Intrusion-Tolerant Membership Management for Peer-to-Peer Overlay Networks

Håvard Dagenborg Johansen

A dissertation for the degree of Philosophiae Doctor

UNIVERSITY OF TROMSØ
Faculty of Science
Department of Computer Science

November 2007

Copyright © 2007 Håvard Dagenborg Johansen

ISBN 978-82-92461-69-3

Abstract

Peer-to-Peer (P2P) computing has emerged as a cost-effective approach for constructing large-scale wide-area Internet services. Many generic P2P middleware platforms, known as overlay networks, do already exist. Because overlay networks are deployed on untrusted hardware, an attacker can gain control of internal system components simply by joining. Consequently, overlay networks should be constructed such that they are not easily corrupted by maliciously induced Byzantine faults. For this, overlay networks must implement mechanisms for intrusion-tolerant membership management.

This dissertation describes *Fireflies*, a novel intrusion-tolerant membership management protocol. *Fireflies* fights membership attacks by organizing members in a strict and verifiable pseudo-random structure such that an attacker can not falsely modify the membership views of correct members. By providing each correct member with an up-to-date view of all members, *Fireflies* avoids the overhead associated with multi-hop routing in strict overlay-network structures. This dissertation also describes FiRe, an overlay-network framework that implements the *Fireflies* protocol. As a case study on the applicability of our solution, this dissertation describes FirePatch, a novel secure dissemination network for software patches. FirePatch enables software vendors and end-users to fight hackers that reverse-engineer software security patches into automated exploits.

We evaluate our findings using both simulations and PlanetLab. Our solution provides a novel tradeoff between scalability and intrusion tolerance and is applicable in many overlay networks.

Acknowledgements

I am grateful for all those that have provided me with support and inspiration through the strenuous course of completing this dissertation. Family, friends, and colleagues have all made their contributions.

In particular, I would like to thank the following: Prof. *Dag Johansen* for being my advisor, for providing insight, motivation, and inspiration, and for authoring three papers with me [77, 80, 81]; Principal Research Scientist *Robbert van Renesse* at Cornell University for collaborating and supervising me on the problems and solutions that form this dissertation, for authoring three papers with me [77, 79, 80], and for helping me out in every other way; Dr. *André Allavena* for working with me on the paper that forms the core of this dissertation [79]; Prof. *Fred B. Schneider* for inviting me to work at Cornell University for a year, which had a profound impact on this dissertation; and *Åge Kvalnes* for reading the drafts of this dissertation and providing me with valuable feedback. Special thanks to my ever so patient girlfriend *Leila I. Johansen* for her faithful loving support.

This work has been founded by the Research Council of Norway IKT 2010 program. The material needed for my daily activates, including an office to work in, a computer to work on, and administrative support, have been provided by the Department of Computer Science, University of Tromsø, Norway.

Table of Contents

Acknowledgements **iii**

Table of Contents **v**

List of Figures **ix**

1 Introduction **1**

1.1 Peer-to-Peer Computing 2
1.2 Research Issues 3
1.2.1 Preventing Attacks 4
1.2.2 Tolerating Attacks 4
1.2.3 Thesis Statement 6
1.2.4 Scope and Limitations 7
1.3 Assumptions 8
1.4 Methodology 9
1.4.1 Disciplines in Practice 10
1.4.2 PlanetLab Experiments 10
1.4.3 Context of this Dissertation 11
1.5 Summary of Contributions 11
1.6 Outline of the Dissertation 12

2 Overlay Networks **13**

2.1 Current Systems and Protocols 13
2.1.1 Search Networks 13
2.1.2 Content-Addressable Networks 17
2.1.3 Content-Distribution Network 21
2.1.4 Storage Networks 23
2.2 General Model 25
2.2.1 Definition 25
2.2.2 Functional Components 27

3 Design Rationale **31**
3.1 Key Design Requirements 31
3.2 Identity Assignment 32
3.3 Topology Management 33
3.4 Messaging 35

4 The *Fireflies* Membership Management Protocol **39**
4.1 Protocol Overview 39
4.2 Certificate Authority 40
4.2.1 Certificates 40
4.2.2 Bounds on the Fraction of Byzantine Members 42
4.2.3 Revoking Certificates 42
4.3 Member Rings 43
4.3.1 Formal Definitions 44
4.3.2 The Probability of Having a Correct Monitor 45
4.3.3 Disabling Byzantine Monitors 46
4.4 Data Structures 47
4.5 Valid Accusations 49
4.6 Blocked Accusations 50
4.7 Failure Detection 51
4.7.1 Setting the Time-out Threshold τ 52
4.7.2 Rounding Error 53
4.7.3 Estimating Packet-Loss Rate 55
4.7.4 Threshold Limits 56
4.7.5 Pinging Attacks 56
4.8 Gossip 57
4.8.1 Ensuring Connectivity 57
4.8.2 Pseudo-Random Mesh 58
4.8.3 Time-out value Δ 58
4.8.4 Communication Efficiency 59
4.9 Protocol Steps 60

5 FiRE: The *Fireflies* Runtime Environment **63**
5.1 Overview 63
5.2 Global Data Structures 65
5.2.1 Configuration Options 65
5.2.2 Data Objects 66
5.2.3 Member Object 68
5.3 Main Functionality 69
5.3.1 Joining a Group 69
5.3.2 Events 70

5.3.3 Functions 72
5.4 Internal Issues 73
5.4.1 Membership Rings 73
5.4.2 Gossip 74
5.4.3 Adaptive Pinging Protocol 76

6 Evaluation **79**
6.1 Simulations 79
6.1.1 Overhead of Membership Maintenance 81
6.1.2 The Effect of Byzantine Members 83
6.2 PlanetLab 85
6.2.1 Experimental Setup 85
6.2.2 Measurement Study 86
6.2.3 Network Performance 92

7 Case Study: Disseminating Software Updates **95**
7.1 Background and Related Work 95
7.2 Architecture and Assumptions 97
7.3 Two-Phase Dissemination 98
7.4 Secure Dissemination Overlay 99
7.4.1 Mirror Mesh 100
7.4.2 Data Dissemination 100
7.4.3 Disconnected Nodes 102
7.5 Evaluation 102

8 Discussion **109**
8.1 Membership Management 109
8.1.1 No Membership 109
8.1.2 Partial Membership 110
8.1.3 View-Synchronous Membership 111
8.1.4 Weakly-Consistent Membership 115
8.2 Timing Attacks 116
8.2.1 Violation of Timing Bounds 117
8.2.2 Weaker Models of Synchrony 117
8.3 Applicability 118
8.3.1 One-Hop Distributed Hash Table 119
8.3.2 Multimedia Streaming 119

9 Conclusions **121**
9.1 Results 121
9.2 Future Work 124

A Publications **127**

References **129**

Abbreviations **147**

List of Figures

1.1 Member state diagram 5

2.1 Search in the Napster protocol 14
2.2 Search in the Gnutella protocol 15
2.3 Pastry routing example 19
2.4 CAN routing example 20
2.5 Overlay-network model 26
2.6 Functional components of overlay networks 27
2.7 Topology management functions 29

3.1 Undesirable overlay topologies 34
3.2 Locality in multi-hop overlay network routing 35
3.3 Required path diversity for multi-hop routing 37
3.4 Messaging overhead due to multi-hop routing 37

4.1 The *Fireflies* membership protocol 40
4.2 The role of the certificate authority 41
4.3 *Fireflies* mesh with three rings 43
4.4 *Fireflies* membership ring 44
4.5 Algorithm for computing the required number of rings 47
4.6 Required number of rings 48
4.7 Basic *Fireflies* member structure 49
4.8 Example of valid and invalid accusations 50
4.9 The likelihood of blocked accusations 51
4.10 Failure detection threshold τ as a function of packet loss . . . 53
4.11 The effect of rounding error on adaptive pinging 54
4.12 Adapting timeout threshold to packet loss rate 56
4.13 Number of rounds required to disseminate an update 60

5.1 Architectural overview of FIRE 64
5.2 Configuration options 66
5.3 FIRE data structures 67

5.4 FiRe member object 69
5.5 A FiRe service that registers to receive neighbor events 73
5.6 Pseudo-code for ring operations 74
5.7 Gossip of accusations, notes, and certificates 75
5.8 Adaptive Pinging Protocol 76

6.1 Simulated network overhead for varying packet-loss rates . . . 82
6.2 Simulated network overhead when under attack 84
6.3 Live members 87
6.4 Rate of timeouts 87
6.5 Aggregate rate of membership events on PlanetLab 89
6.6 Observed churn on PlanetLab 90
6.7 Network performance on PlanetLab 92

7.1 Cleartext dissemination 99
7.2 Two-Phase dissemination 99
7.3 Pseudo-code for the FirePatch dissemination protocol 101
7.4 Effect of the block size on dissemination 103
7.5 Time to complete phase-one 104
7.6 Time to complete phase-two 105
7.7 Reduction in the Window of Vulnerability due to two-phase dissemination 105
7.8 Comparison with naïve pull and push 106
7.9 Dissemination on PlanetLab 106

9.1 The *Fireflies* overlay-network stack 123

Chapter 1

Introduction

Internet services must accommodate an increasing load as their popularity and complexity grow. Although hardware utilization can be made more efficient with careful implementation, there are limitations. At some point scaling up means adding hardware components like CPU, disk, and memory. Cheap commodity class computers have proven themselves capable for this task and are widely available. For instance, in 2003 the popular Google web search service was estimated to run on a cluster of more than 15000 commodity class PCs [14]. The expense to acquire, setup, and maintain such a centralized system is, however, beyond the capability of most people and organizations.

An alternative approach to providing scalable Internet services emerged in 1999 with the Napster application [65]. Napster was a distributed file sharing application that specialized on music files. The application gave home users all over the world the ability to connect with one-another and share music files. Napster became hugely popular. During a four day period in May 2006, the service was visited by more than a half-million users [130]. Indeed, distributing digital content like music and video through file sharing networks has become so popular that international legislation and politics have been changed in order to protect the revenue of traditional content distribution businesses.

Although Napster used a centralized server to implement search, its novelty was that file transfers were done directly between clients machines. Compared to strict client-server architectures, Napster could therefore accommodate a larger number of users and a larger number of file transfers with less bandwidth and less disk capacity at the central location.

While Napster quickly ran into legal problems with the music distribution industry, and was soon shut down in a flurry of lawsuits, it became apparent that useful and scalable applications could be built by utilizing resources

available on the home-user's computers. This became known as *Peer-to-Peer (P2P) computing* [108].

1.1 Peer-to-Peer Computing

The fundamental ideas in P2P computing are *architectural symmetry* and peer *cooperation*. In order to provide a common service, each participating computer acts both as a client and as a server and is willing to share its resources with other members in order to provide a common service to all members. This idea was certainly not new at the time Napster was released. Indeed, the Internet itself and many of its long time services, like the Domain Name System (DNS) and the Usenet, are founded upon the same principle. Still, there is one important difference, that of deployment. A P2P system does not run on dedicated computers under the control of professional administrators. Instead, a P2P system runs on the edge of the Internet: on the personal computers, laptops and desktops, owned primarily by laymen. These computers represent a vast pool of resources. For instance, by May 2006, 84 million households were connected to the Internet [75] in the USA alone. Of these, 42% had high-speed broadband connections.

Although harvesting these hardware resources is useful, it is also challenging. For instance, P2P systems are generally highly *accessible* in that anyone with an Internet connection can participate. There are no predefined set of members. Instead, members join and leave the system continuously. Consequently, P2P systems must organize themselves *dynamically* as membership composition and load changes. The number of members can grow from a few to thousands. Thus, P2P systems must also be *scalable*. Participating machines are often located in different countries and on different continents. This leads to a high-level of *geographical dispersion*, and so P2P systems suffer from higher end-to-end latency and lower bandwidth capacity between system components compared to that of centralized solutions. This property is at odds with the property of architectural symmetry because symmetry disallows members to be specialized into functional classes in order to optimize system functionality. Symmetry is further complicated by the inherent *heterogeneity* of the computers of the home-users in that they have varying resources available to share, including the size of their disk, their processor speeds, and their network bandwidth. Although available resources vary, there are few, if any, members with sufficient resources to provide an acceptable level of service to all members. All functions in the critical path of the system is therefore *decentralized*.

Although P2P computing is challenging, many applications and systems

have been constructed. After the initial wave of file-sharing applications, generic middleware platforms for P2P computing emerged. These are known as P2P *overlay networks*, or overlays.[1]

Overlay networks have become a subject of interest to both system designers and researchers since they allow the Internet to be extended with a wide range of services by performing packet processing and routing in processes running on client machines. For instance, CoDoDNS [115], Overlook [144], and DDNS [39] are overlay networks that provide efficient and scalable name services; SplitStream [24], Bullet [90], and Chainsaw [109] are content distribution networks that achieve high throughput by spreading the data forwarding load amongst the clients; Azureus [27], BearShare [106], and KaZaa [74] are file-sharing networks with a large user base.

