-way programming assignment (N=3, 1000 host Microsoft network)

is flat or resembles random noise. Unfortunately, such an assignment is impossible if global availability displays regular patterns—these patterns imply the existence of hosts whose availability signals are not "canceled out" by those of hosts with inverted uptime behavior.

Many networks will contain enough availability heterogeneity to permit version allocations in which only one cluster displays regular availability patterns. Finding such an allocation is equivalent to solving a combinatorial optimization problem, a problem which can be solved using standard search methods. As a concrete example, suppose that we want to use simulated annealing [65] to assign N versions to the various hosts. First, we randomly assign each host to one of N clusters. We define the *energy* of each cluster as the autocorrelation [51] of its aggregate availability signal; the more random the signal, the lower its autocorrelation will be. We define the energy of the whole system as the average of the N-1 smallest cluster energies. During each annealing iteration, we change the version allocation of a few nodes in an effort to push the total system energy to zero—in other words, we try to make N-1 clusters have random availability signals. We allow one of the clusters to have high energy (i.e., high autocorrelation) because we know that at least one cluster must display the availability fluctuations found in the global network.

During each round of annealing, we allow two types of "moves": two clusters may

swap two nodes, or one cluster may transfer a node to another cluster. If a cluster shrinks to a certain minimum size, we only allow it to engage in swap moves. A swap or transfer is *good* only if it decreases total system energy. All good moves result in an update to the current version allocation. A bad move is applied with a probability that is proportional to the amount of time that the annealing process has run. At the beginning of the process, when the system is "hot," bad moves are often applied so that the system can escape local minima. As the process runs longer and the system "cools," bad moves are less frequently applied.

Figure 3.15 depicts the assignment of three program versions to a thousand hosts from the Microsoft network. As expected, two of the clusters have random aggregate availability, and one cluster displays diurnal availability that mirrors that of the global network.

3.5 Conclusions

Chapter II proved that predictable availability patterns exist in real networks. In this chapter, we proved that *exploitable* patterns exist. In doing so, we have shown that the proactive, defensive approach towards host churn is often needlessly pessimistic. Instead of relying on static assumptions about worst-case availability, distributed systems should use dynamic, reactive approaches which reason about the availability of individual hosts. To justify this claim, we examined several different operating environments and found a variety of ways to employ availability introspection.

Storage systems are the most popular type of peer-to-peer application. Part of their appeal lies in the fact that users can opt in and out of the system as they please, contributing to or withdrawing from the collective on a whim. Providing such freedom is an easy way to attract volunteers, but it results in great challenges for software architects. At first glance, it seems that aggressive replication is the only way to provide high data availability atop a churning storage layer. However, the frothing sea of storage hosts actually contains a surprising amount of order. Although peers are not officially coordinated by a central authority, many peers are unofficially synchronized through the diurnal activity cycles of their users. Non-diurnal machines may also exhibit regularity, e.g., in the duration of their online sessions. Although some machines may be totally unpredictable, *we need not act as if this is generally true.* By forecasting the availability of predictable machines, we can avoid placing data on hosts that are about to go offline. This reduces data regeneration bandwidth and also increases data availability. In a similar vein, we

can modulate the heartbeat probing rate using availability prediction, issuing many probes towards peers that are about to disconnect, and few probes towards hosts that will likely be online for a while. The result is a decrease in control traffic with minimal impact on disconnection detection.

Delay-tolerant networks route messages in the presence of extremely intermittent connectivity between forwarding hosts. Devising efficient routes is hard because poor decisions are often costly in terms of end-to-end delivery latency. A device which forwards data over a current pairwise connection may offload information to a peer that will be offline for a long time; however, holding the data for another peer is also dangerous, since that peer may never show up and the local device may go offline at any moment. Availability prediction allows DTNs to reason about future link connectivities and make better forwarding decisions. Experiments show that the resulting routes have much lower latencies than those constructed by a naive routing scheme.

As the Internet has grown in popularity, it has become a common transmission vector for malicious programs. Researchers have devised various mathematical models to describe malware propagation, but the vast majority of these models ignore fluctuating host availability. By including an explicit parameter for aggregate availability, we can improve the fidelity of contagion models. Unfortunately, these predictions can be used for good and for evil. By extrapolating global availability trends from those of a few bots, an adversary can select the launch time for a worm that results in the optimal saturation speed. Preventing the attack seems difficult since it is hard to stop zombie recruitment and there is no way to force the attacker to sample an unrepresentative set of hosts. Using availability-aware software heterogeneity, one can protect most (but not all) of the version clusters from timing attacks.

CHAPTER IV

Detecting Network Anomalies Using StrobeLight

As distributed applications grow in scale, they become more difficult to manage and understand. Complexity is an inevitable consequence of size, although core systems software can hide this complexity from higher-level applications. For example, many cooperative applications run atop a middleware overlay framework [91, 101]. This framework essentially provides a single function `route(msg, destId)` which allows peers to route data to an overlay address. Implementing this deceptively simple API requires constant work by the overlay to maintain routing table invariants. Since hosts can join and exit the forwarding substrate at will, each node must actively probe the availability of its routing peers. The situation is further complicated by the potential for network failures. Even if a host is online, it may only be *reachable* by a subset of all peers. Such non-transitive connectivity can affect availability probing (and thus routing state) in subtly insidious ways [45].

This example demonstrates two important points. First, it shows that availability is a function of both the host and the network—if a machine is cut off from the network, it does not matter whether it is powered up and running the distributed application. Second, this example shows how difficult it can be to maintain global system invariants in the presence of fluctuating availability.

Sustaining global notions of correctness would be easier if the system could reason about wide-area network problems and the availability of individual hosts. However, prior research has mainly focused on decentralized approaches in which each host only knows the availability of a select set of peers. Indeed, the large-scale availability traces described in the previous chapters [17, 21, 50, 64, 102] were not collected from real-time, in-system instrumentation, but from one-shot tools that were never intended to become permanent infrastructure.

The lack of a persistent infrastructure for availability monitoring is unfortunate because it could benefit a wide variety of systems. For example, distributed job allo-

cators [6] could use historical uptime data in concert with availability prediction to assign high priority tasks to machines that are likely to be online for the expected duration of the job. Distributed storage systems could also use a live feed of availability measurements to guide object placement and increase data availability [1].

Permanent monitoring tools do exist, but they often focus on measuring path characteristics, not individual host availability. Thus, they typically issue measurements to and from a small set of vantage points. For example, RON [5] and iPlane [69] track latency and loss rates between a set of topologically diverse end points, but these machines are assumed to be highly available and small in number; no mechanism is provided for testing individual host availability by subnet or some other aggregation unit. CoMon [84] provides uptime monitoring for individual PlanetLab hosts, but it does not scale to hundreds of thousands of machines. Furthermore, it requires modifications to end hosts, which may be difficult in non-academic settings where people are leery of installing new software.

In this chapter, we introduce StrobeLight, a tool for measuring availability in an enterprise setting containing hundreds of thousands of hosts. StrobeLight issues active probing sweeps at 30 second intervals, archiving the results for the benefit of other distributed services that might find them useful. StrobeLight also performs real-time analysis of availability trends, raising alarms for anomalies such as subnet unreachability, DNS failure, and IP hijacking. StrobeLight detects these anomalies using a new abstraction called an *availability fingerprint*. Under normal conditions, a subnet's fingerprint changes very slowly. Thus, StrobeLight raises an alarm when the similarity between consecutive fingerprints falls below a threshold. Experimental results show that our detection system is accurate and fast.

By using standard ICMP echo packets to test availability, StrobeLight avoids the need to install new software on end hosts or deploy new infrastructure within the routing core. By collecting data from a few centrally controlled vantage points, StrobeLight avoids the trust and complexity issues involved with distributed solutions while making it easy for other systems to access availability data. StrobeLight also demonstrates that frequent, active probing of a large host set is cheap and unobtrusive.

4.1 Design and Implementation

The design of our availability measurement system was guided by three principles. First, keep the system simple. Second, make the system unobtrusive. Third, collect fine-grained data.

Keep it simple. Our primary design principle was to keep everything simple, a philosophy reflected in many different ways. We wanted to avoid solutions which required new software to be installed on end hosts, an arduous task that is difficult to justify on a corporate-wide basis. Similarly, we hoped to avoid major modifications to our internal routing infrastructure. Large-scale decentralization of the probing infrastructure was not a primary concern. Although coordinated distributed monitoring has certain benefits, previous experience had taught us that the road to a bug-free distributed protocol is fraught with peril [22]. Thus, we thought hard about the costs and benefits of a coordinated peer-to-peer design, and ultimately rejected it. One motivating factor was our development of analysis techniques which tolerate temporal gaps in availability data (see Section 4.2.3). These techniques shifted the payoff curve between the better coverage and robustness of a distributed, coordinated solution and the reduced complexity of a centralized one.

Don't annoy the natives. We wanted a system that was unobtrusive—we did not want our measurement activity to disrupt normal network traffic or add significant load. We also required a straightforward mechanism to turn off measurement activity in specific parts of the network. The latter was important because previous experience had taught us that at some point, our new network infrastructure would break someone else's experiment or interact with other components in unexpected ways. When such scenarios arose, we wanted the capability to quickly remove the friction point.

Collect high-resolution data. We wanted our tool to collect per-host availability statistics at a fine temporal granularity. This would allow us to validate previous empirical studies which used coarser data sets [21, 74]. It would also make the service more useful for anomaly detection, since disruptions like IP hijacking may only last for a few minutes [90].

These design considerations led to several "non-goals" for our system.

Infinite scalability is overkill. Our solution only needed to scale to the size of an enterprise network containing hundreds of thousands of hosts. Building a measurement system to cover an arbitrary number of hosts in an arbitrary number of administrative domains would have been extremely challenging. For example, active availability probing from foreign domains might trigger intrusion detection systems. Organizations might also be reluctant to provide outside access to DNS servers and other infrastructure useful for identifying "live" end hosts.

Complete address disambiguation is difficult. Another barrier to performing arbitrary-scale, cross-domain host monitoring is the widespread use of NATs,

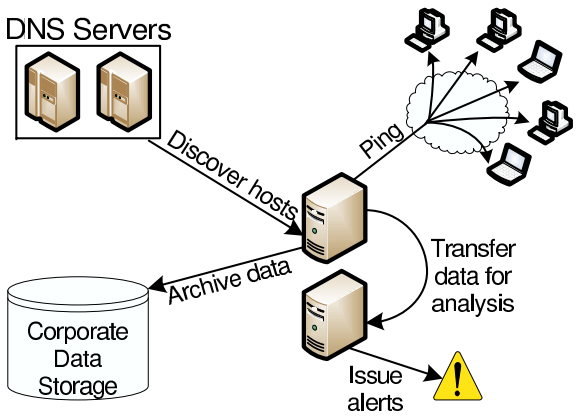


Figure 4.1: StrobeLight Architecture

firewalls, and DHCP. These technologies can create arbitrary bindings between hosts and IP addresses, and prevent some machines from being seen by external parties. Devising a comprehensive monitoring system that can pierce this heterogeneous cloud of addressing policies is an important research topic. However, this goal was beyond the scope of our project. By focusing on enterprise-level solutions, we hoped to avoid many of the issues mentioned above; NATs were relatively rare in our corporate environment, and we could configure our firewalls to trust packets generated by our new monitoring system.

4.1.1 The Winning Design: StrobeLight

As shown in Figure 4.1, we eventually chose a centralized architecture in which a single server would measure availability throughout our entire network. To determine which IP addresses to test, the server would download hostname/IP mappings from corporate DNS servers. It would then test host availability using standard ping probes issued at intervals of 30 seconds. Recent probe results would be transferred to an analysis server for real-time anomaly detection, and longitudinal data would be archived in the corporation's standard distributed data store.

This design, which we named StrobeLight, was extremely attractive from the implementation and deployment perspectives. No new code would have to be pushed to end hosts or internal routers, and the only additional hardware required would be the probing server and the analysis engine. We also expected the probing process to have a light footprint. The total volume of request/response traffic would be very small compared to the overall traffic level in the corporate network. Furthermore, we would not have to deal with control or synchronization issues that might arise in a more decentralized design. Our main concerns involved performance and fault

tolerance. We feared that a single server might be overloaded by sending probes for hundreds of thousands of machines every 30 seconds. A centralized probing design also had obvious ramifications for fault robustness. Despite these weaknesses, we committed to the single-server design due to its relative ease of implementation, and we pledged to revisit the design if we encountered undue difficulties after deployment.

4.1.2 Implementation and Deployment

The core probing infrastructure was deployed first. The pinging daemon, consisting of 2200 lines of C++ code, runs on a standard desktop PC with a 3.2 GHz CPU, 2 GB of RAM, and a gigabit Ethernet card; this machine resides within a corporate subnet in Redmond, WA. At boot time, the daemon reads an exclusion file which specifies the set of IP prefixes that should never be pinged. This file allows us to selectively exclude parts of the network from our probing sweeps. To determine which IP addresses to ping, the daemon downloads zone files from Microsoft's DNS servers at 2:10 AM each day. At any given moment, these zone files contain entries for over 150,000 IP addresses scattered throughout the world. This set of addresses evolves over time due to the introduction of new hosts and the decommissioning of old ones.

Due to these factors, an IP address may not appear in every DNS snapshot. Since StrobeLight only probes the addresses mentioned in the zone data, an IP may have gaps in its availability history. To deal with these gaps, StrobeLight describes the availability of an IP address as online, offline, or unknown. The first two categories result from the outcome of a ping probe, whereas the third is assigned to an IP which was not probed at a particular time.

Once the probing daemon had produced a sizable archive of availability data, we were able to test the offline analysis engine. This engine, totaling about 5,000 lines of C++ code, provides a set of low-level classes to represent per-host availability. It also defines a high-level query interface for use by data mining programs. We used this interface to generate the results in Sections 4.2 and 4.3. Importantly, the interface defines a subnet of size N as a set of N consecutive and *allocated* IP addresses; the querier chooses the starting address, N, and the time period over which "allocated" is defined. An address is considered allocated during a given time period if it appeared in a zone file at least once during that period. In practice, we often set N to a small number like 256 and investigate the subnets contained within a Class A or B prefix.

The anomaly detection engine has been written and it is running on a separate machine from the prober. However, the engine is still the target of active development and it does not yet receive live feeds from the probing daemon. We evaluate it

in Section 4.4 using time-delayed availability measurements from the probing server. As we discuss in that section, we eventually plan to deploy a handful of StrobeLight systems in topologically diverse parts of the Internet. However, individual systems will not need to coordinate since anomalies are defined with respect to local perspectives.

4.1.3 Operational Experiences

The probing server has run with few interruptions for almost two years. With a notable exception described in Section 4.4.3, the server has not struggled with the network load generated by the ping sweeps. We currently spread each sweep across 25 seconds to avoid load spikes on our shared network infrastructure, but brief "full throttle" experiments show that our current prober can scan 270,000 hosts in 7.9 seconds (roughly 35,000 hosts a second).

In general, our ping traffic has not bothered the other members of our network. At the beginning of the deployment, we would occasionally receive emails from the network support staff when they unveiled a new intrusion detection system and determined that the prober was infected with an IP-scanning virus; these incidents became rarer after we explained that StrobeLight was a piece of permanent infrastructure. We also received a complaint from another research group who claimed that our pings were causing problems for their wireless devices. After generating the appropriate exclusion file and restarting the daemon, we received no more complaints from them.

4.2 Long-term Availability Trends

In this section, we investigate the long-term availability trends in our corporate environment, both within wired and wireless subnets. We restrict our analysis to IP addresses with an unknown fraction of less than 5%. During the time period examined below, this included 138,801 wired IPs and 11,670 wireless IPs. In our corporate environment, the DHCP lease time is 20 days for wired machines and 3 hours for wireless ones. Thus, a wireless address is likely to be bound to multiple machines over the course of the day. Although we often refer to "hosts" and "IP addresses" interchangeably, the true unit of uniqueness is an address, not a host.