1.2 Research Issues

The Internet has attracted a non-negligible level of criminal activity [64]. Overlay networks are likely to be targeted due to their accessible, decentralized, and cooperating nature. Hence, to be *dependable and secure*, overlay networks must implement mechanisms that enable them to uphold their specified functions even when under attack.

In general, an overlay network can be subject to three kinds of attack [10]:

- *Denial-of-Service (DoS) attacks*, which aim at slowing down or halting a service.
- *Correctness attacks*, which aim at breaking the correctness of the delivered service such that it does not behave as intended.
- *Confidentiality attacks*, which aim at leaking information to parties that otherwise would not receive it.

Because attacks can be seen as *malicious and intentional failures*, the means to fight them fall within the following two categories [10]:

- *Fault prevention*, which are means to prevent faults from being introduced into the system.
- *Fault tolerance*, which are means to recover from errors such that they do not cause system failure.

[1]For brevity we will in the remainder of this dissertation leave out the adjective P2P where obvious and use the terms overlay networks and overlays interchangeably.

We will in the following describe both categories and identify fault tolerance by masking arbitrary, or *Byzantine failures* [93, 110], as the topic of this dissertation.

1.2.1 Preventing Attacks

Fault prevention techniques can mitigate a large number of attacks. For instance, good engineering practices and good programming skills can ensure that a system is resilient to corrupted messages. High-level programming languages can prevent common software flaws like unchecked array bounds.

Still, the accessible and open nature of overlay networks make them susceptible to Byzantine failures. This is because an attacker can gain control of the software and hardware stack of one or more overlay members simply by joining. Once in control, an attacker is free to generate arbitrary failures. He might for instance, omit storing data, forge messages, claim that other members have failed, or impersonate other members. In this case, we say that the attacker has *intruded* into the system. Other means for intruding include installing Trojan programs and exploiting software vulnerabilities in existing members [9].

Authentication and authorization schemes are inefficient in preventing overlay-network intrusions because participating machines are owned by humans without pre-established trust relationships. As such, members can not in general ascertain the intent and level of cooperation of other overlay members. In addition, an attacker might be able to circumvent established trust relationships by gaining control of network endpoints or by stealing secret cryptographic keys.

Mechanisms to externally verify the integrity of the software and hardware stack could prevent an intruder from inducing Byzantine failures. However, such mechanisms either assume the universal presence of a tamper resistant hardware device [60, 61] or assume that packet latencies are predictable [132]. Neither are realistic in the current wide-area Internet. Also, DoS attacks can not be prevented altogether in the current Internet infrastructure because overlay members can not prevent an attacker from sending large amounts of data.

1.2.2 Tolerating Attacks

Attacks that can not be prevented must be tolerated. Techniques within fault-tolerant computing fall within the following categories:

- *Error detection,* which are means to detect the presence of errors.

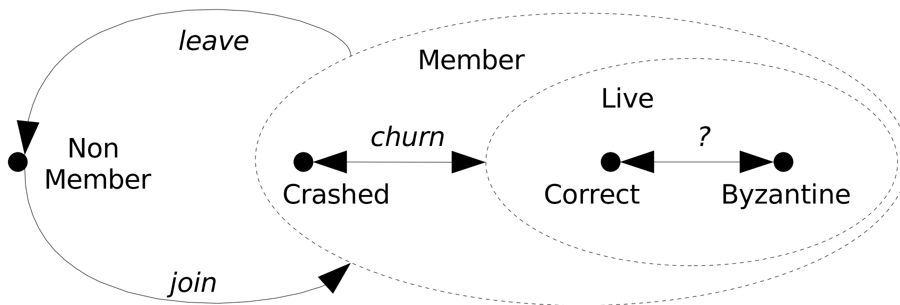


Figure 1.1: Member state diagram

- *Error handling,* which are means to eliminate errors from the system state.
- *Fault handling,* which are means to prevent faults from occurring more than once.

Since some attacks, like omission and forgery attacks, are hard to detect, they can not easily be removed by excluding the attacker from the overlay network. Also, the lack of a central component complicates error handling techniques like roll-back and roll-forward. Hence, to be fault tolerant, overlay networks must be able to mask errors such that they do not result in system failure. In particular, overlay networks must tolerate Byzantine failures due to the possible presence of an attacker.

We define an overlay network to be *intrusion tolerant* if it employs mechanisms that mask Byzantine failures. In such systems, processes may *join* the overlay becoming *members*, and existing members may permanently *leave*, as shown in Figure 1.1. Each member has a state that is either *correct*, *crashed*, or *Byzantine*. Correct members faithfully execute the specified overlay protocol, while crashed members do not execute any protocol steps. Byzantine members are not bound by the protocol and might execute arbitrary instructions. We refer to members that are either correct or Byzantine as *live*. Members might switch between live and crashed state, which is commonly referred to as *churn*. Also, correct members might be unreachable and appear crashed to other members due to transient network outages. Byzantine members can disguise themselves as correct members by executing the protocol, or as crashed members by not executing at all. Hence, correct members can not in general determine which members are Byzantine unless they reveal themselves as such by sending messages that prove that they are not following the protocol.

1.2.3 Thesis Statement

The high level of redundancy and the geographical dispersion, which are often present in overlay-networks, lend themselves naturally to fault masking. Many existing overlay structures leverage this to achieve a high tolerance to benign faults in a scalable manner [31, 116, 127, 140, 155]. These systems do not address intrusion-tolerance, which leads us to the overall problem that motivate this dissertation:

> Can P2P overlay networks be made intrusion tolerant while, at the same time, efficiently support thousands of members?

Several recent papers have addressed the problem of Byzantine failures within P2P overlay networks [22, 50, 117, 134, 135, 138]. A key observation is that an attacker can gain complete control of such networks if he is able to target the mechanism that maintains membership information. Such *membership attacks* include falsely claiming that a correct member has crashed, falsely claiming that a crashed member is live, and falsely modifying overlay link topology such that correct members prefer communicating with the attacker [22, 134].

An effective defense against membership attacks is to organize members in a *strict random structure*. This prevents an attacker from freely choosing his targets or focus his attacks on certain members [22, 134]. However, imposing such a rigid random structure prevents members from self organizing into topologies that are optimal according to runtime metrics like network proximity [23, 67]. As such, overlay routing structures that depend on structural flexibility to optimize end-to-end messaging delays are at odds with structural defenses.

By maintaining at each member a full view of all overlay members, messages can be sent directly to their destinations, avoiding the need for structural flexibility. Such *full membership protocols* have been shunned in the past as building blocks for overlay networks because of their inherent sensitivity to the increase in churn that follows from a growing number of members. A recent study has contradicted this argument and shown that, in many overlay networks, maintaining full membership is both possible and desirable [124]. Increased availability of broadband Internet access [75] might also reduce churn rate.

Full membership protocols that provide agreement on membership views have been extensively researched within the context of multicast oriented Group Communication Systems (GCSs) [33, 44]. Variants of such protocols that tolerate Byzantine failures have been constructed [26, 87, 100, 118]. Unfortunately, the overhead of consensus makes these protocols unscalable [69].

Performance evaluations of such systems typically operate with group sizes from four to a few dozen [26, 119, 156].

Full membership protocols that do not provide agreement on the membership views are more scalable and can be used in wide-area Internet environments [8, 42, 55, 68]. We conjecture that it is possible to extend these weakly-consistent full membership protocols with structural constraints to achieve a novel tradeoff between scalability and intrusion-tolerance that is well suited for overlay-network. The *thesis of this dissertation* is that:

> *Using epidemic techniques it is possible to build overlay networks and peer-to-peer systems that strike a useful balance between intrusion-tolerance and resource usage.*

To evaluate our thesis, this dissertation will devise a membership management protocol suitable for intrusion-tolerant overlay networks. The following properties will be used to measure the level of success:

- *Scalability.* We must show that our protocol can support overlay structures with thousands of members. We conjecture that there is significant overhead associated with fighting membership attacks, so we do not expect that our solution will be able to match the efficiency of protocols that assumes only benign failures. Still, our protocol must be sufficiently efficient such that it can be used within the constraints of current wide-area Internet network technologies.

- *Intrusion-tolerance.* We must show that our protocol can maintain membership information when under attack.

- *Applicability.* We must design and implement a proof-of-concept system based on our membership protocol and show that useful overlay services can be built on top of it.

1.2.4 Scope and Limitations

We limit the scope of this dissertation to the masking of operational (runtime) Byzantine faults and to protocol-level DoS attacks. In particular, this dissertation does not address the following:

- Attacks on system confidentiality.

- Development faults like software bugs and logic bombs.

- An extremely powerful attacker like government institutions (e.g., information warfare).

- Attacks that indirectly affect an overlay network.

As such, we do not consider attacks on systems and services co-located with the overlay network. This includes attacks that exhaust local bandwidth by targeting other service located on the same subnet as one or more overlay members. Also, we do not consider attacks on the software repository, human operators, or the social structures in which the overlay network resides.

1.3 Assumptions

An attacker might successfully attack a system if he is able to negate the assumptions of that system. We therefore make few assumptions on the capabilities of an attacker. We allow Byzantine members to collude and share state. They might also know the state of correct members. Byzantine members might be connected through high bandwidth low latency links and might be running on the same computer as other members. We do, however, make the following assumptions:

- We assume that Byzantine members do not have sufficient computational power to break cryptographic building blocks. In particular, we assume that they can not forge public key certificates, or public key signatures of correct or crashed members.

- We assume that there is a bounded uniform probability P_{byz} that a live member is Byzantine. This is a stronger condition than a bound on the probability that *any* member is Byzantine. Such a weaker condition would not suffice, as in the case that most non-Byzantine members are crashed, the few remaining correct members could be overwhelmed by Byzantine members. Nonetheless, the assumption that among all live members only a fraction is Byzantine is reasonable, particularly since we do not limit the fraction of crashed members among all members.

- We also assume that trivial DoS attacks like flooding can be detected and suppressed using techniques like port randomization, careful resource management, and rate limiting [11].

- We assume correct members have access to clocks running with a bounded difference to real time.

- We assume *synchronous communication* between correct members.

1.4 Methodology

The scientific method is a collection of techniques for gaining new knowledge about naturally occurring phenomenas. A commonly used technique is the *hypothetical-deductive method*, where the predictions of a hypothesis are checked against experimental observation. If experimental data correspond with predictions, the hypothesis is strengthened. If not, the hypothesis is falsified and must either be discarded or modified.

Although computer systems are human made and not naturally occurring, computer science meets every criterion for being a science [45]. The methods of computer science are commonly divided into the three following disciplines [46]:

- *Design,* rooted in engineering.
- *Theory,* rooted in mathematics.
- *Abstraction,* rooted in experimental methodology.

In the discipline of design, a system is systematically constructed to solve specific problems. First, requirements describing the functional and non-functional aspects of the system are stated. Next, the system is specified, designed, and implemented such that it fulfills the stated requirements. The construct is tested to check whether or not it meets the requirements.

In the discipline of theory, the objects of study are clearly defined such that hypotheses about how they relate to one another can be proved using logical reasoning. Studied objects can, for instance, be processes that exchange messages in an asynchronous communication network. How these processes relate to one another is specified with an *algorithm*. Using logical reasoning, hypotheses about such algorithms can be proven. Theoretic computer science often draws upon theorems and lemmas from the fields of pure mathematics and statistics, including graph and number theory.

In the discipline of abstraction, a model is deducted from hypotheses about observable objects or phenomenas. The predictions of the model are then compared to experimentally collected data. An incorrect prediction falsifies the initial hypothesis. If the predictions corresponds with observations, the hypothesis is strengthened. Observable objects of study include running systems like database systems and Internet applications. Abstraction has similarities to the scientific disciplines within natural sciences like biology, physics, and chemistry because their goal is to gain knowledge about the rules and laws that govern the behavior of observable objects.

1.4.1 Disciplines in Practice

As argued by Dennings et al. [46], the three disciplines of computer science are so intertwined that it is hard to separate one from the other. This dissertation therefore draws, to some extent, upon all three disciplines.

For instance, within the discipline of design, we have specified, designed, implemented, and tested a runnable system. The initial requirements for our system were synthesized by surveying existing overlay-network systems and from results within the research literature. Several iterations of the design process were conducted, leading to the system in its current incarnation, as described in this dissertation. Our implementation acts as a *proof-of-concept* that strengthens our thesis.

Within the discipline of theory, we have devised an overlay structure and process relations that provide properties that are beneficial to intrusion-tolerant overlays networks. These properties follows by logical reasoning from our assumptions. To support our claims, we use established mathematical theorems. In particular, we use sound statistical reasoning to deduct properties of the stochastic processes within our system.

Within the discipline of abstraction, we have observed the behavior of our system when running in a simulated environment that models aspects of expected deployment scenarios. Such simulations allow us to reason about or system when all factors are predictable and known. We have done simulations on the system as a whole, and with certain parts in isolation. The goals of our experiments were to verify that the system behave as expected.

1.4.2 PlanetLab Experiments

Ideally, the completed system would be observed within a real deployment scenario with real users and real load. Unfortunately, such an environment was not available to us. As an alternative, we ran our system in the PlanetLab test-bed [7, 111, 112], which allows us to observe the basic behavior of our system in a wide-area Internet setting.

In essence, PlanetLab is a world-wide collection of machines that are made available to scientists and organizations for the purpose of testing new scalable protocols and for deploying novel distributed services. By February 2006, PlanetLab contained over 600 machines at over 275 sites connected to the Internet in 30 countries. Because PlanetLab machines communicate through the Internet, network latency and bandwidth are affected by concurrent Internet traffic. Also, multiple services and experiments run concurrently on the individual PlanetLab machines, all affecting one another.

Because of this unknown concurrent load, PlanetLab experiments are not

considered reproducible [136]. Although this diminishes their scientific value, we consider our PlanetLab experiments important because they are *strong indicators* of how our system would behave in a real deployment scenarios.

1.4.3 Context of this Dissertation

This dissertation has been written as part of the Wide-Area Information Filtering (WAIF) project at the University of Tromsø, Norway. The overall goal of the WAIF project has been to construct an infrastructure for supporting the next generation Internet applications [78]. Its focus has been on issues like pervasive access to computing infrastructure [152], personalization, and high-level push-based communication.

From its infancy, the WAIF project conjectured that the P2P computing paradigm would play a key role [81]. Security issues was at first mostly ignored. It is from that context the topic of this dissertation emerged. Our methods and our subsequent results have been shaped and inspired by the WAIF project.

1.5 Summary of Contributions

This dissertation makes the following contributions:

- We have devised *Fireflies*: a novel membership management protocol that provides to each member an up-to-date view of all members. The views of correct members are made robust to membership attacks using a combination of epidemic dissemination, adaptive pinging, and a strict pseudo-random overlay-network structure. *Fireflies* ensures, with high probability, that all members are monitored by at least one correct member. At the same time, members can thwart high-level DoS attacks by disabling Byzantine monitors.

- We have designed and implemented FiRE: a framework that enables intrusion-tolerant overlay networks to be constructed. FiRE provides intrusion-tolerant membership management by implementing the *Fireflies* membership protocol. We have evaluated FiRE in a simulated environment and on PlanetLab with groups of up to 280 members and with as many as 20% executing membership attacks. Measured overhead on PlanetLab indicates that FiRE can support overlay network with thousands of members.

- We have designed, implemented, and evaluated FirePatch: a novel secure software-patch dissemination overlay network built using FiRe. FirePatch enables software vendors and end-users to fight hackers that reverse engineer software security patches into automated exploits.

Our approach provides a novel tradeoff between intrusion-tolerance and scalability that we find suitable for implementing overlay networks.

1.6 Outline of the Dissertation

In this chapter, we have motivated the research agenda for this dissertation. We have stated our thesis, discussed our methods, and summarized our contributions. The remainder of this dissertation is structured as follows:

- Chapter 2 defines the the concept of overlay networking as used in this dissertation. We do this by first describing several existing overlay networks, then we generalize these into a common overlay-network model.

- In Chapter 3 we state three design requirements for intrusion-tolerant overlay networks. We argue that solving all three is critical for the ability of an overlay network to tolerate intrusions. The design requirements therefore form the rationale for our later design.

- Chapter 4 presents *Fireflies*, our group membership protocol built to meet the design requirements in Chapter 3. *Fireflies* prevents an attacker from modifying membership information to his advantage.

- Chapter 5 describes the implementation of FiRe, our framework for constructing scalable and intrusion-tolerant overlay networks based on the *Fireflies* protocol.

- Chapter 6 presents evaluations of FiRe that we have conducted using a simulated network environment and PlanetLab.

- As a case study on how FiRe can be used to solve a real and important problems, Chapter 7 describes FirePatch, a software patch distribution overlay.

- In Chapter 8 we discuss our findings by describing alternative solutions to membership management, critiquing our assumptions on synchrony, and discussing the applicability of our solution.

- Chapter 9 concludes and outlines future work.

Chapter 2

Overlay Networks

Before we can devise a solution, we must have a clear understanding of relevant work in the domain. This chapter therefore defines the concept of overlay networks, as used in this dissertation, by describing several existing systems and protocols. Next, we outline a general four-layered model that captures common overlay-network functions. This model identifies membership management as a core function for overlay networks.

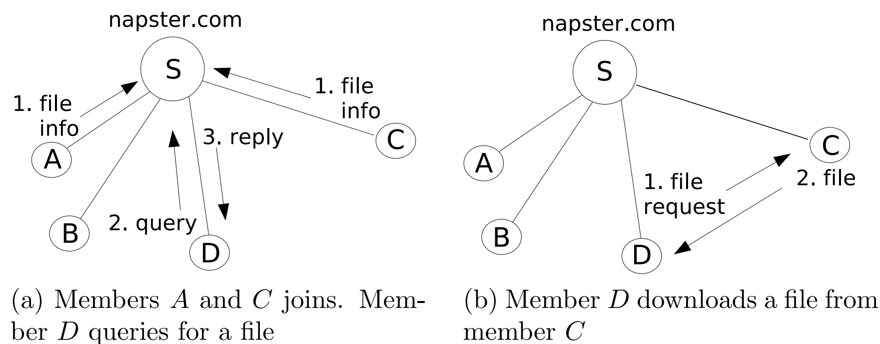
2.1 Current Systems and Protocols

During the last five years there has been a tremendous activity in academia, industry, and in the open-source community on the topic of P2P overlay networks. Four major usage areas have emerged: search networks, Content-Addressable Networks (CANs), Content-Distribution Networks (CDNs), and storage networks. The search networks and CDNs are often combined in file sharing applications.

In the following sections we will describe some existing systems within these four primary usage categories. Comprehensive surveys and taxonomies have been written by, for instance, Lua et al. [97], and Risson and Moors [121]. A good description of the early P2P systems can be found in Oram's book on harnessing the power of disruptive technologies [108].

2.1.1 Search Networks

A search network allows members to locate files or objects that are shared out by other members. Input to a search is a query that specifies required object properties. This can, for instance, be parts of a file name or, for a music file, the name of the artist. Search returns objects that match the



(a) Members A and C joins. Member D queries for a file

(b) Member D downloads a file from member C

Figure 2.1: Search in the Napster protocol

search criteria or a set of Uniform Resource Identifiers (URIs) that enable the objects to be retrieved.

A search network must be able to route query messages from the source to those members that have matching files. Then it must route replies back to the source. Optimizations include fusing, aggregating, and modifying replies before they are delivered to the end-user.

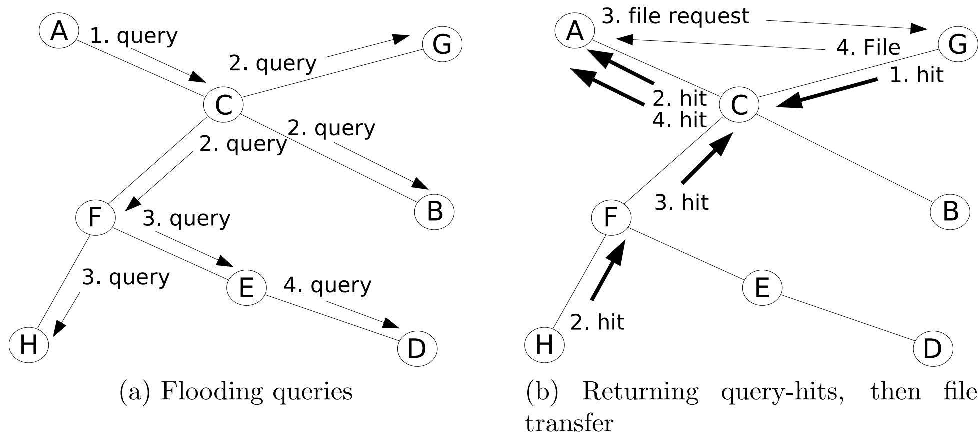
Napster

Napster [65, 108] was a music sharing application where a central server maintained an index of all shared files. Upon connection, a client uploaded its file meta-data to the central index. All subsequent queries were directed to the central server. The novel feature of Napster was that file transfers were done directly between the clients without involving the server. Due to its reliance on a centralized server for query matching, copyright owners were soon able to shut down the Napster service.

Figure 2.1 illustrates a Napster network with members A through D and the central server S. In Figure 2.1a A and C join the network, uploading file meta-data to S. Next, D queries the server and receives a reply that the newly joined member C has a matching file. In Figure 2.1b, D requests the file directly from C. C then replies with the file.

Gnutella

In contrast to Napster, the Gnutella protocol [31] facilitates search without using a central index. Instead, each Gnutella member knows only about its own files. To facilitate search, each query must be routed through the overlay to those members who have matching files. For this, each Gnutella member maintains a list of neighbors, which is a subset of all members. Query



(a) Flooding queries

(b) Returning query-hits, then file transfer

Figure 2.2: Search in the Gnutella protocol

messages are broadcast using "hot potato" forwarding [41], or flooding, where a member forwards all incoming messages to all its neighbors, except to the member which the message was received from. Figure 2.2 shows a Gnutella type of network with members A through H. In the figure, member A floods a query to all members.

Each query message contains a hop counter that limits its range. To decrease load, the maximum number of hops is typically set lower than the expected diameter of the Internet. To increase search anonymity, queries do not contain the identity or network address of the source (i.e., the member that submitted the query.) Instead, query-hit messages are propagated back to the source along the query's forwarding path. In Figure 2.2b, members G and H have matching files for A's query and return query-hit messages along the forwarding path.

To enable back propagation, queries include a pseudo-unique message identifier, which is cached at each visited member. Subsequent file transfers are done point-to-point using the Hypertext Transfer Protocol (HTTP) as illustrated in Figure 2.2b. The URI required for making such requests is included in the returned query-hit messages. Anonymity is not maintained during file transfers.

In general, flooding protocols are highly resilient to omission attacks when the underlying mesh has sufficient link redundancy. If an attacker omits forwarding a query, its neighbors will likely receive it from other members. However, the use of query flooding in Gnutella has been shown to have a negative impact on its ability to scale. By theoretically analyzing properties of the real world Gnutella network, Ritter [122] shows that a single query

can generate as much as 800 Megabyte (MB) of aggregate network traffic.

FastTrack

The FastTrack protocol [72] improves the scalability of Gnutella by enabling members to participate as either leaf nodes or super peers[1]. The super peers form a Gnutella like flooding network that maintains the meta-data of the leaf nodes. Leaf nodes connect to one or more super peers but do not participate in the flooding of ping and query messages. Typically, well-connected high-bandwidth members become super peers.

GUESS

One problem with query flooding protocols is that submitted queries can not be stopped. When a match is found at one member, subsequent redundant forwarding of the query can not be avoided.

The Guess protocol [43] approaches this by making search iterative instead of recursive, as with flooding. To find an object, a member must submit its query to each of its neighbors in turn and await a reply. The implication is that the individual response times all add up. As such, an attacker can easily stall a search by delaying its response. However, the protocol does allow for members to trade increased probability of redundant forwarding for increased response time by submitting a query in parallel to multiple neighbors. To achieve high recall, iterative search protocols requires each member to maintain a large list of neighbors. In particular, to reach all members, complete membership information must be maintained.

PALocate

The PALocate protocol [81] reduces the number of messages on the wire compared to Gnutella by having members store received queries as hints. The idea is that if some member m submits a query, it is likely to find and download matching objects. Member m is therefore a likely candidate to match similar queries. Such queries should therefore be forwarded to m in order to improve forwarding accuracy.

In PALocate, similarity is based on keyword matching. However, any similarity metric can be used in practice. In addition to storing past queries, members also populate their hint caches by actively exchanging hints through gossip.

[1]Super peers are in other systems known as ultra peers or hubs.

To forward a query, a member m first checks its hint cache. If no hint is found, it reverts to Gnutella like flooding and forwards the query to all its neighbors. If m has one or more hints, the query is forwarded to the hinted members. An upper bound is set on the fan-out to limit the forwarding load. Although the protocol is shown to be more efficient than Gnutella, it does not address how cache entries are to be kept up-to-date in face of churn.

2.1.2 Content-Addressable Networks

Content-Addressable Networks (CANs) are somewhat similar to search networks in that their primary goal is to locate objects. However, unlike search networks, CAN localization is based on unique object identifiers, or keys, and not queries. A key maps to at most one object, although each object can be redundantly maintained at multiple members.

Distributed Hash Tables

After the initial flurry of activity around Gnutella and Napster, four overlay substrates emerged in 2001. They all provided CAN functionality and addressed scalability issues in Gnutella by imposing a strict structure on the overlay topology. These four overlays were: Pastry [127], Tapestry [155], Chord [139], and the CAN system [116].[2] In essence, they all provide the same abstraction: a Distributed Hash Table (DHT), which implements the following functions:

- *put(key, object),* stores a persistent binding between a key and an object.
- *get(key),* returns the previously stored object bound to a key.
- *remove(key),* removes any previously object bound to a key.