4.2.1 Global Trends

Figure 4.2 depicts aggregate availability fluctuations from October 21 to November 21 of 2005. The bulk of Microsoft's machines reside in the American west coast, so

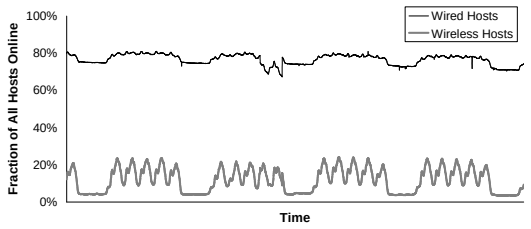
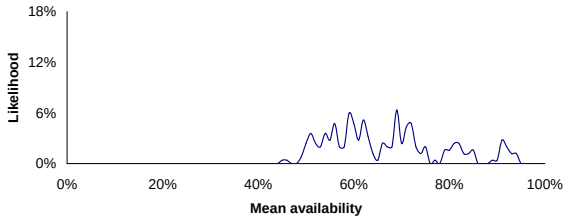
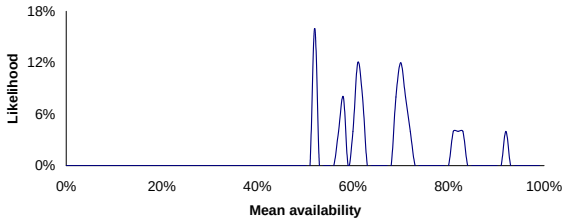


Figure 4.2: Global availability (10/21/2005 to 11/21/2005)



(a) 256 hosts per subnet



(b) 2048 hosts per subnet

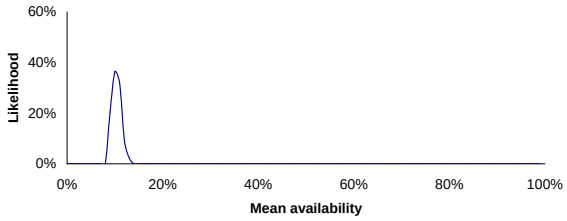
Figure 4.3: PDF for mean subnet availability (wired)

both the wired and wireless networks show large-scale diurnal trends aligned with the work day in this time zone. However, during these large-scale surges and declines in availability, there are regular, smaller-scale peaks and valleys. These additional periodic cycles are driven by phase-shifted diurnal behavior amongst Microsoft hosts in Europe and the Middle East.

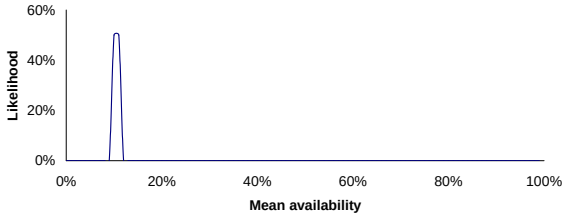
Comparing the two curves in Figure 4.2, we see that wireless IP addresses are much less likely to be associated with online hosts. However, the wireless network demonstrates stronger diurnal trends than the wired network. We investigate this issue further in Section 4.2.3.

4.2.2 Subnet-level Trends

We define the mean availability of a subnet as its average fraction of online hosts. Figure 4.3 depicts the distribution of mean subnet availability in the wired network for subnets of size 256 and 2048. In both cases, we see that mean subnet availability is always higher than 40%. Increasing the subnet size causes probability mass to coalesce around several regions of mean availability. This is a discretization arti-



(a) 256 hosts per subnet



(b) 2048 hosts per subnet

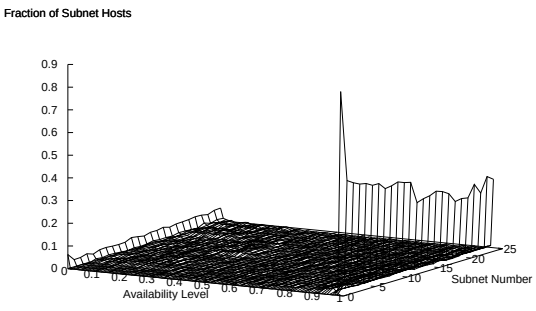
Figure 4.4: PDF for mean subnet availability (wireless)

fact, since increasing the subnet size without increasing the total number of hosts results in fewer subnets to examine and less smoothness in the resultant availability distribution.

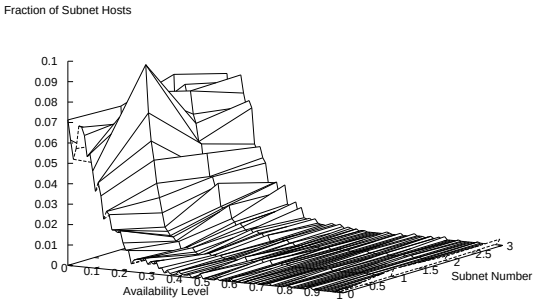
As expected, Figure 4.4 shows that wireless subnets have much lower mean availability. But in contrast to the wired network, increasing the subnet size from 256 hosts to 2048 results in an almost identical availability distribution. The relative lack of discretization artifacts is due to the greater homogeneity of wireless host availability. Figure 4.5 depicts the distribution of per-host uptime fractions within each subnet. Each wired subnet has a skewed bimodal distribution, with a plurality of hosts having very high uptime and a smaller fraction having very low uptime. However, in every wired subnet, roughly 50% of the probability mass is spread across the "plateau" between the two modes. In contrast, the wireless subnets look more unimodal, with the majority of hosts having very low availability and much less probability mass sheared away from the mode. In the context of Figures 4.3 and 4.4, this means that changing the subnet size in the wired network should result in more "jitter" in mean subnet availability.

4.2.3 The Availability of Individual Hosts

To understand the lower-level dynamics driving aggregate availability, we modified the taxonomy described in Section 2.2.1. The original taxonomy used Fourier transforms to extract patterns from availability signals. However, the standard algorithms for Fourier decomposition assume that signals are sampled at a uniform rate and that no samples are missing. In our data set, the assumption of a uni-



(a) Wired subnets (2048 hosts per subnet)



(b) Wireless subnets (2048 hosts per subnet)

Each curve on the "availability level" axis is a pdf for per-host uptime fractions in a particular subnet. In each graph the pdfs are sorted by standard deviation, with higher subnet numbers indicating larger standard deviations. The trends depicted in each graph are insensitive to subnet size.

Figure 4.5: Per-host availability within a subnet

form sampling rate was almost always true, since the vast majority of probe sweeps were separated by 30 second intervals. Unfortunately, missing samples were fairly common for two reasons. First, our network used DHCP to assign IP addresses to physical machines. When an address was dormant (i.e., unassigned), it did not show up in our zone files, meaning that we did not collect availability data for it during the dormant period. Second, the DNS servers occasionally failed, or misbehaved and returned extremely small zone files. Both of these phenomenon introduce brief probing gaps for many hosts.

To deal with missing samples, we replaced the Fourier analyses with two entropy-based techniques. To determine whether an availability signal contained diurnal patterns, we adapted Cincotta's method for period detection in irregularly sampled time

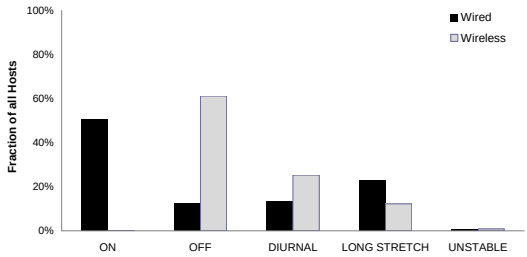


Figure 4.6: Availability taxonomy

series [31]. Let $a_t \in \{0, 1\}$ be the value of an availability signal at time t. Given a hypothetical period τ, we calculate the phase of each a_t as $\phi_t = \frac{t}{\tau} - nearestInteger(\frac{t}{\tau})$; note that $\phi_t \in [-0.5, 0.5]$. We can interpret each (ϕ_t, a_t) pair as a coordinate in $\phi \times a$ space. If the hypothesized period τ is close to the signal's actual period (or a harmonic of it), the (ϕ_t, a_t) points will cluster in the coordinate space. This means that if we divide the coordinate space into bins, the resultant bin distribution will have low entropy. If the hypothesized period is not the signal's true period, points will be scattered throughout the $\phi_t \times a_t$ space and the bin distribution will have high entropy.

To determine whether an availability signal contains diurnal patterns, we check whether the entropy for a τ of 24 hours is less than the entropy for a τ of 23 hours. Availability signals with complex diurnal patterns may have entropy dips in other places, but finding one for a τ of 24 is sufficient for our purposes.

To determine whether an availability signal contains long-stretch behavior, we use an approximate entropy test [87]. Suppose that we have an arbitrary window of k consecutive samples from the signal. We define `ApEn(k)` as the additional information conveyed by the last sample in the window, given that we already know that previous $k-1$ samples. Low values of `ApEn(k)` indicate regularity in the underlying signal. In particular, if we know that a host is not always-on, always-off, or diurnal, but it still has a low `ApEn(k)`, it is likely that the uptime regularity is driven by long-stretch behavior.

The choice of window size k is driven by the time scale over which "long stretch" is defined; k should be small enough that a stretch contains several windows, but not so small that `ApEn(k)` measures the incidence of small k-grams that are actually pieces of larger, more complex availability patterns. In the results presented below, we used a k of 8 and sampled our availability trace in steps of 15 minutes. This meant that we looked for long-stretch behavior at a time scale of roughly two hours.

Figure 4.6 depicts the availability taxonomy for the wired and wireless networks [1].

[1]We defined hosts as long-stretch if their availability signal had an `ApEn(8)` of less than 0.16. This cutoff was

We found that roughly half of the wired hosts were always online. This result is congruent with smaller-scale observations of the Microsoft network which used an hourly sampling period [21, 74]. Indeed, the fact that the always-on fraction is the same at a finer sampling granularity implies that the natural time scale for availability fluctuation in wired corporate environments is hours, not minutes. This claim is further validated by the fact that almost none of the wired hosts had unstable availability. In other words, if a host was not always-on, always-off, or diurnal, then it at least had availability that was stable over the course of one or two hours.

The wireless network was dominated by always-off machines, which comprised 61% of all hosts. The wireless network had almost twice as many diurnal machines as the wired network (25% versus 13% respectively) but almost half as many long-stretch hosts (12% versus 23%). These trends were unsurprising. In contrast to desktop machines that are always left "plugged in," wireless devices are more likely to suffer disconnections at arbitrary moments, and they are prone to being taken home at the end of the day (and thereby removed from the physical proximity of a corporate access point). Thus, we would expect wireless connectivity to show stronger diurnal patterns and less long-stretch behavior.

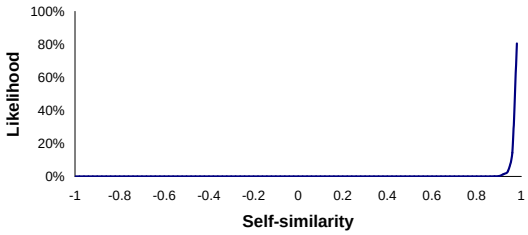
4.3 Short-term Availability Mapping

In the previous section, we investigated aggregate availability trends over a five week window. However, many network anomalies occur over a much smaller time scale. For example, an IP hijacking attack might only last for several minutes [90], and BGP misconfigurations can be just as transient [27].

Both types of anomaly change the mapping between IP addresses and physical hosts. In a hijacking attack, an entire range of IPs is bound to a different set of physical machines; similarly, a misconfigured router can cause arbitrary desynchronizations. Active availability probing can detect such problems if three conditions are true. First, the probing interval must be less than the duration of the desynchronization episode, lest the anomaly escape undetected between probing sweeps. Second, in the absence of anomalies, a subnet's availability "fingerprint" must be stable across multiple consecutive probing periods. This gives us confidence that when the fingerprint changes, an actual problem has arisen. Third, at any given moment, the availability fingerprint for each subnet should be globally unique. This allows us to detect routing problems in which two subnets have their IP bindings swapped.

---

determined by hand, but the results are not very sensitive to the exact value.



Subnet self-similarity between successive probing sweeps is very high. The graph depicts results for wired subnets of size 256, but the outcome is insensitive to subnet size. The results are extremely similar for wireless subnets.

Figure 4.7: PDF for self-similarity of delta fingerprints (15 minute probe interval)

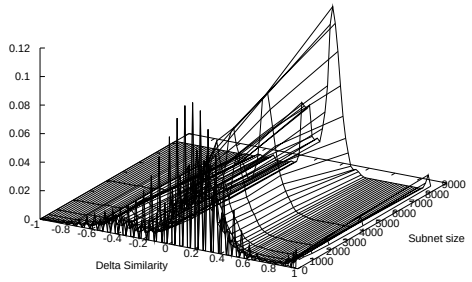


Figure 4.8: PDF for instantaneous cross-subnet similarity (15 minute probe interval)

With these desired characteristics in mind, we can provide a formal definition of a fingerprinting system. Given a specific subnet and a time window of interest, a fingerprinting algorithm examines per-host availability trends during that window and produces a binary string that is a function of those trends. A fingerprinting system also defines a distance metric which determines the similarity of two fingerprints. To detect an anomaly in a subnet, we maintain a time series of its fingerprints and raise an alarm if the most recent fingerprint is too dissimilar from the previous one.

In the remainder of this section, we provide a concrete description of a fingerprinting system and evaluate its performance on trace data collected by StrobeLight. We focus on basic issues such as how a subnet's fingerprint evolves over time and the accuracy with which we can distinguish two subnets based solely on their fingerprints. We present more applied results in Section 4.4, where we describe how minimal extensions to our StrobeLight prototype can provide enterprise-level anomaly detection.

4.3.1 Delta Fingerprints

During a single probe sweep, we test the availability of each known host in our network. Given a subnet of size s, we represent its probe results as an s-bit vector where a particular bit is 1 if the corresponding host was online and 0 if the host was offline or unknown [2]. We call such a vector an instantaneous or delta fingerprint because it represents a snapshot of subnet availability at a specific time.

A natural distance metric for two delta fingerprints is the number of bit positions with equal values. Thus, we define the similarity of two fingerprints as the number of equivalent bit positions divided by s and normalized to the range $[-1, 1]$. For example, if two fingerprints match in half of their bit positions, they will have a similarity of 0. If they match in all positions or no positions, they will have a similarity of 1 or -1 respectively.

Given the availability probing period ρ, we define a subnet's self-similarity as the expected similarity of its fingerprints at time t and time $t + \rho$. Figure 4.7 depicts the pdf for self-similarity in the wired network with a ρ of 15 minutes. As shown in Section 4.2.3, the natural time scale of availability fluctuation in the wired network is hours, not minutes. Thus, with a 15 minute sampling granularity, delta fingerprints are very stable across two consecutive snapshots, with 95% of all fingerprint pairs exhibiting similarities of 0.96 or greater. Decreasing ρ results in even greater stability, which is possible since StrobeLight has a 30 second probing granularity.

The delta similarity of two different subnets at time t is simply the similarity of their fingerprints at t. Figure 4.8 depicts the pdf for cross-subnet similarity as a function of subnet size. As the subnet size grows, probability mass shifts towards the center of the similarity spectrum. However, even for subnets as small as 32 hosts, less than 2% of all subnet pairs have similarities greater than 0.8. The reason is that the various availability patterns described in Section 4.2.3 are randomly scattered throughout each subnet. For example, even though most subnets have a large set of always-on hosts, these hosts are randomly positioned throughout each subnet's fingerprint vector. Thus, two vectors are unlikely to have high correlations in all bit positions, and each fingerprint is likely to be globally unique.

The tiny peaks along the right side of Figure 4.8 indicate a small probability that at any given moment, two subnets have completely equivalent fingerprints. To understand the origin of these peaks, we plotted cross-subnet similarity as a function of time. Figure 4.9 indicates a large spike in fingerprint similarity during the middle of the trace period. As Figure 4.10 shows, this spike was synchronous with a dramatic

[2] Remember that a host is not probed if it is not mentioned in the current DNS mapping.

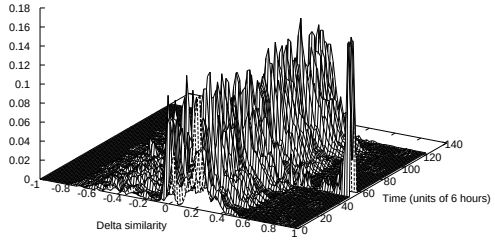
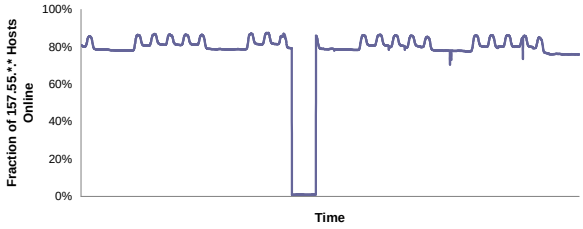
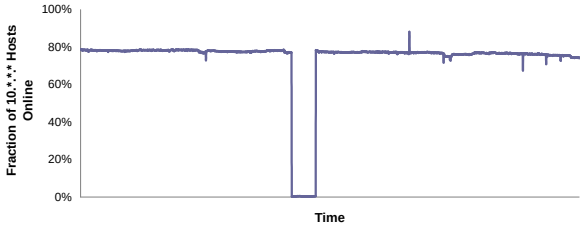


Figure 4.9: Temporal evolution of cross-subnet delta similarity (15 minute probe interval)



(a) Host availability in 157.55.*.*



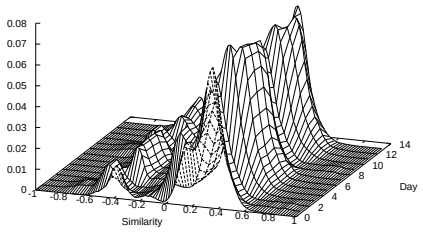
(b) Host availability in 10.*.*.*

Network anomalies during November 3 and 4 of 2005 caused the spike in fingerprint similarity seen in Figure 4.9.