Each DHT member m is assigned an unique random identity, $m.nodeId$, that is drawn from the same id space as the keys. Each objects o is assigned a key, $o.key$, based on, for instance, the hash of its content or a public key. A proximity function is also defined. The *root* of an object o is the member m who, according to the proximity function, is closest to o in the id space. Member m is then required to store o and produce copies of it on request. Since the existence and integrity of o depend on the correct behavior of its root, o is typically replicated by assigning it to k roots. This can be

[2] The CAN system should not be confused with the more general term of a Content-Addressable Network, although the CAN system implements such a structure.

done either by inserting o under k different keys, or by having the k closest members maintain o.

The id space is typically large. For instance, the Pastry id space is set to 2^{128}. As the number of participants is expected to be many orders of magnitude less, each member will be responsible for maintaining multiple keys. The use of consistent hashing to map nodeIds and keys into the same id space ensures that existing objects are spread uniformly among the members. Also, as members join and leave the overlay, root assignment for o might shift from one member to some other member. This requires o to be transferred to those members.

Knowing $o.key$, a member m can access o by routing a message through the overlay to one of o's roots. Each such message, d, contains the key of the object to which d is addressed (i.e., $d.key = o.key$). Varying routing schemes are used in different DHT implementation. We will now briefly describe some of them.

Pastry and Tapestry Routing

Pastry and Tapestry use variants of the greedy prefix routing algorithm suggested by Plaxton et al. [114]. In their scheme, each message d is routed through a sequence of h members $m_1 \rightarrow m_2 \rightarrow \ldots \rightarrow m_h$, ending at o's root. For routing step i, one of the two following invariants is maintained

1. $m_i.nodeId$ and $d.key$ share a common identity prefix that is longer than $m_{i-1}.nodeId$ and $d.key$.

2. $m_i.nodeId$ and $d.key$ is numerically closer than $m_{i-1}.nodeId$ and $d.key$

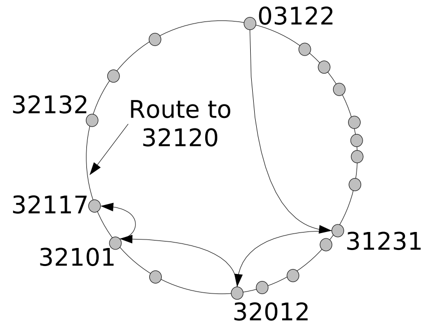
Although Tapestry uses prefixes instead of suffixes, the general principle is the same. The following description will focus on Pastry.

To maintain the routing invariants, each Pastry member maintains a routing table. A routing table contains $log_B N$ rows and $(B-1)$ columns, where B is the chosen numerical base and N is the number of members. Row r of the routing table of member m contains entries for members that share a nodeId prefix of length $r-1$ in common with m. Column c contains entries whose most significant digit after the prefix is $c-1$. For instance, consider the routing table in Figure 2.3a, for an imaginary Pastry member m with $m.nodeId = 32012$. Row 3 column 4 of that table contains the entry for some other member whose nodeId starts with 323. This member has a prefix of length $3-1=2$ in common with m. The most significant digit of those nodeIds, excluding the prefix, is 3. In each row r exactly one entry will have a common prefix of length r. In this case, the next row of the routing

nodeId: 32012			
$0****$	$1****$	$2****$	$3\Downarrow$
$30***$	$31***$	$32\Downarrow$	$33***$
$320\Downarrow$	**32101**	$322**$	$323**$
$3200*$	$3201\Downarrow$	$3202*$	$3203*$
32010	32011	32012	32013

$*$ = any digit $\Downarrow$ = next row

(a) Routing table for member 32012



(b) Routing ring

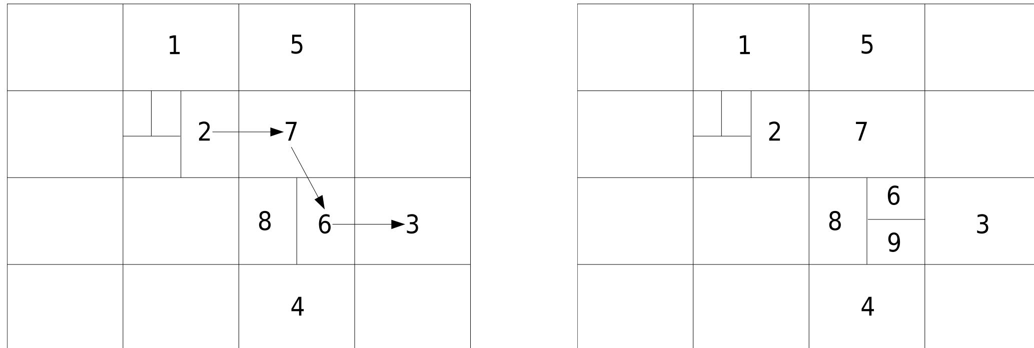
Figure 2.3: Pastry routing example

table should be used and is indicated with a $\Downarrow$ in the figure. As shown in Figure 2.3b, if m is to route a message with key 32120, the node in row 3, column 2 is selected, and the message is forwarded to member 32101.

In addition to the routing table, each Pastry member maintains a leaf set and a neighbors list. The leaf set of member m contains entries for those members whose nodeIds are numerically closest to m. Of these, half have larger and half have smaller nodeIds. The length of the leaf table is fixed. When routing a message d, a member first checks the leaf set to see if $d.key$ is within range. If not, d is forwarded using the routing table. The leaf set is also used to find alternative routing paths such that failed members can be routed around. The neighbors list of m contains entries for members that are close to m with respect to network locality. This increases the routing performance of Pastry since messages use fewer hops in the underlying network topology.

Joining a Pastry network requires several steps. First, the joining member m_1 routes a join message d to the root of its own nodeId (i.e., $d.key = m_1.nodeId$). Let m_2 be that member. All members on the route from m_1 to m_2 respond to m_1 by sending it their routing tables. From these, m_1 computes its own tables. Next, m_1 notifies its arrival by sending its state to all members in its tables. Those members then update their tables.

Pastry takes an optimistic approach to solving contention due to concurrent node arrivals and departures. Each member has a time stamp associated with each table. Whenever tables are exchanged, this time stamp is checked and updated. If inconsistency is detected during a join, the operation is restarted. Stale entries in the tables due to departures are repaired lazily by exchanging state with members numerically close to the departed member.



(a) 2 sends a message that falls within 3's zone

(b) 9 joins, causing 6's zone to be split

Figure 2.4: CAN routing example

Routing in the CAN system

The CAN system uses a different type of routing than Pastry and Tapestry. It assigns each member to a zone in the Cartesian coordinate space defined by a d-dimensional torus. The two-dimensional variant of such a construct can be drawn as a square where the top and bottom edges and the left and right edges connect or "wrap around", as illustrated in Figure 2.4. Zones are assigned such that they do not overlap and such that they cover the entire coordinate space. Their individual size might vary. Messages are routed between adjacent zones until they reach their destination, as shown in Figure 2.4a.

When a member m_1 wants to join the overlay, it contacts the current occupant of its designated zone, say m_2, by routing a join message to its own nodeId. m_1 announces its presence to m_2 and to all adjacent zones. The zone is then split between m_1 and m_2. For instance, let 9 be a member that joins the overlay shown in Figure 2.4a. Its designated zone is currently occupied by member 6. The zone is then split between 6 and 9 as shown in Figure 2.4b. In the CAN system, routing tables do not grow with the number of members.

Chord Routing

Chord [139] organizes its members and keys in a circular address space like Pastry. Unlike Pastry, Chord messages are only routed in one direction in the ring using the successor relationship. The successor of some key k is the member who has the smallest nodeId equal to or larger than k, or if no such

member exists, the member with the lowest nodeId. To limit the number of routing hops, each member maintains a finger table. Member m's i'th finger table entry contains the successor of distance at least 2^{i-1} from m.

2.1.3 Content-Distribution Network

CDNs provide application level multicast functionality targeted at delivering large sized messages from a single source to a large number of receivers. The messages can be large music files, software packages, or live TV broadcasts. The general idea is that the upstream capacity of the source is increased by having members participate in the dissemination process. Many file sharing applications add CDN functionality to their search infrastructure to increase members' download speeds.

Multimedia streaming is a particular type of content distribution where the data consist of a continuous sequence of segments. After consuming a segment, a member must receive the next within a bounded time. If not, that segment will not be useful as the movie or music stream has played beyond that time segment. Members can usually cope with a certain rate of segment loss.

Bullet

Bullet [90] is a CDN protocol that can be layered on top of an existing overlay tree structure in order to provide efficient data dissemination. To disseminate a file, or a multimedia segment, the source data is split into a set of fixed-sized blocks. Each member, including the source, runs a local scheduling algorithm that decides which of it children each block should be forwarded to. By sending disjoint sets of blocks to each child, a member ensures that the different parts of the file are spread to different parts of the overlay. The overlap between block streams is tuned in accordance to bandwidth capacity. To receive all blocks, members exchange blocks parallel to the overlay tree structure. For this, each member maintains a Bloom filter [18] that summarizes received blocks. Members broadcast their Bloom filters to a random subset of the members using the RanSub protocol [89]. Members with less similar Bloom filters become neighbors and exchange blocks. To prevent duplication, each neighbor is assigned responsibility for different parts of the stream.

SplitStream

SplitStream [24] divides each data segment into a fixed set of overlapping stripes using erasure coding so that the original data segment can be reassembled from a subset of the stripes. Each member m has an inbound and an outbound capacity limit. The inbound limit of m specifies how many stripes m can receive concurrently. m's outbound limit specifies the number of stripes m can transmit concurrently to other members. The problem is then to create a graph that ensures that each member receives a sufficient number of distinct stripes while, at the same time, not violating their capacity limits.

SplitStream solves this problem by constructing one dissemination tree for each stripe. The number of trees that member m participates in depends upon its capacity limit. As such, different members might participate in a different number of trees. Using the Pastry prefix scheme, m is made an internal node in exactly one tree while a leaf node in all other trees. m will be responsible for forwarding one stripe to one or more members. If m's outbound capacity is violated, it picks one of its children and redirect that member down the broadcast tree using a push-down protocol. SplitStream also employs a special spare capacity group in which members with spare outbound capacity register. If a member can not find a parent node to attach to through the push-down protocol, it will contact the spare capacity group. The feasibility of the SplitStream approach is based on the observation that most vertices in a tree are leaf nodes. A larger fan-out increases the fraction of leaf nodes. For instance, with a fan-out of 16, over 90% of the vertices are leaf-vertices.

CoolStreaming/DONet

CoolStreaming (DONet) [154] is a data driven multimedia streaming overlay, which has been deployed in practice to deliver TV-quality video (i.e., throughput above 450 Kilobits per second) to more than 4000 simultaneous members.

Each DONet member m maintains a membership cache (mCache) containing those members known to m. To keep the mCaches up to date, all members periodically broadcast a heartbeat message that announces their continued presence in the overlay. If a member fails to send a heartbeat message within a certain period of time, it will time-out and be removed from the mCaches of the other members. Heartbeat messages are disseminated using gossip.

Each member selects a set of neighbors from its mCache and exchanges

multimedia segments with those. CoolStreaming improves the downstream bandwidth using an adaptive neighbor selection algorithm that prioritizes partners based on their mutual ability to deliver content to each other. For this, each member maintains the number of segments received per time unit from each of its neighbors. In order to find better neighbors, each member will periodically include a random member from its mCache as a neighbor, and exclude the one with the poorest performance. Neighbors continuously exchange bitmaps of available segments with one another. A member request segments from its neighbors by sending them a similar bitmap. A local scheduling algorithm uses heuristic in order to best ensure that a segment is delivered before the deadline.

2.1.4 Storage Networks

Storage networks provide files-system operations to their members. DHT-based overlay networks are convenient substrates for global-scale persistent file systems because they provide the infrastructure to maintain the binding between a file name and file data. Also, most DHTs maintain replicas of each stored object to increase availability. By building upon a DHT substrate, a large number of members can be accommodated.

PAST

The Past file system [128] builds upon the Pastry DHT substrate. In Past a 160 bit file-id is generated by hashing the combination of the file name, the owner's public key, and a random salt. The 128 most significant bits of the file-ids are used as the Pastry routing key. Each key uniquely identifies one particular file and its owner. The number of replicas is specified per files upon insertion. Files are immutable once inserted into Past. They can, however, be removed by their owner, although Past does not guarantee that such operations succeed. Some or all replicas might outlive their removal command.

Because file sizes vary and files are assigned randomly, storage requirements might differ from member to member. If a member does not have the needed disk space to store a file, Past divert that file to some other member. A pointer to the new location is maintained at the first location such that the file can be found using Pastry prefix routing. Each file diversion adds a routing hop when locating that file and complicates the task of maintaining the required number of file replicas when members join or fail.

The Cooperative File System

The Cooperative File system (CFS) [40] builds upon the Chord DHT substrate. In CFS, each file is split into a set of fixed-sized blocks. With each data block is associated a routing key, which is calculated by hashing the content of the block. Upon insertion, this key is used to route the block to the set of responsible members. The use of fixed-sized blocks results in better load balancing than when storing whole files, as in PAST. It also enables a file to be downloaded in parallel from multiple sources. Unfortunately, latency suffers as each read operation requires multiple DHT lookups. To improve performance, CFS uses aggressive caching and file read-ahead.