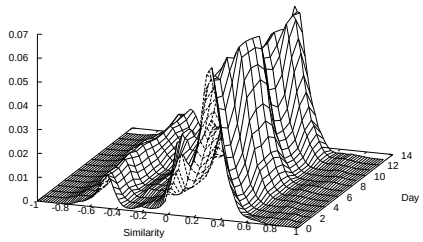
Figure 4.10: Punctuated availability disruptions

availability drop in several IP blocks during November 3 and 4 of 2005. When these blocks went offline, their fingerprint vectors transitioned to an "all-zeros" state, leading to an immediate increase in cross-subnet similarity.

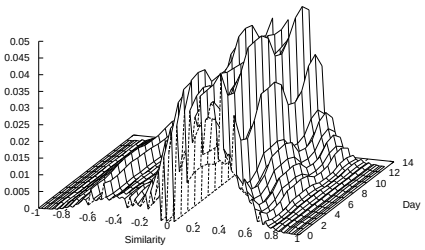
Once the anomaly terminated, the similarity distribution returned to a steady state in which all fingerprints were distinguishable. Thus, during the whole trace period, the global uniqueness property was only violated during the severe network disturbance.



(a) 256 hosts per subnet, 24 hour window, 32-bit float per host



(b) 256 hosts per subnet, 24 hour window, 1-bit float per host



(c) 32 hosts per subnet, 24 hour window, 1-bit float per host

Figure 4.11: Cross-subnet similarity for wired and wireless subnets (24 hour window)

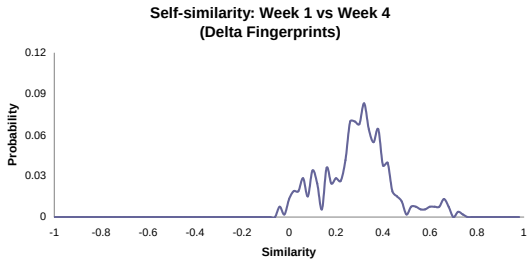
4.3.2 Fingerprinting Over Larger Windows

As currently described, a fingerprint is a bit vector representing the instantaneous availability of a set of hosts. In this section, we briefly describe how to extend our fingerprints to cover longer observation periods.

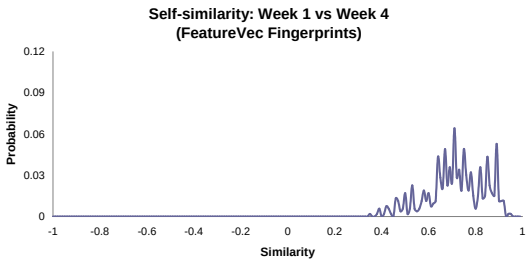
Cross-subnet Similarity

To create a fingerprint which covers a longer time window, we can associate each host with a floating point number instead of a single bit. Each float represents the mean availability of a host during the time period of interest. To compute the similarity between two floating point fingerprints, we examine each pair of corresponding floats and calculate the absolute magnitude of their difference. We sum these absolute magnitudes, divide by the subnet size, and then normalize the result to the range $[-1, 1]$.

Figure 4.11(a) shows the temporal evolution of cross-subnet similarity using a day-long window; the subnet size was 256 hosts and each host was associated with a



(a) If we represent host uptime as a 32-bit floating point availability fraction, subnet self-correlation across weeks is mildly positive.



(b) Self-correlation is higher if we represent host availability using a 2-bit enumeration type {ALWAYS-ON, ALWAYS-OFF, DIURNAL, OTHER} and check for behavioral equivalence amongst corresponding fingerprint entries.

Figure 4.12: Wired subnet self-similarity using week-long windows

32-bit floating point number. Comparing Figure 4.11(a) to Figure 4.8, we see that lengthening the fingerprint window does not change the fundamental distribution of subnet similarity. Most subnets are weakly similar or weakly dissimilar, but almost none are very similar or very dissimilar.

Figure 4.11(b) depicts cross-subnet similarity using a 24 hour window and "1-bit floats." In this scenario, a fingerprint contained a single bit for each host; the bit was 1 if the host was majority-online during the window and 0 if it was majority-offline. Comparing Figure 4.11(a) to 4.11(b), we see that using these truncated floats has little impact on the similarity distribution. Even if we decrease the subnet size to 32 hosts, Figure 4.11(c) shows that 1-bit floats provide enough resolution to keep the likelihood of perfect cross-subnet similarity well below 1%.

Using 1-bit floats, very little storage space is needed to maintain longitudinal fingerprint databases. For example, suppose that one wishes to store fingerprints for a network containing 250,000 hosts. Using 1-bit floats, an individual snapshot would consume 250,000 bits (31.3 KB), meaning that a single 60 GB disk could store over two million snapshots.

Self-similarity

Most subnets exhibit diurnal uptime behavior. However, the true period of their availability is a week, not a day, since availability during the weekend lacks diurnal fluctuation and is depressed relative to that of the work week. Thus, if we examine subnet self-similarity using a day-long window, there are discontinuities during the transitions into and out of the weekend. However, one might expect self-similarity to be high using a week-long window, since this window size would precisely capture a full cycle of the seven day availability pattern.

Figure 4.12(a) shows the distribution of wired subnet self-similarity between the first and fourth weeks of our observation period. Although self-similarity was almost always positive, the correlation was unexpectedly weak, with the bulk of the probability mass residing between 0.0 and 0.5. This surprised us, since we had predicted that a host's availability fraction would not change much across weeks. Confronted with these results, we generated a new hypothesis, predicting that a host's availability *class* would vary less than its availability *fraction*. For example, the uptime fraction of a long-stretch host might vary between weeks, but its availability would be unlikely to transition from long-stretch behavior to (say) diurnal behavior.

To test this hypothesis, we devised a new type of fingerprint called a feature vector fingerprint. Instead of associating each host with a floating point availability fraction, we gave each host a 2-bit identifier representing whether it was always-on, always-off, diurnal, or "other" (either long-stretch or unstable). We defined the similarity between two feature vectors as the number of corresponding positions with equivalent feature identifiers. As before, we divided this number by the vector size and normalized it to the range $[-1, 1]$.

Figure 4.12(b) confirmed our hypothesis that, at the granularity of individual hosts, availability classes are more stable than availability fractions. However, subnet self-similarity was still lower than expected given the observed stability of weekly availability cycles at the subnet level. This topic remains an important area for future research.

4.4 Detecting IP Hijacking

The Internet is composed of individual administrative domains called autonomous systems (ASes). The Border Gateway Protocol (BGP) stitches together these independent domains to form a global routing system [49]. Packets follow intra-domain routing rules until they hit an inter-AS border, at which point BGP data determines the next AS that will be traversed.

As currently described, StrobeLight detects intra-AS anomalies. For example, in Section 4.3.1, we showed how StrobeLight discovered the unreachability of several large subnets from within our corporate network. In this section, we describe how to detect BGP anomalies which affect subnet visibility from the perspective of external ASes. To detect such anomalies, we must deploy StrobeLight servers *outside* of the local domain. We describe the architecture for such a system and provide a preliminary evaluation using our current StrobeLight prototype. We focus on detecting IP hijacking attacks, a potentially devastating form of routing disruption.

4.4.1 Overview of IP Hijacking

An AS declares ownership of an IP prefix through a BGP announcement. This announcement is recursively propagated to neighboring ASes, allowing each domain to determine the AS chain which must be traversed to reach a particular Internet address. BGP updates are also generated when parts of a path fail or are restored. Since BGP does not authenticate routing updates, an adversary can fraudulently declare ownership of someone else's IP prefix and convince routers to deliver that prefix's packets to attacker-controlled machines. An attacker can also hijack a prefix by claiming to have an attractively short route to that prefix.

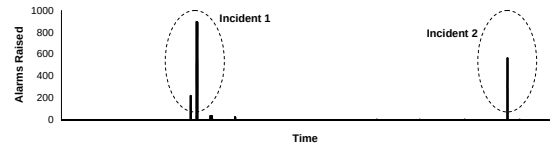
Zheng *et al* describe three basic types of hijacking attack [115]. In a *blackhole attack*, the hijacker simply drops every packet that he illegitimately receives. In an *imposture attack*, the hijacker responds to incoming traffic, trying to imitate the behavior of the real subnet. In an *interception attack*, the hijacker forwards data to its real destination, but he may record packets or otherwise investigate their contents.

Due to vagaries in the BGP update process, the attacker's fraudulent advertisement may only affect certain portions of the Internet. This means that during the hijacking attack, some ASes may route traffic to the legitimate prefix hosts although others will not [10]. If we deploy multiple StrobeLight systems at topologically diverse locations, it is likely that at least one will detect an abrupt change in the availability profile of the target prefix.

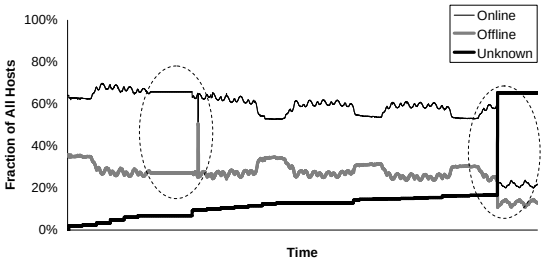
The individual StrobeLight systems do not need to reside within the core routing infrastructure—they merely need to be deployed outside of the AS that they monitor. Furthermore, there is no need for communication between the individual systems since anomalies are defined with respect to each system's local measurements. Thus, a distributed StrobeLight framework should be easy to deploy and maintain.

4.4.2 Experimental Methodology

Since our current StrobeLight prototype is *internally* deployed, its probe results are agnostic to inconsistencies in global BGP state. Thus, we validate our detection algorithms using simulations driven by real-life availability measurements. Specifically, we use data gathered between July 29, 2006 and September 1, 2006. To include the largest possible host set in our evaluation, we did not filter hosts based on their unknown fraction. This also let us validate our hypothesis that filtering out highly-unknown hosts does not change fundamental patterns in subnet availability. During this observation period, we saw 238,951 unique IP addresses.



(a) Alarms were primarily generated by two incidents. Note that there were 917 total subnets, so at most 917 alerts could be generated after any probing sweep.



(b) The first incident arose when we temporarily swapped the probing daemon to a different host, causing problems with data collection. The second incident was caused by a DNS failure which created the illusion that many hosts "disappeared."

Figure 4.13: Detecting blackhole attacks (c=0.78)

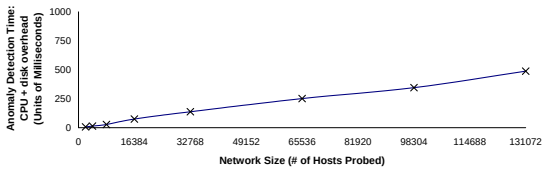
4.4.3 Blackhole Attacks

As shown in Figure 4.7, a subnet's delta fingerprint changes very slowly under normal network conditions. This suggests a simple threshold test for anomaly detection: if the similarity between two consecutive fingerprints is less than some cutoff c, StrobeLight should raise an alarm. To evaluate this technique, we configured StrobeLight to maintain fingerprint histories for subnets of size 256, raising an alarm for a subnet if self-similarity fell below 0.78 [3].

During the month-long observation period depicted in Figure 4.13, StrobeLight generated two large bursts of alarms. The first incident, between August 4 and August 7, was caused by problems with the pinger daemon. On August 4 we transferred the daemon to a different host due to a scheduled power outage in the lab containing the original machine. The new machine had a 100Mb/s Ethernet card, not a gigabit one, and it had trouble coping with the probing load, resulting in dubious availability measurements over several days. These measurements caused the first alarm spike in the incident.

The probing daemon was transferred back to the original machine on August 7, but discontinuities in data collection gave StrobeLight the illusion of widespread plunges in availability. During this episode, StrobeLight raised two alarms for almost every

[3]This value optimizes detection accuracy for experiments that we describe later, so we use it here for the sake of comparability.



Each data point represents the average of 100 trials. Standard deviations were very small.

Figure 4.14: Scalability of Analysis Engine

subnet: one when the subnet appeared to go offline, and another when the subnet appeared to return online. This explains the second alarm spike in the incident.

StrobeLight generated a much larger set of alarms on August 29. On that day, the DNS server for the Redmond domain did not respond to our query for active IPs. Thus, StrobeLight did not probe the large number of hosts residing in that locale, marking their availability status as unknown. When generating fingerprints, StrobeLight treats unknowns as offlines. Thus, the first probe sweep after the DNS failure found drastic changes in fingerprint self-similarity, with each Redmond subnet abruptly transitioning to an "all-zeroes" fingerprint vector. In effect, the DNS disruption simulated a blackhole attack, an attack which StrobeLight was able to detect.

This incident also demonstrates StrobeLight's dependence on DNS. Depending on one's perspective, this is a vice or a virtue. StrobeLight's sensitivity to DNS state means that it can detect some anomalies in DNS operation. However, this opens StrobeLight to DNS-mediated attacks in which adversaries try to disrupt StrobeLight's DNS fetches before tampering with BGP state. The IP prefixes owned by a corporation are fairly stable, so we could manually configure StrobeLight with these prefixes and probe every address without regard to whether it was assigned internally. The penalty would be an increase in the prober's network load; also, if there were many unassigned addresses, cross-subnet similarity would be higher, leading to more false alarms.

Performance

Anomaly detection consists of three steps: issuing the ping sweep from the probe machine, transferring the probe results to the analysis machine, and performing fingerprint computations on the analysis machine. The slowest step is the probe sweep, which requires about 25 seconds to cover the entire network. Transferring the results to the analysis machine is quick since probe results are small and the two StrobeLight machines are connected by a LAN. As shown in Figure 4.14, anomaly detection is

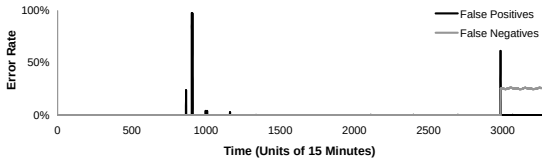


Figure 4.15: Detecting Imposture Attacks

also fast. In the current implementation of the analysis engine, ping results are pulled from the probe machine and cached on the local file system. Figure 4.14 shows that once the analyzer has pulled the ping results onto local storage, the time needed to calculate new fingerprints and perform threshold calculations is less than half a second, even for networks with 130,000 hosts.

4.4.4 Imposture Attacks

In an imposture attack, an adversary diverts traffic from someone else's IP prefix to attacker-controlled machines, attempting to mimic the behavior of the legitimate host set. Since our StrobeLight prototype was deployed within the corporate network, it could not detect availability discrepancies as seen by external ASes. So, we performed two simulation experiments driven by our empirically-gathered availability data.

Intra-corporation imposture: In our first simulation, we examined whether StrobeLight could detect the imposture of one corporate subnet by another. We iterated through our availability data in increments of 15 minutes; during each iteration, we compared each subnet's fingerprint from the previous iteration to all fingerprints from the current iteration. Let $f_{i,t}$ represent the fingerprint of subnet i at time t, and let $sim()$ return the similarity of two fingerprints. Given a similarity cutoff c, we define StrobeLight's detection accuracy at time t as follows:

- `Correct positive:` $sim(f_{i,t}, f_{j,t-1}) < c$. Subnet i's fingerprint at time t is rejected as a legitimate evolution of subnet j's fingerprint from time $t-1$.
- `Correct negative:` $sim(f_{i,t}, f_{i,t-1}) \geq c$. Subnet i's fingerprint was acceptably stable across consecutive probing intervals.
- `False positive:` $sim(f_{i,t}, f_{i,t-1}) < c$. Subnet i's fingerprints across consecutive probing intervals were so dissimilar that we raised an alarm.
- `False negative:` $sim(f_{i,t}, f_{j,t-1}) \geq c$. Subnet i's current fingerprint is so similar to subnet j's previous fingerprint that i could imposture j.

Different values of c lead to different trade-offs between false positives and false negatives. In Figure 4.15, we depict StrobeLight's detection accuracy using c=0.78,

the value that minimizes false negatives. The false negative rate was zero except during the DNS failure. During this anomaly, many subnets appeared to have zero availability, meaning that their fingerprints were indistinguishable and thus useless for detecting imposture attacks. However, as explained in the previous section, StrobeLight raised alarms at the beginning of the anomaly, so high false negatives afterwards are not concerning.

The false positive rate was generally low, with one or two unnecessary alerts issued every week. The two large spikes in false positives were triggered by the incidents described in Section 4.4.3. During the major spikes in false positives, many subnets temporarily transitioned into a zero availability state; during the falling and rising availability edges, fingerprints underwent abrupt changes and were incorrectly flagged as the targets of imposture. However, false positives with respect to imposture attacks are often true positives with respect to some other anomaly. Except for scheduled network shutdowns, it is difficult to imagine a normal operating scenario in which a subnet's availability fingerprint changes drastically over a short period.

Spectrum attacks: Ramachandran and Feamster showed how spammers can elude IP-based blacklists using short-lived manipulations of BGP state [90]. The authors observed spammers hijacking a large `/8` prefix, sending a few pieces of spam from random IP addresses within the prefix, and then withdrawing the fraudulent BGP advertisement a few minutes later. By using short-lived routing advertisements, spammers increase the likelihood that their hosts will be unreachable by the time that white hat forensics begin. By sending a small amount of traffic from each host, and by randomly scattering the traffic throughout a large address space, spammers avoid filtering by DNS-based blacklists [60].

To determine whether StrobeLight could detect spectrum attacks, we made two modifications to the simulator described above. First, instead of examining subnets of size 256, we examined the largest subnets demarcated by standard Class A/B/C rules. Second, instead of comparing subnet fingerprints at time t to those at time $t-1$, we compared subnet fingerprints at time t to those of a similarly sized attacker subnet in which a random fraction of hosts responded to probes.

Figure 4.16 shows StrobeLight's detection accuracy in the five largest subnets. We also show results for an attack against the "meta-subnet" containing all hosts, since this is the best that we can approximate a large `/8` prefix. Each cluster of bars represents detection accuracy for a specific subnet. Within a cluster, the i-th bar is our detection accuracy when a random $i * 10\%$ of hosts in the attacker subnet responded to probes.

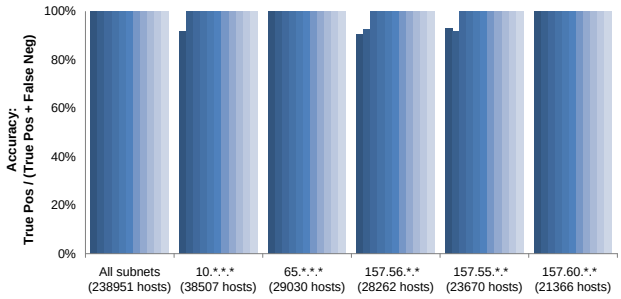


Figure 4.16: Detecting spectrum agility attacks

StrobeLight had perfect detection accuracy across all time steps in the meta-subnet containing 238,951 hosts. StrobeLight also had perfect accuracy for two of the five classful subnets. In the other three, detection accuracy for low response fractions dipped as low as 90%. These subnets were affected by the DNS failure and spent the last part of the observation period in an unknown state. Since StrobeLight presumes unknown hosts to be offline, an attacker could hijack these subnets after the DNS failure and evade detection by rarely responding to StrobeLight pings. However, as shown in Section 4.4.3, StrobeLight would raise alarms at the beginning of the DNS anomaly, so human operators would be more vigilant for additional problems during this time period.

If the attacker could measure availability trends in our subnets, he could mimic the legitimate distribution of probe responses during the spectrum attack and avoid detection by StrobeLight. However, many organizations already perform ingress filtering of ping probes destined for internal hosts, eliminating the most obvious way for an adversary to collect availability data. Other methods, such as passive traffic snooping, seem less accurate and more difficult to implement effectively.

4.4.5 Interception Attacks

In an interception attack, the adversary convinces routers to send other people's traffic through attacker-controlled machines; these machines may inspect or tamper with the packets before forwarding them to their real destination. Previous research has shown that the hop count between two arbitrary prefixes is stable in the short to medium term [106, 115]. Since interception attacks are likely to lengthen the route between sender and the target prefix, they can be detected by monitoring the hop count between the target prefix and distributed vantage points. This idea, first proposed by Zheng *et al* [115], can be integrated into StrobeLight's probing mechanism. If StrobeLight sets the TTLs in its outgoing probes to a value close to

that of the actual hop count to the target hosts, then an increase in route lengths to the targets will cause outgoing probes to be dropped before they reach their destination. StrobeLight will notice a massive drop in probe responses and raise alarms for the affected subnets.

To elude detection, the attacker can try to alter the TTLs inside StrobeLight's probes, e.g., by resetting TTLs to their maximum value to ensure that probes reach their final destinations. To detect such activity, StrobeLight can augment each primary probing sweep with a smaller, concurrent sweep. This smaller sweep targets randomly chosen host pairs which have highly correlated uptime and which are predicted to be online at the current moment; the latter can be determined using availability prediction [74] or the simple observation that two machines that were online 30 seconds ago are likely to be online now. The first host in each pair has a probe with a TTL set to its true distance $ttl_{1,real}$ from the prober. The second host has a probe TTL $ttl_{2,short}$ that is less than its real distance. In normal situations, the first probe will succeed and the second one will fail. During an interception attack in which the distance between the prober and the TTL-modifying interceptor is less than $ttl_{2,short}$ hops, both probes will succeed. This can trigger a more intensive round of TTL staggering that may eventually result in an alarm.

By varying the TTL stagger between each host pair, StrobeLight can detect interceptors at various depths. It is difficult for an adversary to dupe such a system without a priori knowledge of where the probers are and how host pairs and TTL staggers are chosen.

Our current StrobeLight prototype does not track route lengths to destination prefixes or use non-standard TTL settings. Thus, a proper evaluation of the TTL technique must be deferred to future work. However, we are currently making the necessary changes to the probing daemon and the analysis engine, and we are optimistic about the future performance of the TTL technique.

4.5 Conclusions

Many distributed systems would benefit from an infrastructure that collected high resolution availability measurements for individual hosts. Unfortunately, existing frameworks either do not scale, do not track every host in the network, or store data in such a way that makes global analysis difficult. In this chapter we describe StrobeLight, an enterprise-level tool for collecting fine-grained availability data. Our current prototype has measured the uptime of hundreds of thousands of hosts in our corporate network for almost two years. Using the longitudinal data generated by

this tool, we performed extensive analyses of availability in our wired and wireless networks. We also demonstrated how StrobeLight's real-time analysis engine can detect network anomalies such as DNS failure and IP hijacking.

StrobeLight's design was driven by the goals of simplicity, unobtrusiveness, and high fidelity measurement. An individual StrobeLight system is easy to deploy, requiring modification to neither end hosts nor interior routers. Its probing footprint is light and does not interfere with preexisting traffic. StrobeLight's 30 second measurement granularity allows for high-resolution introspection of uptime at the level of individual hosts. When StrobeLight is deployed at multiple vantage points to improve anomaly detection coverage, the individual instances do not need to coordinate their activities. Using our corporation's standard distributed data store to manage StrobeLight's availability traces, we make it easy for other systems in our enterprise to access this rich data set.

CHAPTER V

Detecting Faulty Overlay Forwarders Using Concilium

In a distributed system, hosts are connected by a physical communication network. The most popular network is the Internet, a global routing framework containing a vast number of IP routers, fiber optic cables, satellite links, and so on. The IP protocol supports only one type of routing address, namely, a 32-bit integer. If distributed applications desire arbitrary routing identifiers, they can build an overlay network [91, 101] atop the IP infrastructure. In an overlay, regular end-hosts act as forwarding agents, and multi-hop routes are composed from the IP-level paths that connect overlay peers.

Three conditions must be true for an overlay to successfully deliver a message.

- First, there must be a series of overlay-level segments which connect the sender and the destination. This condition is enforced via a distributed protocol which maintains global routing invariants. These protocols typically assume that faulty peers do not provide improper routing state to their peers.
- Second, for each of the segments in the overlay-level route, the underlying IP path must be functional. This may or may not be true, since IP routes can fail for a variety of reasons that are beyond the control of the overlay.
- Third, for each of the segments in the overlay-level route, the starting peer must successfully forward the message to the ending peer.

Consider an overlay-level route that starts at peer A, goes through B, and terminates at C. Who is to blame if C never receives the message from A? A or B may have inadvertently forwarded the message to the wrong host if the distributed routing protocol has been corrupted. Alternatively, the IP-level path connecting A to B or B to C may be down. A third possibility is that B successfully received the message and could have sent it to C, but it decided to drop the message of its own volition, either due to misconfiguration or malice.

In this chapter, we describe Concilium, a fault diagnosis system that ascribes

blame for message drops in overlays. Once Concilium has detected a misbehaving overlay peer or a bad link in the core IP network, the overlay can route around the problem. It can also notify the owner of the malfunctioning component, since the owner may not be aware of the local fault. Both actions can improve the overall reliability of the distributed service.

Concilium protects the routing maintenance protocol by extending techniques from the secure overlay community [25]. By applying statistical tests to peers' self-advertised routing state, Concilium detects faulty route information and prevents it from contaminating the global forwarding system. To determine whether IP-level paths are broken, Concilium uses distributed network tomography [5, 41]. By comparing application-level drop rates with network characteristics inferred from tomography, Concilium generates the likelihood that message loss is due to a misbehaving overlay host or a poor path in the underlying IP network. Unlike Fatih [75] or packet obituary systems [7], Concilium does not require modification to core Internet routers. In contrast to RON [5], Concilium detects hosts which contribute faulty tomographic data to their peers.

5.1 Secure Overlays

Structured peer-to-peer overlays provide a decentralized, self-managing routing infrastructure atop preexisting IP networks. Each host is associated with an overlay identifier. When a host must forward a message, it consults locally maintained routing state to determine the next hop. In overlays like Pastry [91] and Chord [101], the local routing state consists of two logical components. The *leaf table* points to the peers with the numerically closest identifiers to the local host's identifier. The *jump table* points to peers whose identifiers differ from the local one by increasing, exponentially spaced distances. Messages are typically forwarded using jump tables until the last hop.

Castro *et al* introduced *secure overlay routing* [25] to prevent malicious nodes from subverting the forwarding process. In a secure routing framework, messages are delivered with very high probability if the fraction of non-faulty hosts is at least 75%. Concilium uses several features of secure routing to protect its distributed tomographic protocol. We briefly describe these features before discussing Concilium in more depth.

Before a host can join a secure overlay, it must acquire a certificate from a central authority. The certificate binds the host's IP address to a public key and an overlay identifier. Since identifiers are static and randomly assigned, adversaries cannot

deliberately move their hosts to advantageous regions of the identifier space. Hosts also enforce strict constraints on the peers which can occupy each jump table slot. For example, in standard Pastry [91], a peer in row i and column j of a routing table must share an i-character identifier prefix with the local host and have j as its $i+1$ character; there are no constraints on the remaining characters. In secure Pastry, the peer must be the online host whose identifier is closest to point p, where p is the local host identifier with the i-th character substituted with j. These stronger peering constraints, in concert with random identifier distribution, limit the fraction of malicious peers in local routing state to the fraction of malicious nodes in the total overlay.

Using a *density test*, a host can probabilistically detect when peers misreport their leaf sets. By comparing the average inter-identifier spacing in its own leaf set to that of a peer's leaf set, a host can identify advertised leaf sets that are too sparse. In the absence of such checks, an adversary could suppress knowledge of peers that it does not control, forcing routing traffic or data fetches to go through corrupt peers.

For performance reasons, peers maintain both secure routing tables and "standard" routing tables. Standard tables can use techniques like proximity affinity [26] to minimize routing latency or maintenance bandwidth; secure routing is only used when standard routing fails. Messages requiring Concilium's fault attribution must always be forwarded using secure routing. Other messages can be forwarded using either mechanism.

5.2 The Concilium Diagnostic Protocol

Concilium diagnoses faulty overlay routes using a multi-step process. First, hosts exchange their routing tables so that they can determine the first few hops that a locally forwarded message will take. Second, hosts test IP-level network conditions using locally-initiated network probes. By exchanging the results of these tests, individual peers synthesize a global picture of link quality throughout the Internet. By combining routing data with the collaborative map of network conditions, nodes can identify broken IP links and misbehaving overlay forwarders; the latter are defined as end-hosts which drop messages when the IP-level paths to their routing peers are good.

When a host is deemed faulty, Concilium issues a *fault accusation* against that host. Each accusation is provisional, since the accused host may be able to prove its innocence by showing that messages were actually being dropped further down the route. If the accused host can generate a verifiable *fault rebuttal*, Concilium will

revise its original accusation. Otherwise, hosts may refuse to peer with the accused node or treat its behavior with extra suspicion.

In this section, we describe the Concilium protocol in the context of a particular implementation strategy. We then discuss alternative implementations.

5.2.1 Validating Routing State

To troubleshoot end-to-end overlay routes, Concilium must validate the routing state that peers self-report. Concilium validates leaf sets using Castro's test and introduces a new test to verify jump tables. Like Castro's leaf test, Concilium's jump table test is a density check. However, instead of examining the average inter-identifier spacing in a jump table, it checks how many slots are occupied. Jump tables with low occupancy are considered suspicious. For the sake of concreteness, we describe the test in the context of a secure Pastry overlay, but the test can be extended to other overlays in a straightforward manner.

In secure Pastry, overlay identifiers are ℓ characters long and each character can assume one of v different values. ℓ is typically 32 or 40, and v is usually 16. Each node maintains a jump table with ℓ rows and v columns. The identifier in row i and column j shares an i character prefix with the local host's identifier and has an $i+1$-th character of j. Assuming that identifiers are randomly distributed throughout the identifier space, the probability that a node does not have a particular prefix of length ℓ_{prefix} is $1-(1/v)^{\ell_{prefix}}$. The probability that an entry in row i of a routing table is filled is equal to one minus the probability that no identifier exists with the appropriate prefix. Thus,

$$Pr(entry\ filled\ in\ row\ i) = 1 - \left[1 - \left(\frac{1}{v}\right)^{i+1}\right]^{N-1} \tag{5.1}$$

where N is the total number of nodes in the overlay. Nodes can estimate N by inspecting the inter-identifier spacing in their leaf sets [70].

Let $p_{i,j}$ denote the probability that an entry in row i and column j is filled, as given by Equation 5.1. Each $p_{i,j}$ is an independent Bernoulli random variable, so the occupancy distribution for the entire table is governed by a Poisson binomial distribution. The mean and variance are

$$\mu = \frac{1}{\ell v}\sum_{i=1}^{\ell}\sum_{j=1}^{v} p_{i,j} \qquad \sigma^2 = \frac{1}{\ell v}\sum_{i=1}^{\ell}\sum_{j=1}^{v}(p_{i,j}-\mu)^2.$$

Computing exact values for the Poisson binomial distribution is intractable for non-trivial numbers of Bernoulli variables. Thus, it is difficult to directly calculate the

likelihood that a table contains a particular number of occupied slots. Fortunately, since σ^2 is high, we can use a normal approximation with little loss in accuracy [62]. In this approximation, the mean μ_ϕ and the variance σ_ϕ^2 are

$$\mu_\phi = \ell v \mu \qquad\qquad \sigma_\phi^2 = \ell v \mu (1-\mu) - \ell v \sigma^2.$$

The cumulative distribution function for table occupancy is $\phi(\mu_\phi, \sigma_\phi)$ where $\phi()$ is the cdf for the normal distribution. To test whether an advertised jump table is too sparse, a host compares its local jump table density d_{local} to the advertised d_{peer}. If $\gamma d_{peer} < d_{local}$ for some small $\gamma > 1$, the peer's jump table is deemed invalid. In Section 5.3.1, we use $\phi(\mu_\phi, \sigma_\phi)$ to select γ based on the resulting likelihood of false positives and false negatives.