In addition to the data blocks described above, CFS defines root blocks, directory blocks, and inode blocks. These block types contain pointers to other blocks structures so that hierarchical file-system structures can be implemented. Root blocks form the entry point to such file structures and can be updated in-place. The key for a root block is the hash of the owner's public key. Each root block is signed with the private key of the owner. Time stamps are used to prevents replay attacks.

CFS stores blocks only for a finite period of time. Members must refresh their blocks periodically in order to keep them in the system. When a block lease expires, a server is free to delete it. To prevent exhaustion attacks CFS enforces a weak form of quotas. Each server allows a single entity to occupy only 0.1% of its storage. Since blocks are assigned uniformly and randomly to members, each member should be able to use on average 0.1% of the available storage capacity.

OceanStore

The OceanStore [91] project, which is partially implemented in the Pond prototype [120], builds on the Tapestry substrate and allows flexible object update semantic. Each OceanStore object is represented as a sequence of successive versions. Objects are not locked for writing. As such, multiple clients can concurrently read and write to the same object. To ensure that all replicas eventually observe the same sequence of updates, a small subset of the replicas are selected to act as an object's primaries. Members submit all updates to these members. The primaries agree upon the update sequence using a Byzantine fault tolerant consensus protocol. The resulting schedule is broadcasted to all secondary replicas. Primaries are selected dynamically based on their proximity to objects in the Tapestry identity space. Primaries are assumed to be stable since view changes block the consensus protocol.

Freenet

Freenet [36] addresses privacy issues within file storage networks using a combination of random walks, to locate files, and layered asymmetric encryption. Although not considered a DHT, Freenet builds upon a similar principles of mapping hashed file content keys to member identities.

In Freenet, object bindings are not mapped to members deterministically as in DHTs. Instead, members specialize in delivering certain content. Members forward messages to those neighbors that they consider more specialized in finding the requested content than themselves. Hop counters and message identifiers are used to prevent infinite routing loops, as in Gnutella. Values are sent back through the forwarding path. Unlike DHTs, Freenet does not guarantee that lookups for existing objects succeed.

2.2 General Model

An overlay network is a virtual packet processing and routing network built on top of an existing network infrastructure like the Internet. As with any network, an overlay is commonly represented as a graph where the vertices are member processes and the edges are communication links. An overlay network is constructed from a subset of the members in the underlying network. Its links are logical in that they can be made up of multiple links in the underlying network, as shown in Figure 2.5. Overlay links exist only as part of the overlay state.

Although overlay networks can be constructed on top of any communication network, and can even be stacked on top of each other, this dissertation is only concerned with overlays within the P2P computing paradigm. Hence, our member processes are running on the desktop and laptop machines of end-users that are situated on the edges of the Internet. Some overlay networks, like the CoDNS Internet name lookup service [115], enforce an open group policy [32] where contribution is not required in order to utilize the service. We will, however, in this section define an overlay network to include only those members that actively participate in providing the overlay service.

2.2.1 Definition

Given the above, we define an overlay network as a graph $O = \{M, E\}$, where $M = \{m_1, m_2, \ldots, m_n\}$ is an unordered set of n member processes, and E is the set of virtual communication links connecting those members. A member m_i can only send messages to some other member, m_j, if the link $(m_i, m_j) \in E$. Links can be either symmetrical or asymmetrical. For a

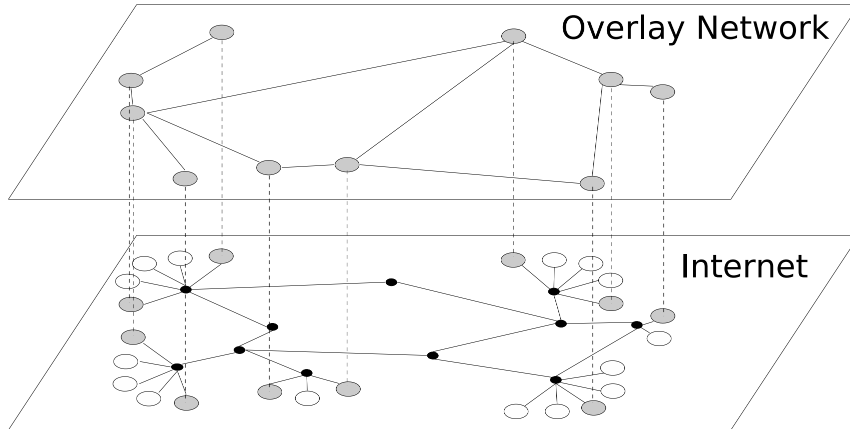


Figure 2.5: Overlay-network model

symmetrical link, $(m_i, m_j) \in E \Leftrightarrow (m_j, m_i) \in E$. In this case, we say that m_i and m_j are *neighbors*. For asymmetric links, $(m_i, m_j) \in E$, does not imply that $(m_j, m_i) \in E$. In this case, we say that m_i is neighbor to m_j.

Whether an overlay network has symmetric or asymmetric links depends on the flow of high-level messages within the overlay structure, and not on the flow of low-level transport messages. For instance, a single Transmission Control Protocol (TCP) connection can be used to represent a symmetric overlay link although symmetry of low level signalling needs to be broken in order to establish that connection. Similarly, acknowledgements and other control messages are allowed to flow back across an asymmetric link. As such, Chord links are asymmetrical since messages travels in only one direction along the ring. Gnutella links are symmetrical because query broadcasts are not directional.

The overlay assigns to each member m an unique identity $m.id$. In addition, m has a network address $m.address$ on which it can receive messages. Each member m has a view $m.view$, which is a subset of all participating members, M. Given two members m_1 and m_2, $m_2 \in m_1.view$ means that m_1 considers that m_2 is at least until recently live. The opposite, $m_2 \notin m_1.view$, means the m_1 considers that m_2 is at least until recently crashed. Each member m maintains a neighborhood $m.neighbors$, which is a subset of $m.view$. Although members can communicate directly with one another in the underlying network, correct behavior within an overlay protocol dictates that m_1 can only send a message to m_2 if m_1 is a neighbor to m_2.

Services
Messaging
Topology Management
Identity Assignment

Figure 2.6: Functional components of overlay networks

2.2.2 Functional Components

Based on the description of current systems in Section 2.1 and the definitions in Section 2.2.1, we have identified a model that captures an overlay network's functional components and their relation to one another.

Our model is divided into four layers, each on top of the next. As shown in Figure 2.6, each layer depends upon the functionality of all layers below it. Hence, the topmost layer is the service layer which interacts with the user. This layer defines service specific functionality like user interfaces and specific data decoding. The next layer is messaging, which service logic uses to communicate with other members. Messaging is based on the topology management layer, which maintains the set of virtual links. The topology is dependent upon the assignment of identities in order for messages to be routed to specific members. Our model is illustrated in Figure 2.6. We will in the following describe the layers in more detail.

Services

Overlay networks provision for high-level services that typically mask the distributed nature of the underlying network substrate. The most common services fall withing the classification in Section 2.1, and include

- file search,
- persistent file storage and retrieval, and
- multimedia streaming.

Messaging

In its simplicity, the Internet, with the Internet Protocol (IP), only provides one service: best effort unicast message delivery. Layered on top of IP, three additional protocols are considered part of the core Internet protocol stack. These are the Internet Control Message Protocol (ICMP), the User Datagram

Protocol (UDP), and the Transmission Control Protocol (TCP). Overlay networks extend these protocols and provide additional primitives like:

- Limited range broadcast, as in Gnutella.
- Location independent addressing, as in DHTs.
- Application level multicast, as in Bullet.
- Multimedia streaming, as in CoolStream.

As members only send messages to their neighbors, the overlay-network topology dictates how different forms of messaging can be implemented. For instance, the random structure of Gnutella lends itself well to flooding, but makes point-to-point messaging expensive. The Pastry overlay structure is well suited for prefix routing but makes range queries difficult to implement.

Topology Management

The members of an overlay network span out a communication mesh of links through which messages can be sent. Each link is represented by an entry in a members neighbor list. For instance, in Pastry, a member's view is the union of the entries in its routing table, leaf set, and neighbor list. Overlay links are logical in that the existence of such a link between two members does not imply that there needs to exist an established connection between those members in the underlying network (e.g., TCP connections), nor needs there ever be messages exchanged between those members. For instance, a Pastry routing entry might never be used to route a message. The existence of a link from member m_1 to m_2 is based upon the following three elements:

1. *Knowledge.* That m_1 knows about m_2 and consider it live (i.e. $m_2 \in m_1.view$). In particular, m_1 must know $m_2.address$.
2. *Selection.* A local algorithm at m_1 selects m_2 as a neighbor (i.e., $m_2 \in m_1.neighbors$).
3. *Authorization.* m_2 is willing to accept messages from m_1.

From m_2's perspective this sequence can be reversed. Authorization for m_1 to connect to m_2 is given only when m_2 has selected m_1 as its neighbor. To select m_1 as its neighbor, m_2 must have knowledge about m_1.

The dynamic property of overlay networks, which is due to members continuously joining and leaving, requires the link structure to continuously be updated and reorganized. This means that each member actively participate in the following three *membership management mechanisms:*

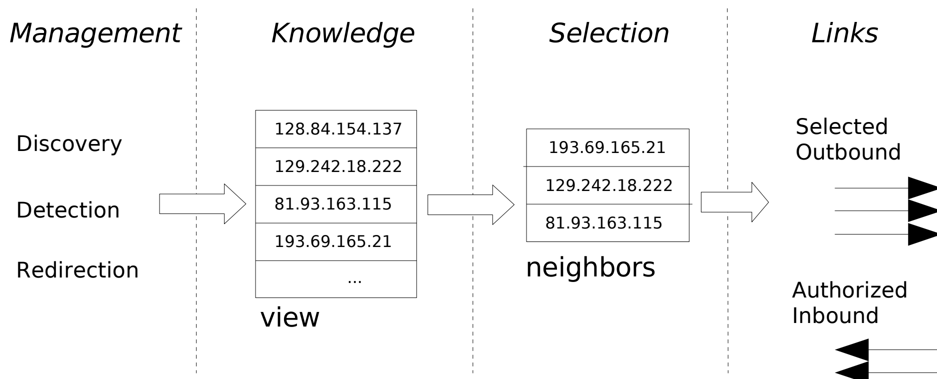


Figure 2.7: Topology management functions

- the *discovery* of other members,
- *detection* and exclusion of failed or non responding members, and
- the *redirection* of neighbors to better partners.

For instance, in Gnutella, a member m_1 actively discovers other members by flooding ping messages. Upon receiving the ping message, member m_2 returns its network address in a pong message. Upon receiving the pong message, m_1 enter m_2 in its view. If another pong message is not received within some bounded time, m_1 will remove m_2 from its view. Gnutella members are free to choose their neighbors. Consequently, the Gnutella mesh does not follow a particular structure, although it has been shown to exhibit a small-world topology in practice [130]. Pastry members discover one another by exchanging routing tables when new members join. Probing and timeouts are used to detect failed members.

The membership management function establishes and maintains for each member m the knowledge that m has of other members in the overlay. In particular, $m.view$ contains members that m considers at least until recently as live. From $m.view$, m selects a set of neighbors $m.neighbors$ using some service specific criteria. $m.neighbors$ dictates which members m is to establish connections to, and which member are to connect to m. These functions and how they relate to one another is illustrated in Figure 2.7.

Identity Assignment

Explicit identities are required in order to solve fundamental distributed problems like routing and election. As such, an overlay network assigns to each member an unique identity. IP addresses can not be used in general

because some overlays require a naming scheme with other properties. This includes properties like:

- randomness,
- uniformity,
- persistence, and
- size of the address space.

For instance, to balance load, DHT systems require members to be dispersed uniformly and randomly in the identity space. Due to their hierarchical nature, IP addresses do not provide this property. The CAN system, for instance, solves this by having each member pick a random identity from a large identity space.

Persistence is another common problem with IP addresses since some members have their network address dynamically allocated. Some member might also be running behind Network Address Translation (NAT) servers that assign non-routable private class IP addresses. If state is to be kept on such members across TCP connections, an overlay network must implement some other identity scheme. The problem of traversing NAT boxes in order to make masked members reachable, is addressed in [66].

Chapter 3

Design Rationale

The previous chapter identified membership management as a key functional component in overlay networks. This chapter describes three important design requirements for an intrusion-tolerant overlay network that relate to membership management. These three requirements form the rationale for our design and implementation.

3.1 Key Design Requirements

We have identified the following three design requirements for intrusion-tolerant overlay networks.

Requirement 1 *The fraction of Byzantine members within the set of all members must be upper bounded.*

Requirement 2 *The integrity of the overlay link structure must be maintained through intrusion-tolerant techniques for discovery, failure detection, and redirection of members.*

Requirement 3 *Multi-hop overlay-network routing should be avoided in the critical path.*

Each of these design requirements relate to a functional components, identified in Section 2.2.2, as follows:

Requirement 3 $\longrightarrow$	Messaging
Requirement 2 $\longrightarrow$	Topology Management
Requirement 1 $\longrightarrow$	Identity Assignment

As such, Requirement 1 and Requirement 2 relate to intrusion-tolerant membership management. Requirement 3 relate to messaging. We will in the following discuss each requirement.