The occupancy test prevents malicious hosts from advertising jump tables that are too sparse. We also wish to prevent hosts from advertising tables that are too dense. Since identifiers are centrally issued, a misbehaving host cannot fabricate an identifier for an arbitrary jump table slot. However, a host can collect identifiers from peers that have gone offline and use these identifiers to inflate its advertised table density [25]. To protect against inflation attacks, Concilium requires a jump table entry referencing peer H to contain a signed timestamp from H. Whenever host G probes H for availability, H piggybacks a signed timestamp upon the probe response. Later, when G advertises its jump table, it includes the signed timestamps for each non-empty entry. Peers will reject the table if it has stale timestamps.

5.2.2 Collecting Tomographic Data

Each host H is connected to its routing peers by a set of links in the underlying IP network. These links induce a communication tree T_H whose root is H and whose leaves are H's routing peers. We define the forest F_H as the union of the tree rooted at H and the trees rooted at each of H's routing peers. Concilium's goal is to estimate link quality in F_H. To do so, each tree root periodically probes the link quality in its tree. Peers then exchange their tomographic results to create a collaborative estimate of link quality in F_H.

Before a host can initiate the tomography process, it must determine the physical IP links which comprise its tree. These link maps can be derived using tools such as RocketFuel [99]. Internet routes are often stable for at least a day [113], so topological data need not be fetched often.

Once the topology is known, hosts infer link quality using lightweight proactive probing and heavyweight reactive probing. Lightweight tomography uses the avail-

ability probes that hosts already send to their routing table peers [91, 101]. The period of these probes is a minute or less, and the duration of high loss events in IP links is on the order of tens of minutes [71]. Thus, H can use these preexisting probes to detect high intensity packet loss inside T_H. More specifically, H schedules a lightweight probe of T_H as a periodic task whose inter-arrival time is picked randomly and uniformly from the range $[0, max\_probe\_time]$; max_probe_time is on the order of one or two minutes. H probes its entire routing table at once using a simplified version of Duffield's striped unicast scheme [41]. H generates a single probe packet for each routing peer, but it issues these packets back to back. Since these packets will stay close to each other as they traverse shared interior routers, they emulate a single multicast packet sent to the leaves of a multicast tree. If H receives acknowledgments from all peers, it assumes that there is no link loss. Otherwise, it sends a few more probes to silent peers to determine if they are truly offline or situated along a lossy IP link.

If link loss is detected or H's application-level messages are not being acknowledged, H initiates heavyweight probing. Heavyweight tomography also uses striped unicast probing, but H sends many probes to each leaf using Duffield's full scheme. Loss rates for each root-leaf path are inferred using the number of acknowledgments received from each leaf host. Using maximum likelihood estimators, these end-to-end loss rates induce loss rates for each internal IP link.

When H initiates heavyweight probing, it asks its routing peers do the same. This ensures the availability of fine-grained, high quality tomographic data for the entire forest during the speculated fault period. To avoid probe-induced congestion, each peer waits for a small, randomly picked time before initiating heavyweight tomography.

After H has probed T_H using lightweight or heavyweight mechanisms, it sends a timestamped snapshot of T_H and its summarized probe results to its routing peers. The probe results for each path can be encoded in a few bits representing predefined loss rates. H signs the tomographic snapshot with its public key, both to prevent spoofing attacks and to prevent H from disavowing previously advertised probe results.

Each leaf node in T_H is one of H's routing peers, so H implicitly advertises its forwarding state when it publishes its tomographic data. This data also includes the signed freshness timestamps for each routing entry as described in Section 5.2.1. When a node receives a snapshot from H, it verifies all the signatures, checks the freshness of each entry, and performs the density checks. If any of these tests fail, the

node may issue a fault accusation against H as described in Section 5.2.4. Regardless, the node archives H's snapshot. As the node receives snapshots from other peers, it constructs a distributed view of the forwarding paths emanating from its routing peers and the quality of IP links in these paths.

5.2.3 Error-checking Tomographic Data

Striped unicast tomography assumes that leaf nodes will return acknowledgments for received probes. A faulty or malicious leaf can try to respond to probes that were actually lost in the network, or drop acknowledgments for probes that were received. The former only affects inferences over the last mile to the misbehaving leaf, but the latter can ruin many inferences throughout the tree [8]. Fortunately, we can detect both types of misbehavior. To detect spurious responses to non-received probes, the probing node includes nonces in its probes. To detect leaves which faultily suppress acknowledgments, the probing node applies statistical tests to verify that the acknowledgment patterns of its leaves are consistent with each other [8]. Thus, an intentionally malicious leaf can accomplish the most damage by responding correctly to the probes of other nodes, but misreporting the results of its own probes. We explore this issue further in Section 5.3.3.

We assume that interior IP routers can be faulty but not actively malicious. We assume that they do not interfere with probes or their responses in a byzantine way.

5.2.4 Attributing Fault

Armed with link measurements and routing information, each Concilium node can issue accusations for dropped messages. Suppose that at time t, host A sends a message to Z through B. By checking its copy of B's routing table, A can determine the host C to which B will forward the message. If A never receives a signed acknowledgment from Z, it checks its tomographic data for probes which test links in the path between B and C. If one or more links were probed as down, Concilium assigns blame to the network. Otherwise, Concilium determines that B was faulty. This judgment may be erroneous, since the true culprit may lie downstream from B. We describe how Concilium recovers from these mistakes in Section 5.2.5. For now, we restrict our attention to the original issuance of blame.

Let $B \rightarrow C$ represent the path between B and C, and let $probes$ be the set of probe results covering links in $B \rightarrow C$. We allow this set to contain results from probes initiated within the interval $[t-\Delta, t+\Delta]$, where Δ might equal sixty seconds. Let $probes(link)$ be the set of probes covering a particular link. For $p \in probes(link)$,

let $p.l\_up \in \{0\ or\ 1\}$ be the probed status of the link, with 1 representing a link that was up and 0 representing a failed link. Let $a \in [0, 1]$ be the accuracy of probes in diagnosing link failure. Returning to our running example, when A fails to receive an acknowledgment from Z, it ascribes blame to B as follows:

$$\begin{aligned} Pr(B\ faulty) &= Pr(B \rightarrow C\ good) \\ &= 1 - Pr(B \rightarrow C\ bad) \\ &= 1 - Pr(B \rightarrow C\ has \geq 1\ bad\ link) \end{aligned} \tag{5.2}$$

where $Pr(B \rightarrow C\ has \geq 1\ bad\ link)$ equals

$$max_{l \in B \rightarrow C} \left(\frac{\sum_{p \in probes(l)} [p.l\_up(1-a) + (1-p.l\_up)a]}{|probes(l)|} \right). \tag{5.3}$$

We use `max` as the `OR` operator from fuzzy logic [14]. In the context of Equation 5.3, it selects the link in $B \rightarrow C$ for which A has the highest confidence that it was bad, with each probe result weighed equally. For example, suppose that Q and R probe a link as down (0) and S probes the same link as up (1). If a equals 0.8, A believes that the link was bad with confidence (1/3)(0.8)+(1/3)(0.8)+(1/3)(0.2)=0.6.

Importantly, when A judges the trustworthiness of B, it does not incorporate B's probe results into Equation 5.3. This prevents a malicious B from influencing the amount of blame that A ascribes to it. For example, if A included B's probe results in Equation 5.3, B could reduce its level of blame by claiming that it probed a link in $B \rightarrow C$ as down.

Using Equation 5.2, A determines the amount of blame that it ascribes to B for a particular dropped message. If the blame is larger than a threshold described in Section 5.3.3, A assigns a *guilty verdict* to B; otherwise, A assigns a guilty verdict to the network. A maintains a sliding window of the last w verdicts that it issued for B, archiving the tomographic data used to make each verdict. If B receives m or more guilty verdicts in this window, A inserts a *formal fault accusation* into a DHT which exists atop the secure overlay. The insertion key for the accusation is B's public key, and the accusation contains all of the signed tomographic data that A used to derive its fault assessments. Insertions and fetches of the formal accusation are secured using Castro's techniques [25], and the statement is signed by A so that it can be held accountable for spurious accusations. When another host considers B as a routing peer, it first retrieves accusations against B from the DHT. For each accusation, the host uses the associated tomographic data to independently verify

the fault calculations. If the host verifies the accusations, it considers B to be a "bad peer" and sanctions it according to network-specific policies.

5.2.5 Revising Incorrect Fault Attributions

As currently described, a Concilium node cannot ascribe blame beyond the next overlay hop. Returning to our running example, if A does not receive a signed acknowledgment from Z, and A estimates all links in $B \rightarrow C$ to be good, then A will always blame B for dropping the message, even if B successfully forwarded the message and it was actually dropped further downstream. To correctly ascribe blame in these situations, Concilium uses *recursive stewardship* of messages and *recursive revision* of fault accusations.

Whenever a peer along $A \rightarrow Z$ forwards a message, it treats the message as if it were generated locally—in other words, each forwarding peer expects to receive an acknowledgment from Z. If Z receives the message successfully, it routes its acknowledgment along the reverse forwarding path. If Z never receives the message or its acknowledgment is dropped along the reverse path, a chain of guilty verdicts will be issued. By considering them as a whole, Concilium can determine where blame should ultimately be placed. For example, suppose that D faultily drops A's message to Z along $A \rightarrow B \rightarrow C \rightarrow D \rightarrow ... \ Z$ and that all IP links are good. Using recursive stewardship, B and C will await an acknowledgment from Z. When this acknowledgment does not arrive, A will blame B, B will blame C, and C will blame D. D will not be able to blame a forwarding peer since it lacks incriminating tomographic data—D's peers in F_D will not have probed any links as down [1], and D cannot fabricate such probes itself because a node's own probes are ignored when calculating blame for that node. Thus, the accusation chain stops at D and nodes absolve themselves of unfair blame by pushing locally generated verdicts upstream. First, C presents its guilty verdict against D to B. B examines the signed, timestamped tomographic data in the verdict, verifies the blame calculation, and amends its accusation against C to be an accusation against D. B presents its amended verdict to A. After A verifies the inference, it amends its accusation against B to an accusation against D. Innocent nodes have now been exonerated, and blame has been fairly attributed to D. Note that an amended accusation contains the signed, timestamped data from both the original verdict and the revision that was pushed upstream. This allows amended verdicts to be self-verifying.

[1] This assumes that the nodes in F_D are not colluding with D. We return to the issue of colluding nodes in Section 5.3.

Faulty nodes may not push revision information upstream. They do so at their own peril, since they will receive the blame for the message drop. For example, if C does not push its accusation against D to B, then B will not amend its original fault claim against C, and A will eventually blame C, not D, for the message drop.

A faulty node may receive a revision but refuse to update its local accusation. For example, A may receive B's blame against a node further downstream but continue to blame B. To guard against such misbehavior, B archives its local fault attributions and revisions. If another host believes that B is untrustworthy, it allows B to defend itself before any punitive steps are taken. The host presents B with the relevant formal accusations. If B can rebut these accusations using its local archives, the other host will recalculate B's trustworthiness in light of the new evidence.

5.2.6 Preventing Spurious Accusations

Up to this point, we have focused on detecting hosts which fail to forward messages. However, the original message sender can also misbehave. Suppose that each link in $B \rightarrow C$ is good. If A accuses B of dropping its message without actually sending one to B, other nodes will believe the accusation; they will verify the tomographic information in A's accusation and derive the same blame probability as A.

To prevent such spurious accusations, Concilium uses *forwarding commitments*. When A sends a message through B, B sends a signed statement to A indicating its willingness to forward the message. The commitment includes a timestamp, A's identifier, B's identifier, and the identifier of the ultimate destination Z. When A issues an accusation against B, it includes this forwarding commitment along with the relevant tomographic data and routing state. In this fashion, B can only be blamed for dropping messages that it agreed to forward. B can batch its commitments and asynchronously piggyback them upon its responses to A's availability probes. Like message stewardship and accusation revision, forwarding commitment is also recursive.

A malicious B may refuse to issue forwarding commitments for A's packets. Without support from core IP routers on the path between A and B, there is no way for Concilium to establish that A actually sent a message to B, or that B sent a forwarding commitment which A ignored. Lacking such knowledge, Concilium cannot comment on the trustworthiness of either peer. Fortunately, B's misbehavior can be detected by other mechanisms. For example, if overlay hosts are part of a decentralized reputation system such as Creedence [107], A can issue a vote of no confidence

in B using this reputation system. Since honest hosts trust each other's votes, they will eventually determine that B makes a poor peer and treat it accordingly.

Note that standalone reputation systems cannot replace Concilium's full accusation protocol. Reputation systems allow a node to make a direct accusation against the next hop in a route, but they provide no structured way to propagate accusations against nodes that are farther downstream. Using recursive stewardship of messages and recursive revision of accusations, Concilium provides such a capability. Concilium also provides self-validating accusations which can be confirmed by arbitrary third parties.

5.2.7 Putting It All Together

In this section, we provide a concise summary of the entire Concilium protocol.

Collecting Routing State and Tomographic Data: Each host periodically measures the network quality in the IP link tree induced by its routing state. Hosts exchange their signed, timestamped tomographic data to create a distributed view of link quality along each possible next-hop route. A node's tomographic data implicitly advertises its routing state. Peers validate this state using Castro's leaf set test [25] and Concilium's new jump table test. Tomographic probe results are also validated using statistical checks [8].

Generating and Revoking Accusations: When host A sends a message through B, B generates a signed routing commitment indicating its willingness to forward the message; these commitments can be batched and piggybacked in bulk atop responses to availability probes. If A does not receive a commitment, it issues a vote of no confidence in B through a reputation system. Otherwise, it awaits an acknowledgment from Z. As the message travels down the overlay route, peers use recursive stewardship and wait for an acknowledgment as well. If there are no problems, the acknowledgment is sent through the reverse forwarding path. Otherwise, one or more hosts issue local fault accusations. These accusations may be incorrect, since the misbehavior of downstream nodes falsely implicates upstream nodes. Falsely accused nodes can clear their reputation by pushing their local accusations upstream. Since accusations contain self-validating, signed data, honest nodes will be vindicated and faulty nodes will be punished. If a host accumulates too many local accusations for a peer, it inserts a formal accusation into a secure DHT. This accusation contains the signed, self-verifying information used to justify the accusation.

5.2.8 Implementation Options

Up to now, we have assumed that each node performs its own tomographic probing. However, hosts which trust each other and reside in the same stub network can consolidate probing responsibility. For example, hosts could take turns issuing the probes for the multi-forest induced by their collective routing state. Alternatively, all hosts could defer probing responsibility to a shared administrative machine such as a RON gateway [5]. Either solution would make heavyweight probing less onerous, since the bandwidth cost for probing shared links could be amortized across multiple nodes.

As described in Section 5.2.4, a fault judgment is based on the acknowledgment of an individual message reception. If two peers exchange many packets, it may be useful for a single acknowledgment to cover multiple messages. The acknowledgment could indicate loss rates in several ways [75], e.g., through simple counters indicating how many packets arrived, or packet hashes identifying the specific packets which were received.

Concilium's goal is to find misbehaving overlay hosts and broken IP links, but it is agnostic about the response to its fault identifications. Broken IP links are often discovered quickly by the responsible ISP, so an overlay may simply avoid certain overlay paths until the fault is fixed [5]. With respect to faulty overlay hosts, Concilium allows each system to set an appropriate sanctioning policy. For example, accused hosts may not be trusted to forward sensitive messages.

If the overlay is used as a substrate for a higher level service such as a DHT, then honest nodes must not make local decisions to evict accused nodes from leaf sets. Otherwise, inconsistent routing [24] will arise and the higher level service may break. A network can mandate that a node be *universally* blacklisted if it receives accusations at a certain rate. In such an environment, nodes would check the accusation repository before agreeing to peer with a new host. If the prospective peer was discovered to be faulty, it would not be added to the local routing table.

5.3 Evaluation

In this section, we use extensive simulations to evaluate the accuracy of our jump table check, the coverage properties of our collaborative tomography, and the error rate of our accusation algorithm. We also investigate the bandwidth overhead of the Concilium protocol.

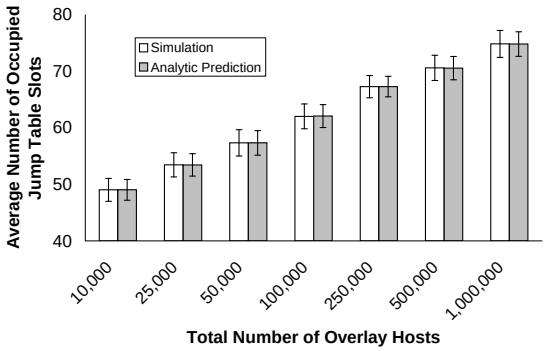


Figure 5.1: Modeling jump table occupancy

5.3.1 Jump Table Validation

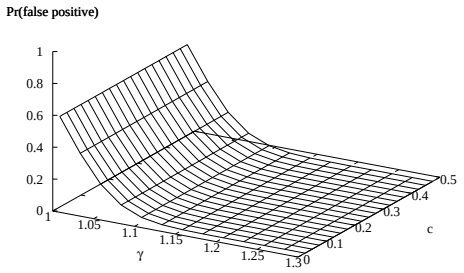
Peers exchange their routing state so that Concilium can determine the IP-level tomographic data needed to make fault accusations at the overlay level. If peers can advertise incorrect routing tables without detection, innocent peers may be accused and faulty peers may go unpunished. Thus, the success of Concilium hinges on its ability to detect fraudulent routing advertisements. In this section, we analyze Concilium's jump table tests; we defer an analysis of leaf set checks to Castro's work [25].

Our jump table test uses the cdf $\phi(\mu_\phi, \sigma_\phi)$ to model the distribution of occupancy fractions. Figure 5.1 compares the occupancy levels predicted by the analytic model with the occupancy levels seen in Monte Carlo simulations of table occupancy (y-bars indicate standard deviations). We see that the $\phi(\mu_\phi, \sigma_\phi)$ distribution accurately approximates real occupancy levels.

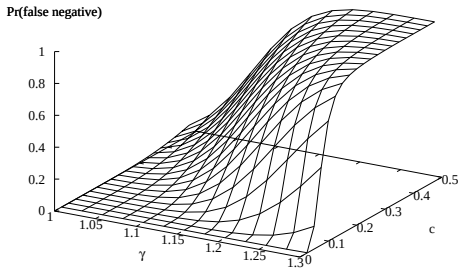
Our density test declares that a jump table is faulty if $\gamma d_{peer} < d_{local}$. The test can produce both false positives and false negatives. A false positive occurs if a non-faulty peer has a legitimately sparse jump table but is deemed faulty anyways. The likelihood of a false positive is equivalent to

$$Pr(\gamma d_{peer} < d_{local}) = \sum_{0 \leq d_i \leq \ell v} \left[Pr(d_i) Pr(d < \frac{d_i}{\gamma}) \right]$$
$$= \sum_{0 \leq d_i \leq \ell v} \left[(\phi(d_i + \frac{1}{2}) - \phi(d_i - \frac{1}{2}))\phi(\frac{d_i}{\gamma}) \right]$$

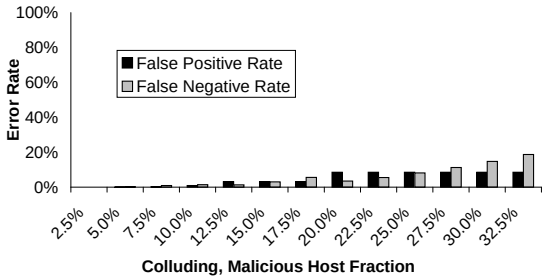
Figure 5.2(a) depicts the false positive rate as a function of γ and the fraction c of colluding malicious nodes. This graph assumes that malicious nodes may drop messages, but they may not try to go offline in a concerted attempt to skew local density estimates [25]. Thus, the false positive rate is independent of the fraction



(a) False positive probability.



(b) False negative probability.



(c) Overall misclassification rate when γ is chosen to minimize the sum of the two error probabilities.

Figure 5.2: Error rates (no suppression attacks)

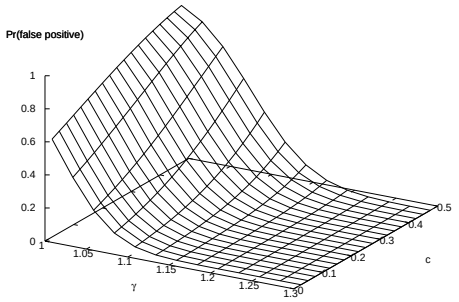
of malicious peers. Later in this section, we will revisit this graph in the context of suppression attacks.

The likelihood of a false negative is

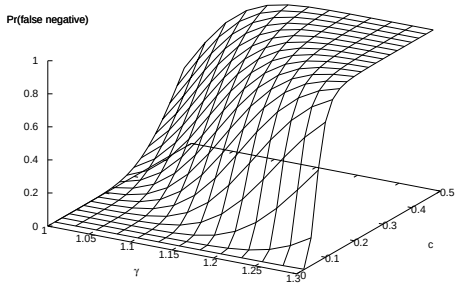
$$\begin{aligned} Pr(\gamma d_{peer} \geq d_{local}) &= \sum_{0 \leq d_i \leq \ell v} [Pr(d_i) Pr(d < \gamma d_i)] \\ &= \sum_{0 \leq d_i \leq \ell v} \left[(\phi(d_i + \frac{1}{2}) - \phi(d_i - \frac{1}{2})) \phi(\gamma d_i) \right] \end{aligned}$$

A false negative occurs when a peer advertises a jump table that only contains attacker-controlled nodes and the table passes the density test. Figure 5.2(b) shows the false negative probability in the absence of suppression attacks. Due to the properties of secure routing tables, an attacker is expected to control only c percent of all nodes in a jump table. Thus, the density of the attacker's fraudulent table is modeled as that of a legitimate table in an overlay with Nc total hosts. In the previous equation, when we calculate $Pr(d_i)$, i.e., the probability that the advertised jump table contains d_i nodes, we use Equation 5.1 but set the number of nodes to Nc.

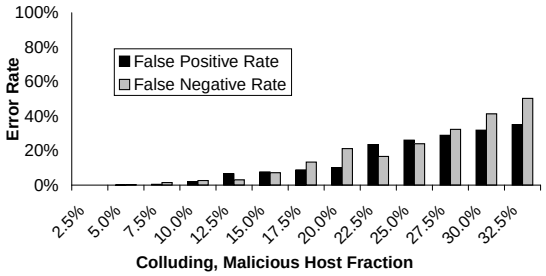
Using Figures 5.2(a) and (b), we can choose the γ which minimizes some error



(a) False positive probability.



(b) False negative probability.



(c) Overall misclassification rate when γ is chosen to minimize the sum of the two error probabilities.

Figure 5.3: Error rates (suppression attacks)

metric. For example, Figure 5.2(c) shows the misclassification rate when γ is chosen to minimize the sum of the false positive probability and the false negative probability. If 30% of all peers are malicious and colluding, the false positive rate is 8.5% and the false negative rate is 14.8%. If 20% of hosts collude, the false negative rate decreases to 3.5%.

Figure 5.3 shows misclassification rates when adversaries can launch suppression attacks. We model these attacks by supplying our false positive/negative equations with the appropriately skewed versions of N as we did above. Like Castro's density tests for leaf sets [25], our jump table checks are not very reliable if more than 20% of hosts are malicious and colluding. For example, with a c of 20%, the false positive rate is 10.1% but the false negative rate is already 21.1%. Devising effective defenses against suppression attacks is an important area for future research. However, we note that c represents the largest set of *colluding* malicious nodes. The total number of malicious nodes may be much larger, but their power is limited by the extent to which they can coordinate the suppression of their identifiers.

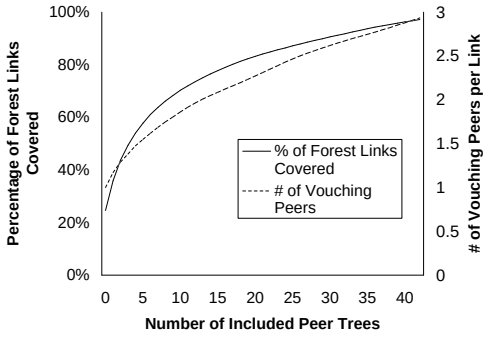


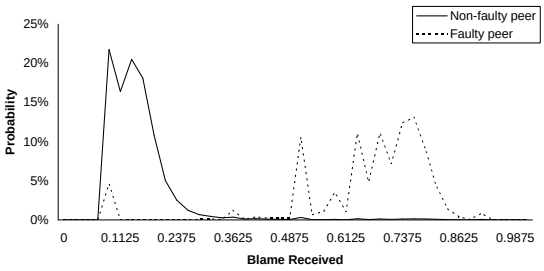
Figure 5.4: Trees Sampled vs. Forest Coverage

5.3.2 Link Coverage

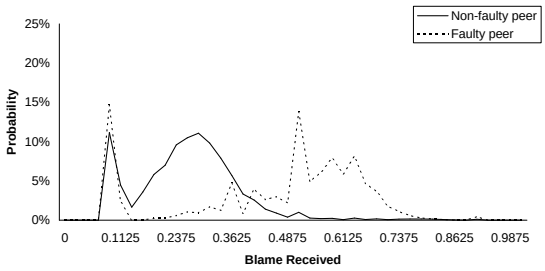
To test the coverage of tomographic probing and the accuracy of Concilium's accusation algorithm, we used a discrete event network simulator. The simulator modeled link failure, tomographic probing, the collaborative dissemination of probe results, and three types of message events (message sent, message acknowledged, message not acknowledged). The simulator placed a Pastry overlay atop an IP topology gathered by the SCAN project [48]. The topology contained peering information for 112,969 routers connected by 181,639 links. Following the methodology of Chen *et al* [29], we defined end hosts as routers with only one link and randomly selected 3% of these machines to be Pastry nodes. The resulting overlay possessed 1,131 nodes.

In the simulations, 5% of links were bad at any moment. Average link downtime was 15 minutes with a standard deviation of 7.5 minutes; this accords with empirical observations of high loss incidents lasting for a few tens of minutes [71]. Failures were biased towards links at the edge of the network [71]. To select a new link for failure, we randomly picked an overlay host and a random peer in that host's routing state. We then used a beta distribution with α=0.9 and β=0.6 to select the depth of the link that would fail. Simulations lasted for two virtual hours. We did not model fluctuating machine availability since we wanted to focus on the fundamental properties of our fault inference algorithm.

Figure 5.4 shows the average percentage of IP links in F_H that are covered when H includes a given number of peer trees. If a node probes only its own tree, it can gather tomographic data for 25% of its forest links. Increasing the number of included peer trees results in large initial gains, but the improvement in coverage diminishes as more trees are included. This is because only a few trees are needed to cover highly shared links in the center of the Internet, but many trees are needed to cover all of the last-mile links that are only used by a few hosts.

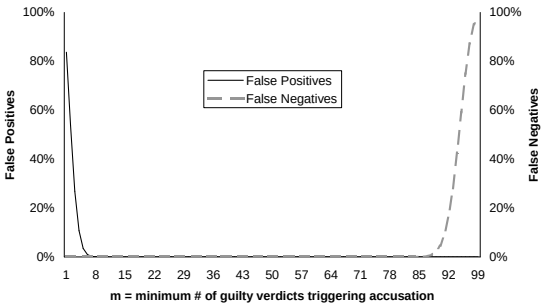


(a) The two pdfs are very distinct when nodes correctly report their tomographic data.

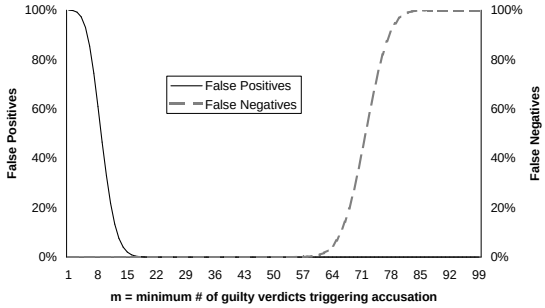


(b) The pdfs are less distinct when 20% of peers maliciously invert probe results, but Concilium can still make accurate judgments using a thresholding scheme.

Figure 5.5: PDFs for blame as generated by Equation 5.2 (max_probe_time=120 secs, Δ=60 secs)



(a) If nodes faithfully report probe results, an m of 6 drives both error rates below 1%.



(b) If 20% of hosts are colluding and malicious, an m of 16 drives both error rates below 1%.

Figure 5.6: Accusation error (w=100)

As shown in Figure 5.4, gathering probe results from more peers increases the average number of hosts that test a given link and can potentially vouch for the status of that link at an arbitrary time. By increasing the number of vouching peers for a link, we improve the quality of tomographic inferences for that link. Greater link coverage also reduces the ability of malicious nodes to taint the diagnostic process by submitting bad tomographic data.

5.3.3 Accuracy of Fault Accusations

Accurate fault accusation requires accurate tomography. Duffield *et al* reported high levels of accuracy for striped unicast probing, with inferred link loss rates within 1% of the actual ones [41]. High accuracy rates have also been reported for other tomographic techniques [71]. In this section, we assume that hosts can identify whether a link was up or down with 90% accuracy.

Given the probe accuracy, we are interested in the amount of blame assigned to a forwarding peer when a message is dropped. Figure 5.5 depicts the probability distribution functions for the blame that Concilium assigns to faulty and non-faulty

nodes. We generated the pdf by taking each triple of hosts (A, B, C) [2] and picking ten random times within the simulation period for A to route a message through $B \rightarrow C$. By comparing the actual link state along $B \rightarrow C$ to the tomographic information available to A at that time, we determined the amount of blame that A would assign to B if A did not receive an acknowledgment from the message recipient. B was a faulty node if it dropped a message despite $B \rightarrow C$ being good; it was non-faulty if at least one link in $B \rightarrow C$ was bad. Due to space constraints, we do not show results for the recursive revision of accusations; thus, the simulator ensured that a message was dropped either by B or a network link along $B \rightarrow C$, not by another peer or link further down the overlay route to the destination host.

Figure 5.5(a) depicts the blame pdf when all peers faithfully reported their probe results. Figure 5.5(b) depicts the blame pdf when 20% of peers colluded to maliciously flip their probe results. In the latter scenario, when a non-faulty node was being judged, malicious peers would always claim that their probed links were up (increasing the false positive rate); when a malicious peer was being judged, other malicious peers would always claim that their probed links were down (increasing the false negative rate). Comparing Figure 5.5(a) to Figure 5.5(b), we see that incorporating erroneous probe results into Equation 5.2 causes more blame to be assigned to non-faulty nodes and less blame to be assigned to faulty ones. However, Concilium can still make accurate fault accusations using a thresholding scheme which produces binary verdicts. For example, suppose that for any message drop, nodes receiving less than 40% blame are proclaimed innocent and all other nodes receive a guilty verdict. If all peers report their probe results faithfully, then innocent peers will receive guilty verdicts 1.8% of the time whereas faulty peers will receive guilty verdicts 93.8% of the time. If 20% of peers collude and contribute malicious probe results, then innocent peers will receive guilty verdicts 8.4% of the time and faulty peers will receive guilty verdicts 71.3% of the time.

A host issues a formal accusation against a peer if that peer accumulates at least m guilty verdicts for the w most recent message drops. To determine the false positive and false negative rates of formal accusations, let p_{good} be the probability that a non-faulty node receives a guilty verdict for a message drop, and p_{faulty} be the probability that a faulty node receives a guilty verdict; these probabilities are derived from the blame pdfs and thresholds as described in the previous paragraph. Let W be a random variable describing the number of guilty verdicts in a w-slot window.

[2] This selection was constrained by the routing tables of each node, i.e., B had to be in A's routing table and C had to be in B's routing table.

W is a binomial random variable, meaning that the error rates can be described as follows:

$$\begin{aligned} Pr(false\ positive) &= Pr(W \geq m) \\ &= \sum_{k=m}^{w} \binom{w}{k} p_{good}^{k} (1 - p_{good})^{w-k} \end{aligned}$$

$$\begin{aligned} Pr(false\ negative) &= Pr(W < m) \\ &= \sum_{k=0}^{m-1} \binom{w}{k} p_{faulty}^{k} (1 - p_{faulty})^{w-k}. \end{aligned}$$

Figure 5.6 depicts the error rates with a blame pdf threshold of 40% and a sliding window size of 100. If all nodes faithfully report probe results, then we can drive both error rates below 1% with an m of 6. If 20% of hosts maliciously invert their probe results, we can achieve equivalent error rates with an m of 16.

5.3.4 Bandwidth Requirements

Concilium has two primary sources of network overhead. Peers must exchange signed, timestamped copies of their routing state, and they must perform tomographic probing. We expect local routing state to reference $\mu_\phi + 16$ peers, where 16 is the number of leaf nodes. Each routing entry contains a 16 byte node identifier and a 4 byte freshness timestamp. Using PSS-R [13] with 1024 bit public keys, both quantities plus a signature consume 144 bytes. The exchanged routing state also includes tomographic probe results for the IP path to each routing peer. As explained in Section 5.2.2, the results for each path can be encoded in a few bits. Assuming 1 byte for each path summary and a 100,000 node overlay, an entire advertised routing table is about 11.5 kilobytes. This overhead can be decreased by sending diffs for updated entries instead of entire tables.