3.2 Identity Assignment

To participate in an overlay network, a member must have a valid identity. If, for instance, IP addresses are used as overlay identities, each member must possess one. The assignment of identities is a form of access authorization mechanism and plays a key role in the ability of an overlay network to fight attacks.

In distributed systems, the masking of both Byzantine and benign faults is achieved through redundancy. If an attacker is able to control a large fraction of the members, redundancy will be undermined. In this case, the correct behavior of the system would be at the mercy of the attacker.

The fraction of Byzantine members that can be tolerated depends on the protocol used and its underlying assumptions. However, there exist some general upper bounds. For instance, for consensus the fraction of Byzantine members among live members must be less than one-third [93]. Such limits must hold within the overlay network as a whole and within any subsets of the members selected for particular tasks. For instance, in OceanStore, the number of Byzantine members within each primary replication group must be less than one-third.

If an attacker is able to acquire a large number of identities, an overlay is at risk of being compromised. A system is in particular at risk if members may choose their identities freely. The forging of multiple identities in order to gain control of a system is often referred to as to as a *Sybil attack* [50]. Hence, to be intrusion tolerant, an overlay network must implement some mechanism that limits the fraction of members that are Byzantine.

Bazzi and Konjevod [15] suggest that triangulation of packet latencies can certify that two identities are not held by a single host. Their approach is based on the observation that physical dispersion of Internet hosts implies an unforgeable lower bound on packet latencies due to the speed of light limit. Consequently, an attacker that runs multiple protocol instances from a single subnet can be detected and removed. The practicality of their approach remains unproven. It is doubtful that this approach can be applied in the wide-area Internet due to jitter in packet latencies and packet loss.

Castro and Liskov [25] argue that Byzantine members can be recovered to a benign state by having all members periodically reboot. Such proactive recovery would be useful to ensure that the fraction of Byzantine members does not grow above assumed bounds during the lifetime of long running services. However, their scheme assumes that the attacker does not have access to the hardware. Also, it is not clear how reboot times can be organized to avoid a large number of correct members rebooting at the same time.

Although the above mechanisms will make it harder for an attacker

to accumulate and control a large number of overlay-network identities, Douceur [50] argues that only a central authority can establish distinctiveness. The mechanisms by which this is done includes tying overlay identities to real-world identities like social security numbers or driver-licence identities. Unfortunately, such schemes do not provide any guarantees since real-world identities can be forged as well. In the end, a resourceful attacker can simply circumvent these mechanisms by recruiting a large number of people through mundane means such as bribes, social engineering, or physical threats.

3.3 Topology Management

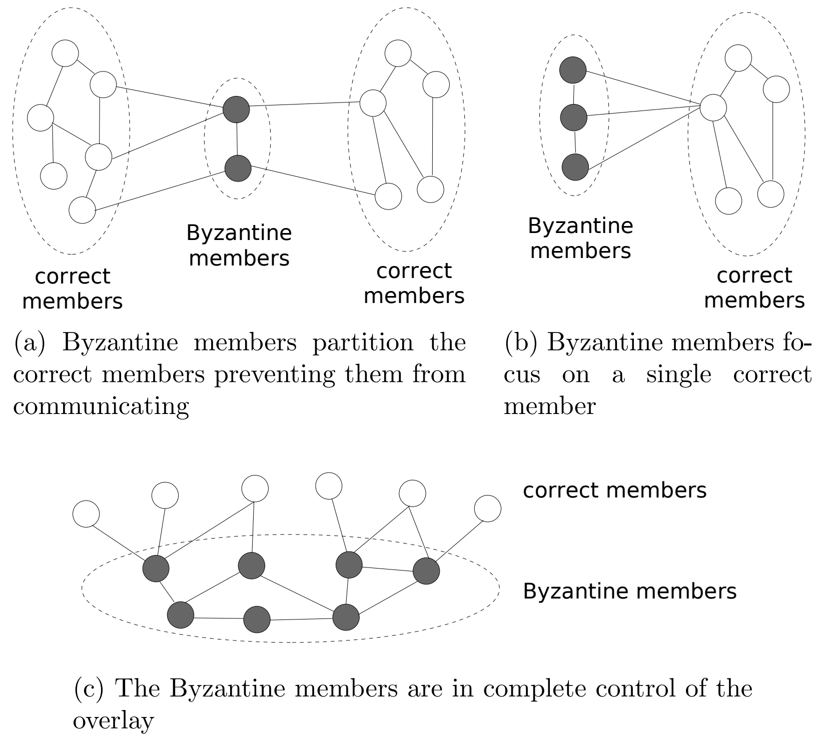
Limiting the fraction of Byzantine members among live members is not sufficient to thwart all types of attack. The overlay structure must also be resilient to certain modifications to its topology. An attacker might, for instance, try to modify the overlay link structure into undesirable patterns by affecting correct members' neighbor tables. Undesirable topological structures include the following:

- *Partitioning.* Colluding Byzantine members divide the correct members into two or more partitions. All communication between the partitions of correct members must go through the set of Byzantine members, as shown in Figure 3.1a.

- *Focusing.* Byzantine members collude and focus their effort on a single correct member in order to deny service to it or break assumed properties on the fraction of correct neighbors, as shown in Figure 3.1b.

- *Eclipsing.* Byzantine members arrange themselves such that a correct member has only Byzantine neighbors. In this case, they mediate most or all traffic to and from that member. Such attacks are known as *Eclipse attacks* [134]. If the Byzantine members are able to repeat such an attack for all correct members, they will end up with complete control of the overlay, as show in Figure 3.1c.

As argued in Section 2.2.2, an overlay link from member m_1 to member m_2 exists only because m_2 has authorized m_1 to connect and m_1 has selected m_2 as a neighbor. This implies that both m_1 and m_2 have knowledge of each other. To affect the overlay structure, an attacker can target the mechanisms that governs neighbor selection and knowledge maintenance. Neighbor selection attacks on member m include



(a) Byzantine members partition the correct members preventing them from communicating

(b) Byzantine members focus on a single correct member

(c) The Byzantine members are in complete control of the overlay

Figure 3.1: Undesirable overlay topologies

- having m select Byzantine members as neighbors, which is necessary to execute focus attacks.
- having m not select correct members as neighbors, which is necessary to execute eclipse attacks.

Attacks on knowledge maintenance include:

- Targeting the discovery mechanism such that correct members discover only Byzantine members.
- Targeting the failure detection mechanism such that correct members are considered crashed.
- Targeting the redirection mechanism such that correct members are directed to a set of only Byzantine members.

To be intrusion-tolerant, an overlay network must implement mechanism that prevent an attacker from using these topological functions to his advantage.

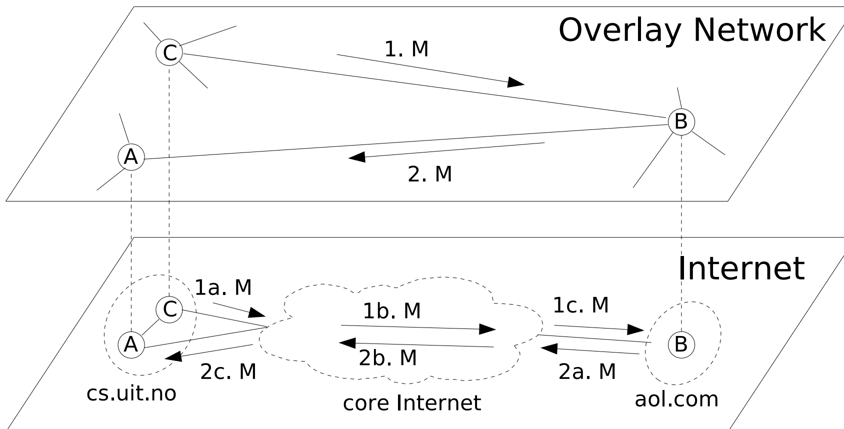


Figure 3.2: Locality in multi-hop overlay network routing

3.4 Messaging

Each routing hop in an overlay network increases the end-to-end latency and the redundancy required to ensure correct message delivery. We will in this section discuss the cost of multi-hop point-to-point messaging and show that it should be avoided in the critical path of intrusion-tolerant overlay-networks.

Proximity Routing

Because overlay links are independent of the underlying network, inefficient routing paths might occur. For instance, consider the three members A, B, and C in Figure 3.2. A and C are on the same subnet, and are well connected to each other. B is on a different subnet located in another continent such that it is connected to A and C through a low-bandwidth and high-latency link. The members have configured themselves within the overlay such that there is a link between C and B, and a link between B and A, as shown in the figure. In this scenario, if C sends a message to A, that message must be routed through B. This is inefficient because the message will be required to travel over the slow link twice even though its source and destination is on the same subnet.

Such inefficiencies can be mitigated by constructing the overlay link topology with the underlying network structure in mind. This is known as *proximity aware routing* [23, 67]. For instance, in the above example, routing

efficiency can be improved by adding an overlay link between A and B. Proximity aware routing is implemented in many routing substrates, including Chord, Pastry, and Tapestry. The routing algorithm of these systems is modified such that m will prefer forwarding messages to members who are close to m with regards to the chosen proximity metric. Metrics for closeness include the distance in the IP address space, or measured round-trip latencies.

Although proximity routing can improve performance of multi-hop routing, it also enables eclipse attacks because correct members can not determine whether or not routing table updates are legitimate or biased by an attacker. The effect of false membership information cascades with each subsequent update enabling the attacker to modify the overlay structure to his advantage [22, 134]. As such, proximity aware routing is at odds with using strict random structures to defend against membership attacks.

The Cost of Path Diversity

In overlay networks, each routing hop that is needed to deliver a message m to its destination increases the probability that m will be routed through a Byzantine member. Having Byzantine members in the routing path is not an ideal situation because they can not be trusted to forward or process messages correctly. For instance, in DHTs systems that provide location-independent routing like Pastry and Tapestry, an attacker can falsely claim to be the destination of all messages routed through him. He may falsely claim that object bindings do not exist or produce stale versions of stored objects.

In general, if a message is routed in h hops, and if each member has an uniform probability P_{byz} of being Byzantine, then the probability of a message being delivered successfully through a path of only correct members is given by $\sigma_h = (1 - P_{byz})^h$. Although, in many overlay networks, the number of routing hops increases slowly with the number of members (i.e., $\log N$ in Pastry), even a few hops significantly decreases the probability of successful routing. For instance, consider a Pastry overlay with 10^5 members and $P_{byz} = 0.1$. Simulations shows that the majority of messages in such a network require 4 routing hops [127]. In this case, $\sigma_4 = 0.66$. Hence, 34% of the messages would route through a Byzantine member.

Given R_h distinct and redundant routing paths in a h-hop routing overlay, the probability that a message is delivered through a correct path, γ, can be expressed as:

$$\gamma = 1 - (1 - \sigma_h)^{R_h}$$

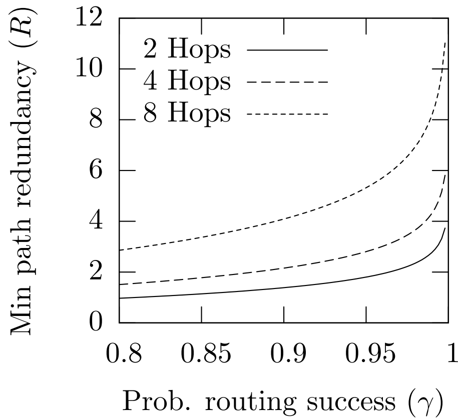


Figure 3.3: Required path diversity for multi-hop routing

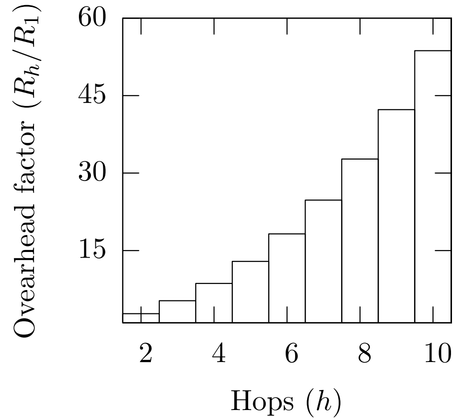


Figure 3.4: Messaging overhead due to multi-hop routing

Hence, the required number of routing paths given some γ is

$$R_h = \frac{\log(1-\gamma)}{\log(1-\sigma_h)} = \frac{\log(1-\gamma)}{\log\left(1-(1-P_{byz})^h\right)} \tag{3.1}$$

Figure 3.3 plots the required number of routing paths in a 4-hop overlay, R_4, for varying target probability of routing success, γ, with $P_{byz} = 0.1$ and using Equation 3.1. As expected, the required number of distinct routing paths grows quickly when $\gamma \to 1$. More alarmingly, the number of messages on the wire is $h \times R_h$. Routing overhead can then be calculated using R_1 as a baseline (i.e., when messages can be sent directly to each destination) as follows:

$$\frac{R_h}{R_1} = \frac{\log(1-\gamma)/\log\left(1-(1-P_{byz})^h\right)}{\log(1-\gamma)/\log\left(1-(1-P_{byz})^1\right)} = \frac{\log(P_{byz})}{\log\left(1-(1-P_{byz})^h\right)} \tag{3.2}$$

Figure 3.4 plots the multi-hop overhead factor calculated using Equation 3.2 when varying the number of hops from 2 to 10 and with $P_{byz} = 0.1$. As can be seen from the figure, it is imperative to keep the number of routing hops low. Ensuring with even a modest certainty that messages are delivered will significantly increase overall system load. Embedding in each message a route that has previously shown itself capable of deliver messages correctly can mitigate some of this cost, although establishing such routes is costly in a dynamic environment [107].