In the absence of forwarding faults, lightweight tomography requires no additional bandwidth beyond that already required for availability probing. The outgoing bandwidth required for heavyweight striped probing of a tree is

$$\binom{|leaves \in T_H|}{2} (stripes\_per\_pair)(stripe\_size)(pkt\_size).$$

In a 100,000 node overlay, the average node has 77 entries in its local routing state. Suppose that each node sends 100 stripes to each ordered pair of peers, that each stripe contains two UDP probes, and that each probe is 30 bytes long (28 bytes for

IP+UDP headers and 16 bits for a nonce). Probing an entire tree will require 16.7 MB of outgoing network traffic. Incoming probes will require no more than this amount and less if there are legitimately lossy network links.

The probing cost can be reduced in several ways. If IP multicast were widely deployed, we could reduce the probe traffic sent from the root of a tree to its leaf nodes. Also, as described in Section 5.2.8, cooperative hosts on the same stub network can share probe results, reducing the probing bandwidth for the collective.

5.4 Conclusions

A reliable distributed system must be able to pinpoint its causes of failure, both to reconfigure itself around problems and to notify failed components of their status. Since problems can occur at various levels in the software stack, each abstraction layer must be able to reason about its particular failure modes. Overlay middleware provides the routing substrate for many distributed systems. Thus, if these systems want to be robust, they must use a diagnosable overlay which can identify the faulty IP links and end hosts within the communication network.

In this chapter, we introduced Concilium, a distributed diagnostic protocol for overlay networks. By aggregating peer-advertised routing state, Concilium determines forwarding paths at the overlay level. Using collaborative network tomography, Concilium discovers the IP links which comprise these paths and the quality of these links. By combining the topological and tomographic data with application-level message acknowledgments, Concilium judges whether dropped overlay messages are due to failures in the core Internet or failures in overlay forwarders. Concilium's fault accusations are self-verifying and robust to tampering, but they may place blame on nodes which are the victim of misbehavior further downstream in their routes. Thus, Concilium provides mechanisms to revise such incorrect accusations. It also has methods for detecting peers which publish faulty routing state or tomographic data.

CHAPTER VI

Related Work

This chapter provides an overview of prior research into host availability. We first discuss empirical availability studies and the mathematical models inspired by these works. We then examine the few distributed systems which have tackled availability introspection in a serious manner. We conclude with a survey of the network monitoring tools that relate to StrobeLight and Concilium.

6.1 Empirical Availability Studies

There are many observational studies of availability in distributed systems. Most of them used active network probing to detect uptime changes. For example, Bolosky *et al* described the uptimes of over 50,000 PCs belonging to the Microsoft Corporation [21]. Saroiu *et al* studied Napster and Gnutella, popular peer-to-peer file trading services [95]. Douceur performed a meta-analysis of availability data [37], examining the Microsoft, Gnutella, and Napster traces, as well as a trace from a sampling of global Internet hosts [68]. Douceur posited two broad classes of machine availability; those in the first are almost always online, whereas those in the second have diurnal uptime periods. Bhagwan *et al* studied nodes in the Overnet DHT and also found diurnal uptime patterns [17]. In Chapter II I extended these binary categorizations, providing a richer taxonomy of uptime classes.

Rather than rely on coarse-grained probing, other studies have used operating system logs to infer downtime. For example, Simache's analysis of a Unix workstation cluster found a median downtime of 38.5 minutes [98]; roughly 35% of reboots were caused by 10% of the machines, primarily between 8AM and 6PM.

6.2 Analytical Models of Availability

Analytic models of cooperative systems typically include a parameter for host availability. Douceur and Wattenhofer expressed a machine's availability in terms of its fractional downtime [40]. To account for time-of-day effects on global aggregate availability, Bhagwan *et al* assumed a pessimistic mean availability based only on hosts online at night [16]. Blake and Rodrigues also used conservative, average-based metrics in their model [20].

In the computational grid community, availability prediction is done by fitting empirically observed uptime traces to well-known statistical distributions such as the Pareto or Weibull distribution [80]. Using the derived model parameters, one can estimate how long a random machine will be online before failing. These approaches suffer from the same pitfalls as worst-case availability estimations, since coarse-grained statistics are not useful for per-host characterization and forecasting.

6.3 Availability-aware Distributed Systems

Overlays like Pastry [91] and Chord [101] use an opt-in model for participation—each host may join and leave the network as it pleases. To detect node exits, each host periodically probes the members of its routing table. This probing rate is typically chosen to guarantee a minimal level of table consistency in the face of worst-case churn rates.

Mahajan *et al* showed how overlay nodes can estimate the mean session time in the network by observing the churn rate in their routing table [70]. Machines can then reduce the rate at which they probe each other while maintaining the same level of consistency in their routing tables. This technique provides a substantial decrease in control traffic; however, the mean session time of the network at a particular moment cannot be used to predict the availability of individual machines. As shown in Chapter II, my forecasting techniques enable more dramatic decreases in control traffic.

Schwarz *et al* proposed a distributed object store which biases data storage towards peers with high predicted availability [96]. Each node has a counter which is initialized to 0. During a periodic system-wide scan, a node's counter is incremented by 1 if it is online, otherwise the counter is decremented by 1. Data storage is biased towards nodes with high counter values. Such a system will correctly identify consistently online nodes as good storage hosts and consistently offline hosts as poor replica sites. However, the counter mechanism is too crude to capture more complex

patterns. For example, if a host has diurnal availability, it will be online for the longest consecutive stretch starting in the morning, when its counter is lowest. This would be the *best* time to transfer data to the node, not at the end of the work day, when the counter is high but the node is about to go offline. The DHT of Chapter III does not encounter this pitfall—its predictors will forecast that diurnal nodes will have low availability at night, making these nodes unattractive storage candidates as the work day draws to a close.

TotalRecall is a peer-to-peer storage system which adjusts replication strategies to meet specified data availability goals [18]. It adapts to changing file workloads and node churn, providing two methods for maintaining data redundancy. In eager repair, the system reacts to a host going down by replicating its data elsewhere. With lazy repair, the system estimates worst-case host availability using Bhagwan's conservative estimates [16]; it then over-replicates data such that redundancy targets are still met in these worst case scenarios. Lazy repair trades increased disk requirements for reduced bandwidth. Using our availability predictors, we can reduce both bandwidth and disk consumption. By identifying nodes that are highly available and biasing data storage towards them, we decrease on-demand object regeneration while reducing the need for over-replication.

6.4 Network Monitoring Tools

Both StrobeLight (Chapter IV) and Concilium V use active probing to measure availability and detect anomalous network conditions. In the text below, we discuss prior research in path estimation, fault diagnosis, and anomaly detection.

6.4.1 Measuring Path Quality

Chapter IV described StrobeLight, an active probing system for monitoring enterprise-level availability and detecting routing anomalies. There are several other tools which use active probing to infer path characteristics like latency or bandwidth. For example, RON uses special gateways to probe the paths between stub networks, forwarding packets along gateway-level overlay routes if the standard IP routes have poor quality [5]. Duffield *et al* use the end-to-end loss rates at the leaves of a multicast tree to deduce loss rates for internal IP links [41]. These projects focus on measuring path quality between a small set of endpoints that are known to be online. In contrast, StrobeLight seeks to measure the availability of hundreds of thousands of individual hosts that may or may not be online.

Passive introspection of preexisting traffic can also be used to infer path characteristics. For example, Padmanabhan *et al* record the end-to-end loss rate inside a client-server flow and use Bayesian statistics to extrapolate loss rates for interior IP links [82]. The iPlane system [69] performs active probing between carefully chosen vantage points, but it also runs BitTorrent [33] clients on PlanetLab [12] hosts, extracting passive bandwidth measurements along the paths to the random BitTorrent peers.

Passive probing is attractive for two reasons. First, it does not generate new traffic which might interfere with preexisting network flows. Second, explicit probing traffic may trigger intrusion detection systems deployed on leaf networks, a problem observed with active probing systems deployed on PlanetLab [100]. Despite these advantages, passive probing was ill-suited for StrobeLight's goal of tracking per-host availability in a large network. The time that a host is online is a superset of the time that it is generating network traffic, so passive observations of per-host packet flows may underestimate true availability. Also, a key design goal was to minimize the new infrastructure that had to be pushed to end hosts or the corporate routing infrastructure. Installing custom network introspection code on every end host was infeasible. Placing such code inside the core network infrastructure was also untenable due to the complex web of proxies, firewalls, and routers that would have to be instrumented to get a full view of each host's network activity. Indeed, understanding this complex web of dependencies is a full research project by itself [10]. StrobeLight avoids these problems using a centralized, active prober which is trusted by our network IDS.

6.4.2 Detecting IP Hijacks

Most prior work on IP hijack detection has required modification to core Internet routers. Some systems require routers to perform cryptographic operations to validate BGP updates [3, 23, 56], whereas others require changes to router software to make BGP updates more robust to tampering [47, 103, 114]. We eschewed such designs due to the associated deployment problems.

Several systems use passive monitoring of routing dynamics to detect inconsistencies in global state [66, 67, 97]. These systems typically search for anomalies in one or more publicly accessible databases such as RouteViews [81], which archives BGP state from multiple vantage points, or the Internet Routing Registry [4], which contains routing policies and peering information for each autonomous system. Passive monitoring eases deployability concerns. However, data freshness becomes a

concern when dealing with "eventually updated" repositories such as the IRR, and even RouteViews data is only updated once every two hours. Legitimate changes to routing policy may also be indistinguishable from hijacking attacks in terms of BGP semantics, making disambiguation difficult in some cases. In contrast, if our availability fingerprints indicate that a large chunk of hosts have suddenly gone offline or changed their availability profile, it is extremely unlikely that this is a natural network phenomenon.

Hu and Mao were the first to use data plane fingerprints in the context of hijack detection [55]. In their system, a live BGP feed is monitored for suspicious updates. If an IP prefix is involved in a questionable update, its hosts are scanned from multiple vantage points using nmap OS fingerprinting [46], IP ID probing [15], and ICMP timestamp probing [55]. The results are presented to a human operator who determines if they are inconsistent. Our system differs in three ways. First, we do not require privileged access to a live BGP feed, easing deployability. Second, we continually calculate subnet fingerprints, whereas Hu's system only calculates fingerprints upon detecting suspicious BGP behavior, behavior which may take several minutes to propagate to a particular vantage point. Third, we can finish a probing sweep in less than 30 seconds, whereas several of Hu's scans may take several minutes to complete. Given the short-lived nature of spectrum agility attacks [90], we believe that quick, frequent scanning is preferable, if only to serve as a tripwire to trigger slower, "deeper" scans.

Zheng *et al* detect hijacking attacks by measuring the hop count from monitor hosts to the IP prefixes of interest [115]. For each prefix, the monitor selects a reference point that is topologically close to the prefix and lies along the path from the monitor to the prefix. In normal situations, the hop count along the monitor-reference point path should be close to that of the monitor-prefix path. When the prefix is hijacked, the hop count along the two paths should diverge. Zheng's system avoids the deployability problems mentioned above, since hop counts can be determined by any host that can run traceroute. However, the system assumes that a reference point can be found which is immediately connected to the target prefix and responds to ICMP messages; if the reference point is further out, the hijacker can hide within the extra hops. Our system only requires that end hosts respond to pings. Furthermore, our system tracks the availability of individual hosts, whereas Zheng's system only tracks the availability of a few representative hosts in each target prefix.

6.4.3 Diagnosing Message Drops

Concilium attempts to locate routing faults at both the overlay and the IP level. Several other systems try to pinpoint problems in the routing substrate. Packet obituary systems [7] allow end hosts to determine the autonomous system (AS) which dropped a particular packet. Each AS deploys an "accountability box" at each border link. When an incoming packet hits a box, the box records the next AS that the packet will traverse. Boxes periodically push these records along the reverse box paths, allowing each packet source to determine the last AS which successfully received their datagrams. Concilium differs from obituary systems in three ways. First, Concilium does not require the modification of core Internet routers. Second, Concilium protects and validates its network data using various cryptographic and statistical techniques. Finally, obituary systems cannot arbitrate between two adjacent ASes when the first claims that the second dropped its packet, and the second claims that the first never sent the packet. Concilium resolves such disputes using reputation systems.

Concilium assumes that end hosts may be malicious but core routers will not fail in a byzantine way. Fatih [75] is designed to detect core routers which maliciously drop or reorder packets. Each router maintains a summary of the traffic it has forwarded. Signed versions of these summaries are periodically exchanged with other routers, and misbehavior is detected by comparing summaries from routers that share links. Like obituary systems, Fatih requires modification to core Internet infrastructure.

In RON [5], each stub network has a special gateway which sits between the stub and the larger Internet. The RON gateways monitor the loss, latency, and throughput along the $O(N^2)$ paths which connect them. When a gateway must forward a locally generated packet outside its stub, it forwards the message through other RON gateways if the default IP path is poor. Like Concilium, RONs use active probing to detect link quality. The key difference is that RON always ascribes blame to the network—misbehaving RON nodes must be detected and removed by human operators. Concilium provides a mechanism for blaming the network *or* an overlay node.

CHAPTER VII

Conclusions

Software systems are becoming more difficult to understand. Part of the difficulty arises from the sheer amount of code and configuration state that resides on an individual machine; explaining the behavior of a standalone desktop application often requires a sprawling narrative spanning millions of lines of code written by multiple developers at different times. This narrative becomes even more complicated for distributed applications. Semantically related state is often scattered across many machines. At any given time, some of these machines may be powered off, disconnected from various parts of the network, or engaged in activities that are not the ones we wish to encourage. Indeed, hosts may be misconfigured, selfish, or outright malicious. These problems are particularly relevant in cooperative systems. In these environments, hosts can enter and leave the collective at will, and there are weak (or non-existent) reputation checks on the nodes which volunteer their services.

At first glance, such an unreliable computing base seems like an unattractive substrate for distributed computing. However, pushing server tasks onto the unwashed masses has a singular advantage: most commodity PCs have substantial spare resources, and the aggregate amount of these resources grows as the number of commodity PCs grows. Thus, cooperative computing offers the possibility for massively scalable services. The key challenge is to design these services in a way that maximizes performance and robustness in the face of a constantly churning, possibly untrustworthy server set.

My dissertation examines one aspect of this challenge: fluctuating host availability. Historically, cooperative systems have viewed churn as an enemy of robustness. This perspective has led to extremely pessimistic designs. For example, the first overlay-based storage systems regenerated a replica's objects as soon as the replica went offline [101, 91]. This policy was driven by the fear that if regeneration were delayed, the rest of an object's replicas might go offline in the interim, leading to

the permanent loss of the object. Later systems took less aggressive approaches, but their strategies were still driven by pessimistic analyses of availability. For example, in the TotalRecall system [18], objects are not automatically regenerated when a replica goes offline. Instead, TotalRecall "over-replicates" each object beyond the required redundancy target, allowing this target to be met during worst-case global availability. This policy saves a large amount of data copying, but the penalty is an increased storage requirement.

My dissertation provides a key insight: *despite our lack of control over host availability, we can still make predictions about it.* Individual users often operate their machines in habitual ways, and seemingly uncoordinated users may be driven by the same exogenous force, e.g., the diurnal work cycle. This dissertation proves that many networks contain such regular uptime patterns, and that these patterns can be exploited by introspective distributed systems.

In peer-to-peer overlays and storage systems, much of the design effort is devoted to coping with churn. Thus, these applications are the most obvious beneficiaries of availability prediction. Using these forecasts, overlays can maintain consistent routing tables with less probing traffic, and DHTs can provide higher data availability while using less regeneration bandwidth. However, availability introspection is useful in domains besides peer-to-peer computing. For example:

- In delay-tolerant networks, the routing substrate is a fluid, time-sensitive graph that depends on ephemeral pairwise connections between devices; using availability prediction to guess when these contacts will take place, we can select better routes and reduce end-to-end message latencies.

- A deep knowledge of availability patterns makes it easier for network operators to detect routing anomalies. Each subnet contains a unique distribution of availability behaviors. This unique distribution allows subnets to be fingerprinted using lightweight probe sweeps. These snapshots of instantaneous availability have two important properties. First, at any given time, each subnet's fingerprint is extremely likely to be unique. Second, an individual subnet's fingerprint evolves slowly over time. Thus, if a subnet's fingerprint undergoes significant, rapid change, it is likely the victim of a routing attack or misconfiguration. Evaluation of a deployed fingerprinting system shows that the probing burden is minimal and the anomaly detection is fast and accurate.

- Incorporating availability models into epidemiological frameworks leads to better predictions of malware spread. Unfortunately, these improved models can

be used by malicious parties to pick the most damaging time to launch a worm.

These examples show that fluctuating availability is an important phenomena in a wide variety of settings. By applying a deep understanding of host behavior, we can unearth previously hidden opportunities for improving performance, reliability, and security.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. Adya, W. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. Douceur, J. Howell, J. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of OSDI*, pages 1–14, Boston, MA, December 2002.

[2] J. Agosta, C. Diuk-Wasser, J. Chandrashekar, and C. Livadas. An Adaptive Anomaly Detector for Worm Detection. In *Proceedings of SysML*, Cambridge, MA, April 2007.

[3] W. Aiello, J. Ioannidis, and P. McDaniel. Origin Authentication in Interdomain Routing. In *Proceedings of CCS*, Washington, DC, October 2003.

[4] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language (RPSL). RFC 2622, June 1999.

[5] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of SOSP*, pages 131–145, Banff, Canada, October 2001.

[6] D. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceedings of the IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, November 2004.

[7] K. Argyraki, P. Maniatis, D. Cheriton, and S. Shenker. Providing Packet Obituaries. In *Proceedings of ACM SIGCOMM HotNets*, San Diego, CA, November 2004.

[8] V. Arya, T. Turletti, and C. Hoffmann. Feedback Verification for Trustworthy Tomography. In *Proceedings of IPS-MoMe*, Warsaw, Poland, March 2005.

[9] A. Avizeinis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions of Software Engineering*, SE-11(12):1491–1501, December 1985.

[10] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services via Inference of Multi-level Dependencies. In *Proceedings of SIGCOMM*, pages 13–24, Kyoto, Japan, August 2007.

[11] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of NDSS*, San Diego, California, February 2005.

[12] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of NSDI*, San Francisco, CA, March 2004.

[13] M. Bellare and P. Rogaway. The Exact Security of Digital Signatures – How to Sign with RSA and Rabin. *Advances in Cryptology–EUROCRYPT '96*, 1070:399–416, 1996.

[14] R. Bellman and M. Giertz. On the analytic formalism of the theory of fuzzy sets. *Information Sciences*, 5:149–156, 1973.

[15] S. Bellovin. A Technique for Counting NATted Hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop*, pages 267–272, Marseille, France, November 2002.

[16] R. Bhagwan, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer systems. Technical Report CS2002-0726, UCSD, November 2002.

[17] R. Bhagwan, S.Savage, and G. Voelker. Understanding Availability. In *Proceedings of the 2nd IPTPS*, Berkeley, CA, February 2003.

[18] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proceedings of NSDI*, March 2004.

[19] J. Binkley and S. Singh. An Algorithm for Anomaly-based Botnet Detection. In *Proceedings of SRUTI*, pages 43–48, San Jose, CA, July 2006.

[20] C. Blake and R. Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *Proceedings of the 9th HotOS*, May 2003.

[21] W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *Proceedings of ACM SIGMETRICS*, Santa Clara, CA, June 2000.

[22] W. Bolosky, J. Douceur, and J. Howell. The Farsite Project: A Retrospective. *ACM SIGOPS Operating Systems Review*, 41(2):17–26, April 2007.

[23] K. Butler, P. McDaniel, and W. Aiello. Optimizing BGP Security by Eploiting Path Stability. In *Proceedings of CCS*, pages 298–310, Alexandria, VA, November 2006.

[24] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of DSN*, Florence, Italy, June 2004.

[25] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI*, pages 299–314, Boston, MA, December 2002.

[26] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, 2003.

[27] D.-F. Chang, R. Govindan, and J. Heidemann. Locating BGP Missing Routes Using Multiple Perspectives. In *Proceedings of the SIGCOMM Workshop on Network Troubleshooting*, pages 301–306, Portland, OR, September 2004.

[28] K. Chen, L. Naamani, and K.Yehia. miChord: Decoupling Object Lookup from Placement in DHT-Based Overlays. http://www.loai-naamani.com/Academics/miChord.htm.

[29] Y. Chen, D. Bindel, H. Song, and R. Katz. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proceedings of ACM SIGCOMM*, pages 55–66, Portland, OR, September 2004.

[30] B. Chun, J. Lee, and H. Weatherspoon. Netbait: a Distributed Worm Detection Service. Intel Research Berkeley Technical Report IRB-TR-03-033, September 2003.

[31] P. Cincotta, M. Mendez, and J. Nunez. Astronomical Time Series Analysis I: A Search for Periodicity Using Information Entropy. *The Astrophysical Journal*, 449:231–235, August 1995.

[32] Cisco Systems. *Cisco SMB Class Solutions: Reliability for Your Phone System*, 2004. White paper.

[33] B. Cohen. Incentives Build Robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, pages 251–260, Berkeley, CA, June 2003.

[34] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of SRUTI*, pages 39–44, Cambridge, MA, July 2005.

[35] L. Cox, C. Murray, and B. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of OSDI*, pages 285–298, December 2002.

[36] D. Dagon, C. Zou, and W. Lee. Modeling Botnet Propagation Using Time Zones. In *Proceedings of the 13th NDSS*, San Diego, CA, February 2006.

[37] J. Douceur. Is remote host availability governed by a universal law? *SIGMETRICS Performance Evaluation Review*, 31(3):25–29, 2003.

[38] J. Douceur and W. Bolosky. A Large-Scale Study of File-System Contents. In *Proceedings of ACM SIGMETRICS*, Atlanta, GA, May 1999.

[39] J. Douceur and R. Wattenhofer. Large-Scale Simulation of Replica Placement Algorithms for a Serverless Distributed File System. In *Proceedings of the 9th MASCOTS*, pages 311–319, Cincinnati, OH, August 2001.

[40] J. Douceur and R. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proceedings of the 20th IEEE SRDS*, pages 4–13, New Orleans, LA, October 2001.

[41] N.G. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring Link Loss Using Striped Unicast Probes. In *Proceedings of IEEE INFOCOM*, pages 915–923, Anchorage, AK, April 2001.

[42] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of ACM SIGCOMM*, pages 27–34, August 2003.

[43] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of HotOS*, pages 67–72, Cape Cod, MA, May 1997.

[44] J. Frauenthal. *Mathematical Modeling in Epidemiology.* Springer-Verlag, New York, NY, 1980.

[45] M. Freedman, K. Lakshminarayanan, S. Rhea, and I. Stoica. Non-Transitive Connectivity and DHTs. In *Proceedings of WORLDS*, pages 55–60, San Francisco, CA, December 2005.

[46] Fyodor. nmap security scanner. http://insecure.org/nmap/.

[47] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working Around BGP: An Incremental Approach to Improving Security and Accuracy of Interdomain Routing. In *Proceedings of NDSS*, San Diego, CA, February 2003.

[48] R. Govindan and H. Tangmunarunkit. Heuristics for Internet Map Discovery. In *Proceedings of IEEE INFOCOM*, pages 1371–1380, Tel Aviv, Israel, March 2000.

[49] IETF IDR Working Group. A Border Gateway Protocol 4 (BGP-4). RFC 1771, March 1995.

[50] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proceedings of IPTPS*, Santa Barbara, CA, February 2006.

[51] S. Haykin. *Adaptive Filter Theory.* Prentice Hall, Englewood Cliffs, NJ, 3^{rd} edition, 1996.

[52] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. In *ACM Transactions on Computer Systems*, pages 51–81, February 1988.

[53] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[54] T. Howes. The Lightweight Directory Access Protocol: X.500 Lite. Technical Report 95-8, CITI (University of Michigan), July 1995.

[55] X. Hu and Z. Morley Mao. Accurate Real-time Identification of IP Prefix Hijacking. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 3–17, Oakland, California, May 2007.

[56] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure Path Vector Routing for Securing BGP. In *Proceedings of SIGCOMM*, pages 179–192, Portland, OR, September 2004.

[57] Information Sciences Institute. Internet Protocol. RFC 791, September 1981.

[58] International Business Machines Corporation, S/390 Division. *Five Nines / Five Minutes: Achieving Near Continuous Availability*, 1999. White paper.

[59] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *Proceedings of ACM SIGCOMM*, pages 145–158, September 2004.

[60] J. Jung and E. Sit. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of IMC*, pages 370–375, Taormina, Sicily, Italy, October 2004.

[61] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. Voelker. Surviving Internet Catastrophes. In *Proceedings of the USENIX Technical Conference*, pages 45–60, Anaheim, CA, May 2005.

[62] R. Jurgelenaite, P. Lucas, and T. Heskes. Exploring the noisy threshold function in designing bayesian networks. In *Proceedings of SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 133–146, Cambridge, UK, December 2005.

[63] J. Kephart and S. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Computer Symposium on Research in Security and Privacy*, pages 343–359, May 1991.

[64] M. Kim and D. Kotz. Modeling users' mobility among WiFi access points. In *Proceedings of the International Workshop on Wireless Traffic Measurements and Modeling*, pages 19–24, Seattle, WA, June 2005.

[65] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[66] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Topology-based Detection of anomalous BGP messages. In *Proceedings of RAID*, Pittsburgh, PA, September 2003.

[67] M. Lad, D. Massey, D. Pei, Y. Wu, B. Zhang, and L. Zhang. Phas: A Prefix Hijack Alert System. In *Proceedings of USENIX Security*, pages 153–166, Vancouver, Canada, August 2006.

[68] D. Long, A. Muir, and R. Golding. A Longitudinal Survey of Internet Host Reliability. In *Proceedings of the 14th IEEE SRDS*, 1995.

[69] H. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iplane: An Information Plane for Distributed Services. In *Proceedings of OSDI*, pages 367–380, Seattle, WA, November 2006.

[70] R. Mahajan, M. Castro, and A. Rowstron. Controlling the Cost of Reliability in Peer-to-Peer Overlays. In *Proceedings of the 2nd IPTPS*, Berkeley, CA, February 2003.

[71] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. In *Proceedings of SOSP*, pages 106–119, Lake George, NY, October 2003.

[72] S. McFarling. Combining branch predictors. Technical Note TN-36, DEC WRL, June 1993.

[73] N. Merhav and M. Feder. Universal Prediction. *IEEE Transactions on Information Theory*, pages 2124–2147, October 1998.

[74] J. Mickens and B. Noble. Exploiting Availability Prediction in Distributed Systems. In *Proceedings of NSDI*, pages 73–86, San Jose, CA, May 2006.

[75] A. Mizrak, Y.-C. Cheng, K. Marzullo, and S. Savage. Fatih: Detecting and Isolating Malicious Routers. In *Proceedings of DSN*, pages 538–547, Yokohama, Japan, June 2005.

[76] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1:33–39, July 2003.

[77] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of ACM Internet Measurement Workshop*, pages 273–284, Marseille, France, November 2002.

[78] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial-of-Service Activity. In *Proceedings of USENIX Security*, pages 9–22, Washington, DC, August 2001.

[79] C. Neuman, J. Schiller, and J. Steiner. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of USENIX Winter Conference*, Dallas, TX, February 1988.

[80] D. Nurmi, J. Brevik, and R. Wolski. Modeling Machine Availability in Enterprise and Wide-area Distributed Computing Environments. In *Proceedings of EUROPAR*, 2005.

[81] University of Oregon. *Route Views Project*. http://www.routeviews.org.

[82] V. Padmanabhan, L. Qiu, and H. Wang. Passive Network Tomography Using Bayesian Inference. In *Proceedings of SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002.

[83] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing Streaming Media Content Using Cooperative Networking. In *Proceedings of NOSSDAV*, pages 177–186, May 2002.

[84] K. Park and V. Pai. CoMon: A mostly-scalable monitoring system for PlanetLab. *Operating Systems Review*, 40(1):65–74, January 2006.

[85] R. Pastor-Satorras and A. Vespignani. Epidemic Spreading in Scale-Free Networks. *Physics Review Letters*, 86(14):3200–3203, April 2001.

[86] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD*, pages 109–116, 1988.

[87] S. Pincus. Approximate entropy as a measure of system complexity. In *Proceedings of the National Academy of Science*, pages 2297–2301, USA, March 1991.

[88] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C*. Cambridge University Press, New York, NY, 2^{nd} edition, 1992.

[89] P.Tino and G. Dorffner. Predicting the future of discrete sequences from fractal representations of the past. *Machine Learning*, 45(2):187–218, 2001.

[90] A. Ramachandran and N. Feamster. Understanding the Network-Level Behavior of Spammers. In *Proceedings of SIGCOMM*, pages 291–302, Pisa, Italy, September 2006.

[91] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, November 2001.

[92] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th ACM SOSP*, pages 188–201, October 2001.

[93] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Technical Conference*, pages 119–130, Portland, OR, 1985.

[94] S. Saroiu, K. Gummadi, R. Dunn, S. Gribble, and H. Levy. An analysis of internet content delivery systems. In *Proceedings of OSDI*, pages 315–327, December 2002.

[95] S. Saroiu, P. Gummadi, and S. Gribble. A Measurement Study of Peer-to-peer File Sharing Systems. In *Proceedings of the Multimedia Computing and Networking Conference*, January 2002.

[96] T. Schwarz, Q. Xin, and E. Miller. Availability in Global Peer-To-Peer Storage Systems. In *Proceedings of the 2nd IPTPS*, Lausanne, Switzerland, July 2004.

[97] G. Siganos and M. Faloutsos. Neighborhood Watch for Internet Routing: Can We Improve the Robustness of Internet Routing Today? In *Proceedings of INFOCOM*, pages 1271–1279, Anchorage, AK, May 2007.

[98] C. Simache and M. Kaaniche. Measurement-based Availability Analysis of Unix Systems in a Distributed Environment. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 346–355, Hong Kong, China, November 2001.

[99] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proceedings of ACM SIGCOMM*, pages 133–145, Pittsburgh, PA, August 2002.

[100] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using Planetlab for Network Research: Myths, Realities, and Best Practices. In *Proceedings of WORLDS*, pages 67–72, San Francisco, CA, December 2005.

[101] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A Scalabale Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, pages 149–160, San Diego, CA, August 2001.

[102] J. Stribling. All-pairs PlanetLab Ping Data. http://www.pdos.lcs.mit.edu/~strib/pl_app/.

[103] L. Subramanian, V. Roth, I. Stoica, S. Shenker, and R. H. Katz. Listen and Whisper: Security Mechanisms for BGP. In *Proceedings of NSDI*, pages 127–140, San Francisco, CA, March 2004.

[104] G. Sugihara and R. M. May. Nonlinear forecasting as a way of distinguishing chaos from measurement error in time series. *Nature*, 344(6268):734–741, April 1990.

[105] Symantec Corporation. *Symantec Internet Security Threat Report: Trends for January 06–June 06*, September 2006.

[106] R. Teixeira, S. Agarwal, and J. Rexford. BGP Routing Changes: Merging Views from Two ISPs. In *SIGCOMM Computer Communications Review*, pages 79–82, October 2005.

[107] K. Walsh and E. G. Sirer. Experience with an Object Reputation System for Peer-to-Peer Filesharing. In *Proceedings of NSDI*, pages 1–14, San Jose, CA, May 2006.

[108] Y. Wang, D. Chakrabarti, C. Wang, and C. Faloutsos. Epidemic Spreading in Real Networks: An Eigenvalue Viewpoint. In *Proceedings of the Symposium on Reliable Distributed Computing*, pages 25–34, Florence, Italy, October 2003.

[109] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of USENIX Security Symposium*, pages 29–44, San Diego, CA, August 2004.

[110] M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proceedings of ACSAC*, pages 61–68, Las Vegas, NV, December 2002.

[111] Rich Wolski. Experiences with predicting resource performance on-line in computational grid settings. *SIGMETRICS Performance Evaluation Review*, 30(4):41–49, 2003.

[112] Zattoo. *Zattoo: TV to Go*. http://zattoo.com/.

[113] Y. Zhang, V. Paxson, and S. Shenker. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical Report, AT&T Center for Internet Research at ICSI, May 2000.

[114] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. Detection of Invalid Routing Announcement in the Internet. In *Proceedings of DSN*, pages 59–68, Bethesda, MD, June 2002.

[115] C. Zheng, L. Ji, D. Pei, J. Wang, and P. Francis. A Light-Weight Distributed Scheme for Detecting IP Prefix Hijacks in Real-Time. In *Proceedings of SIGCOMM*, pages 277–288, Kyoto, Japan, August 2007.