Object replication schemes that stores the replicas of each object o under different routing keys, as in Tapestry and the CAN system, make redundant routing easy to implement because the paths to each replica of o is, by design, likely to be diverse. Systems like Pastry and Chord, where o is replicated based on members' proximity to $o.key$, require each member to maintain redundant entries in their routing tables and use randomized message forwarding. To reduce performance degradation due to multi-hop routing in such systems, Castro et al. [22] suggest that redundant routing should only be used when normal routing fails. For this, they suggest a probabilistic routing failure test for DHT systems based on comparing the density of members around the sender with the density of members around the root. It is unclear how such a test helps when an attacker legitimately controls a replica and what impact the proposed protocol will have on messaging overhead.

Chapter 4

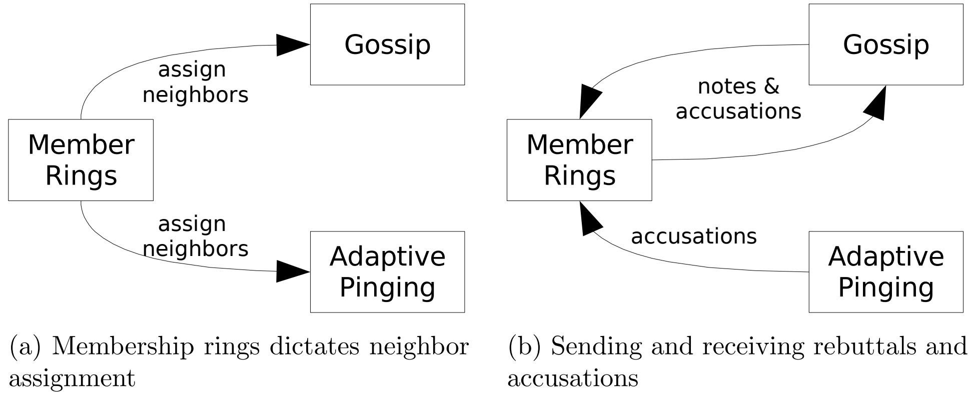
The *Fireflies* Membership Management Protocol

The previous chapters identified intrusion-tolerant membership management as a key function for intrusion-tolerant overlay networks. Chapter 3 stated three requirements. In this chapter we present the design of *Fireflies*, a group membership protocol that fulfils all three requirements. Our protocol can as such be used as a building block for intrusion-tolerant overlay networks.

4.1 Protocol Overview

The *Fireflies* group membership maintains at each member a fairly up-to-date view of all members. In essence, members monitor one another using an *adaptive failure detection* protocol and issue *accusations* (failure notices) whenever a member is suspected to have failed. When a member m_1 receives an accusation for a member m_2, m_1 waits a time period of length 2Δ before removing m_2 from its view. The value Δ is a probabilistic upper bound on end-to-end latency. Should m_2 receive an accusation about itself, then m_2 has the opportunity to issue a *rebuttal* before the timeout of 2Δ expires. The rebuttal will invalidate any previous accusations for m_2. Upon its reception, m_1 will stop its removal timer. Both accusations and rebuttals are disseminated to all members using a secure *gossip channel*.

Each member m is assigned a set of gossip partners and a set of monitors using a set of *member rings*, as shown in Figure 4.1a. The rings organize members into a *strict random structure*. Each member calculates similar rings based on their membership views such that they are able to *validate* inbound accusations, as shown in Figure 4.1b. A central *Certificate Authority (CA)* is used to assign member identities, which we will describe in the next section.



(a) Membership rings dictates neighbor assignment

(b) Sending and receiving rebuttals and accusations

Figure 4.1: The *Fireflies* membership protocol

4.2 Certificate Authority

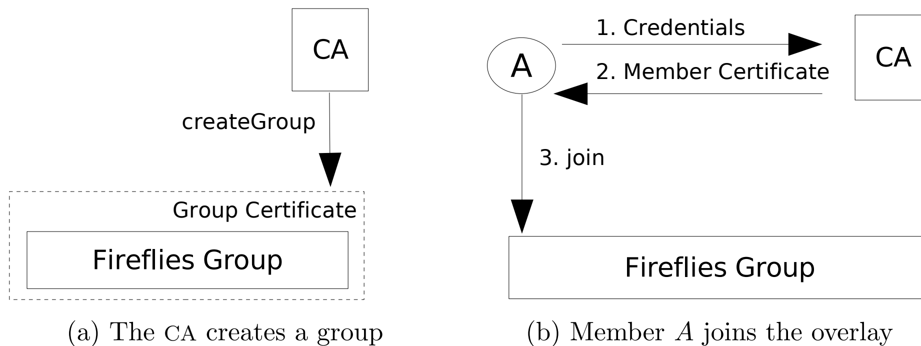
Fireflies uses a central Certificate Authority (CA) for the assignment of member identities. The CA enables *Fireflies* to fight the membership attacks listed in Chapter 3 by ensuring that each member receives a *random identity* and that there is a probabilistic upper bound, P_{byz}, on the fraction of Byzantine members among all members. By providing P_{byz}, the CA implements design Requirement 1 in Chapter 3. The assignment of random member identities is also necessary, although not sufficient, for implementing design Requirement 2. The CA ensures the probabilistic upper bound on the fraction of Byzantine member, P_{byz}, using a combination of *public-key cryptology* and *background identity checks*. We assume that the CA, correct members, and crashed members never reveal their private keys.

4.2.1 Certificates

A certificate is an unforgeable binding between a set of attributes and a public key attested by some private key using a digital signature. Having the corresponding public key, the integrity of the certificate can be checked [123]. *Fireflies* uses two types of certificates, which we will discuss in the following sections.

Group Certificate

The CA initiates a *Fireflies* group by creating a *group certificate*, as illustrated in Figure 4.2a. The group certificate contains the public key of the CA, a textual group name, a version number, and one or more configuration



(a) The CA creates a group (b) Member A joins the overlay

Figure 4.2: The role of the certificate authority

options. The group name allows a single public key to be used for multiple groups. For instance, if the CA is a news publisher, he might have one group for each news topic with names like "headlines," "sports," or "finances." Configuration options are convenient for transmitting global static information like default timeout values or the addresses of trusted *bootstrap nodes*. The version number allows the group certificate to be updated. The group certificate is signed with the private key of the CA and is made public such that all potential members may download it. It is the responsibility of each group member to check the validity of downloaded group certificates.

Member Certificate

To join, a process m first generates a public-private key pair $m.public$ and $m.private$. Next, m securely sends its credentials, including its public key $m.public$ and its network address $m.address$ to the CA. If the CA accepts m as a member, it generates and returns a *member certificate*, $m.certificate$, containing $m.public$, $m.address$, an expiry date, a version number, and a member identity. The identity is chosen at random by the CA such that m can not modify it. Alternatively, if the CA generates the public-private key pairs on behalf of m, the public keys, or their secure hash, could be used as member identities. However, since key generation is a CPU intensive operation, moving this responsibility to the members decreases the load on the CA thus making it more resilient to DoS attacks. Using identities computed from public keys when those keys are generated by members, is not safe because it allows an attacker to pick identities that are beneficial to him.

All member certificates are signed with the private key of the CA. Whenever $m.certificate$ expires, m must apply for a new member certificate. In this case, the CA sets a new date, increases the version number, and signs the

certificate. After member m has obtained a valid member certificate, m can join the group without further involvement by the CA, as shown in Figure 4.2b.

4.2.2 Bounds on the Fraction of Byzantine Members

In order to make it hard for an attacker to obtain a large number of member certificates, the CA is required to do a thorough background check on each potential member such that each member identity is tied to a real-world identity like a driver-licence number or a social-security number. Client puzzles [83] can also be used to increase the expense of obtaining an identity, although this increases the load for correct members. Physical artifacts like smartcards [51] might also be useful in practise since members must acquire and handle a physical artifact in order to join.

Although using the above techniques the CA can make it hard for an attacker to acquire control of a large fraction of the member identities, it can not in general prevent an attacker from joining. Hence, we must allow the CA to make mistakes at some rate. Given that, for instance, on average every 10th member that join is controlled by an attacker, then the overlay will contain about 10% Byzantine members. Because of the randomization of identity assignment, those members will be scattered uniformly within the *Fireflies* identity space. As such, we let P_{byz} denote the expected upper bound on the uniform probability that a randomly chosen live member is Byzantine. Although this is a stronger condition than a bound on the probability that *any* member is Byzantine, such a weaker condition would not suffice, as in the case that most non-Byzantine members have crashed, the few remaining correct members could be overwhelmed by Byzantine members. Nonetheless, the assumption that among all live members only a fraction is Byzantine is reasonable, particularly since we do not limit the fraction of crashed members among all members.

4.2.3 Revoking Certificates

Certificate revocation is desired when a Byzantine member has been identified and needs to be removed from the overlay. Also, when a correct member has detected an intrusion within its own software stack, it applies for a new member certificate, but it may want to revoke its old certificate as quickly as possible. While member certificates contain an expiration date and will automatically expire after some period of time, this might take too long in practice.

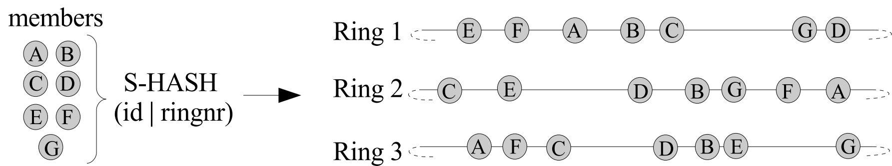


Figure 4.3: *Fireflies* mesh with three rings

In order to make immediate revocation possible, we essentially use certificate revocation lists. We employ two kinds of certificates: the public key certificates that we have already introduced, and *revocation certificates*, which contains the public key of the certificate that is being revoked, as well as a serial number of its own. Revocation certificates can be reliably distributed among correct members just like public key certificates.

4.3 Member Rings

Since Byzantine members might not accuse failed members in order to keep them in the group, each member must be assigned a sufficient number of monitors such that at least one is correct. However, there is a network overhead associated with gossip, so we have to prevent Byzantine members from submitting frequent accusations about correct members. This is a complicated issue because correct members might accidentally accuse other correct members due, for instance, to transient link failures. Thus not every false accusation is from a Byzantine member.

To solve this *Fireflies* organizes the members in a virtual pseudo-random mesh structure made up of k member rings. Each member ring is a circular address space where each member m sits between exactly two other members (assuming there are at least three members). Of those, one is called m's predecessor, the other is called m's successor. On each ring, m is responsible for monitoring its successor and will be monitored by its predecessor. The number of rings, k, can be adjusted to trade attack resilience with network overhead. The position of each member m on each ring is determined by applying a secure hashing function on $m.id$ concatenated with a ring identifier. As a result, a random mesh is formed where each member m has $2k$ neighbors whereof k are monitoring m (its predecessors), and k are monitored by m (its successors). As an example, consider the seven members A through G in Figure 4.3 securely hashed into three rings. The successors of C are $\{G, E, D\}$, and its predecessors are $\{B, A, F\}$.

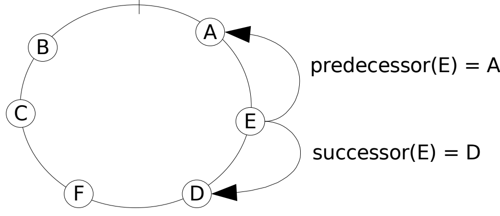


Figure 4.4: *Fireflies* membership ring

4.3.1 Formal Definitions

More formally, we define a ring r to be the graph $r = \{M, id\}$ where M, the set of n members, represents the vertices, and id is an unique ring identifier known to all members. The set of edges, E, connecting the members in M, is calculated deterministically from M and id. For this, we impose a total ordering, $<_r$, on the members, which is specific to each ring r. The ordering function $\mathcal{H}$ is specified by applying a Secure Hash Algorithm (SHA) on the concatenation ($\|$ symbol) of the members' identities and ring identity in the following manner:

$$\mathcal{H}(m, r) = SHA(m.id \parallel r.id) \tag{4.1}$$

The SHA function is required to provide a large address space with a low probability of collision. Hence, $\mathcal{H}$ defines a total ordering on the set of members that is different than the ordering of their identities. Given the two members m_i and m_j, the ordering is defined as:

$$m_i <_r m_j \Leftrightarrow \mathcal{H}(m_i, r) < \mathcal{H}(m_j, r) \tag{4.2}$$

The edges in E are defined by this ordering such that there is one edge between all adjacent members. Because $<_r$ is not circular, we also define an edge between the highest and lowest numbered member. More formally, there exists an edge between m_i and m_j iff there exists no other m_k such that $m_i <_r m_k <_r m_j$ or such that $m_i >_r m_k >_r m_j$. This results in a 2-connected Harary graph [71]—or a ring like structure as seen in Figure 4.4.

On each ring r we define the following relations:

- *Successor.* We say that m_j is the *successor* of m_i in ring r iff there exists an edge between m_i and m_j and either $m_i <_r m_j$ or there exist no $m_k >_r m_i$. Each member has exactly one successor in r.

- *Predecessor.* We say that m_j is the *predecessor* of m_i in ring r iff there exists an edge between m_i and m_j and either $m_i >_r m_j$ or there exist no $m_k <_r m_i$. Each member has exactly one predecessor in r.

- *Rank.* The rank relation adds transitive properties to the successor and predecessor relationships. We say that iff there are exactly x successor edges connecting m_i and m_j, then m_j rank from m_i is x. In this case, we may also say that m_j is m_i's x'th successor. While rank is primarily associated with the successor relationship, we will also refer to m_j's x'th predecessor in a similar manner. Note that although a member ranks itself as zero, we refer to a member's lowest ranked successor as its immediate successor.

For instance, consider the members A through E in Figure 4.4. The address space in the figure is drawn in circular clockwise manner. Then the successor of E is D since D is the next clockwise member from E. Member C has rank 3 from E since it is the third member from E. By combining k rings with different ring identifiers, each member m will be assigned k predecessors and k successors for a total of $2k$ neighbors.

4.3.2 The Probability of Having a Correct Monitor

Because a Byzantine member might not accuse crashed members in order to make them appear as live, we must make sure that all members have at least one correct monitor assigned to it. Due to the randomization of the $\mathcal{H}$ function and the assignment of random member identities using a trusted CA, each neighbor of member m is assumed to have an uniform and independent probability P_{byz} of being Byzantine. Hence, the probabilities on the number of Byzantine monitors of m has a *binomial distribution* [95].

Let X denote the binomial distributed random variable of the number of correct monitors of m in a mesh of k rings. The probability $Pr(X = t)$ that m has exactly t out of k Byzantine monitors is given by the binomial *probability density function*:

$$Pr(X = t \mid k) = \binom{k}{t} {P_{byz}}^{t}(1 - P_{byz})^{k-t}, \quad t = 0, 1, \ldots, k$$

The probability of a member m having no correct monitor can then be found by setting $x = k$, which gives

$$Pr(X = k \mid k) = \binom{k}{k} {P_{byz}}^{k}(1 - P_{byz})^{k-k} = {P_{byz}}^{k} \quad (4.3)$$

For example, if $k = 7$ rings are used and $P_{byz} = 0.10$, then the probability of m having no correct monitor becomes 10^{-7}. Hence, even a few rings can ensure that each member has at least one correct monitor with high

probability. A single correct monitor is sufficient to ensure that if m crashes, it will eventually be detected and subsequently excluded from the views of correct members. However, having a large number of rings does not prevent m from having Byzantine monitors assigned to it. Indeed, the probability of this happening increases with the number of rings:

$$Pr(X \geq 1 \mid k) = 1 - Pr(X = 0 \mid k) = 1 - (1 - P_{byz})^k \tag{4.4}$$

In the above example with $k = 7$ and $P_{byz} = 0.10$, the probability of m having a Byzantine monitor becomes 0.523.

4.3.3 Disabling Byzantine Monitors

A Byzantine monitor can falsely accuse those members that he is assigned to monitor. Although false accusations are rebutted by the accused member and will not affect the views of correct members, repeated false accusations can be used to execute a DoS attack. To deal with such attacks we allow each member m to disable those monitors that are falsely accusing it of having crashed. This must, however, be done in such a manner that m can not, intentionally or unintentionally, disable all its correct monitors. In that case, m could end up having only Byzantine monitors.

To solve this, let $k = 2t+1$ where t is the *maximum* number of Byzantine monitors that some member m can tolerate. Next, allow m to disable t of its monitors. Then, m can disable all of his Byzantine monitors. At the same time, even if m only disables correct monitors, one correct monitor remains. For instance, given 7 rings, m can tolerate having up to 3 Byzantine monitors. After disabling 3 monitors, m still has 4 active monitors, whereof at least one is correct.

In general, given $2t + 1$ rings, the probability that m has more than t Byzantine monitors is given by:

$$\begin{aligned} P\,(X > t \mid 2t+1) &= 1 - P\,(X \leq t \mid 2t+1) \\ &= 1 - \sum_{i=0}^{t} \binom{2t+1}{i} {P_{byz}}^{i} (1 - P_{byz})^{2t+1-i} \end{aligned}$$

We want to find the minimum number rings $2t + 1$ so that the above probability is smaller than some value ε. We also want to make ε smaller for larger sets of members, so that the expected number of members that have more than t Byzantine monitors does not grow linearly with the number of members N. For example, if we set $\varepsilon = 1/N$, then the expected number

```
N    // Number of members
Pbyz // Probability of a member being Byzantine
binom.cdf //cumulative function of the binomial distribution

def findK(Pbyz, N):
  for t in 1 to INF: //count to a large number
    if binom.cdf(t, 2t+1, Pbyz) <= 1/N:
      return 2t + 1
```

Figure 4.5: Algorithm for computing the required number of rings

of such unfortunate members is 1, independent of N altogether. Hence, we want to solve the following

$$\min_t : \varepsilon > \left(1 - \sum_{i=0}^{t} \binom{2t+1}{i} {P_{byz}}^i (1 - P_{byz})^{2t+1-i}\right) \times N \tag{4.5}$$

Binomial sums like the above can not easily be solved symbolically. Fortunately, statistical software packages that can compute such sums are readily available. If the cumulative binomial distribution is available, k can be found using the algorithm shown in Figure 4.5. In Figure 4.6 we have computed k for various N and P_{byz} using $\varepsilon = 1/N$. Note that k rises slowly with N.

4.4 Data Structures

The members are gossiping two kinds of data structures: *notes* and *accusations*. A member m creates accusations to convince other members that it has detected a crashed member. It creates notes in order to notify and convince the other members that it is live. In particular, m creates notes in order to rebut false accusations.

Notes. Each member m is represented in the system by a note. A note is a tuple (*cert, version, enabled*), signed using the private key of m. Here *cert* is the member certificate of m issued by the CA of the group as described in Section 4.2.1, *version* a monotonically increasing number used to order the notes of m, and *enabled* is a bitmap that controls which of m's monitors are allowed to issue accusations against m, allowing m to disable misbehaving monitors such that they can not repeatedly accuse it. By issuing a new note

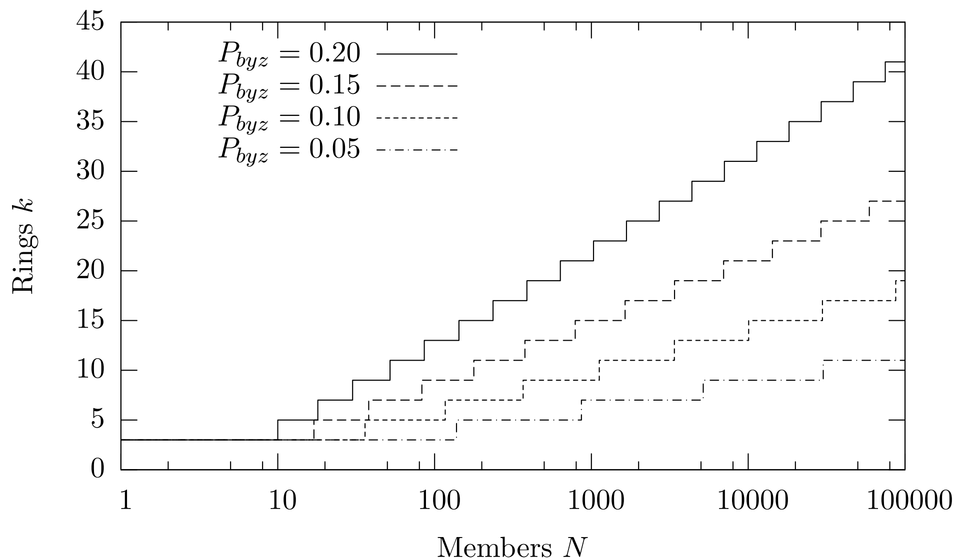


Figure 4.6: Required number of rings

with a larger version number, m will invalidate all accusations made on all of its previous notes.

Accusations. If a member m_1 suspects a successor m_2 on a particular ring of having crashed, then m_1 accuses the note of m_2 last known to m_1 by creating an accusation. An accusation is a tuple (*note, ring, accuser*), signed by m_1, where *note* is the note of m_2, *ring* is the ring on which m_2 is a successor of m_1, and *accuser* is the identifier of m_1.

Views. Each member m_1 maintains a view $m_1.view$ containing information on all overlay members known to m_1. The value $m_1.view[m_2] = \perp$ means that m_1 does not have any information about m_2. For each member m_2 known to m_1, m_1 maintains in its view m_2's most recent note and any accusations for m_2. The *Fireflies* protocol strives to ensure that the set of accusations is empty for a correct member and non-empty for a crashed member. If m_1 is monitoring m_2, then m_1 also maintains the number of pings sent to m_2 and measured packet-loss rate, as will be described in Section 4.7. The fields in the member structure are summarized in Figure 4.7.

note	most recently known note of the member
accusations	accusations, at most one per ring
nPings	#pings sent since last “pong” response
avgLoss	smoothed average of #pings lost + 1

Figure 4.7: Basic *Fireflies* member structure

4.5 Valid Accusations

A valid accusation indicates that the accused member might have crashed. Upon member m_1 receiving a valid accusation for member m_2, m_1 starts a removal timer. If m_1 has not received a rebuttal for that accusation within 2Δ time, then the removal timer will expire and m_1 will consider m_2 as crashed. If m_2 is in fact live, then it will rebut the accusation before the removal timer expires.

As we have not bounded the probability that a member is crashed, all predecessors of a member may be crashed with non-negligible probability. In order to allow such members to be accused in case they fail, a member must be able not only to accuse its immediate successor, but must also be able to make accusations skipping over crashed successors. Doing so may allow a Byzantine member to accuse any of its successors simply by claiming that it believes that the more immediate successors are all crashed.

In order to counter such attacks, we create rules that govern which accusations are considered *valid*. Informally, m_1 only allows the *highest ranked* live member to make *valid* accusations of m_2, and only on those rings that are enabled by m_2. Validity is defined recursively. Member m_1 considers an accusation for m_2 valid *iff*

- the accusation is correctly signed; *and*
- the note in the accusation corresponds to $m_1.view[m_2].note$; *and*
- the ring in the accusation is enabled in the note's *enabled* bitmap; *and*
- m_1 holds valid accusations for all members it ranks (on the given ring) between the accuser and m_2 itself, if any.

In Figure 4.8, we show a schematic depiction of how one of the members observes a group with 7 members, A through G. Valid accusations are shown with solid arrows, while invalid ones are shown in dashed arrows. In this case, $k = 3$, and for simplicity we ignore ring deactivation. An accusation

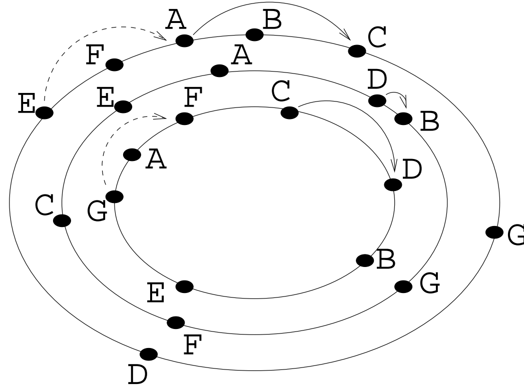


Figure 4.8: Example of valid and invalid accusations

of B by D on the middle ring is a valid accusation (assuming the accusation refers to the note of B and is correctly signed by D) because there are no nodes in between D and B. This accusation is valid even though the accuser D is validly accused by C. The accusation by A of C on the outer ring is valid because there is a valid accusation against B, the node in between A and C. The accusation of A by E is invalid as there is no valid accusation of F.

4.6 Blocked Accusations

Even if a crashed member m_1 has fewer than $t+1$ Byzantine live predecessors, it is possible that an enabled correct live predecessor m_2 can not accuse m_1. Consider the crashed members between m_2 and m_1 on the corresponding ring and call them $s_1, \ldots, s_m$. If the ring is enabled for all s_i, then m_2 can accuse all s_i and thus m_1. Assume there is a member s_j that disabled the ring. Then m_2 can not accuse m_1 until m_2 has received a valid accusation for s_j on a different ring. We say that the accusation of m_1 is blocked by s_j. Unfortunately, m_2 may never receive an accusation for s_j. Member s_j may have fewer than $t + 1$ correct live predecessors, all of which being disabled. It may even be that the accusation of s_j is blocked by m_1, thus creating a loop preventing both m_1 and s_j of being accused.

To find how likely this situation is to occur, we constructed a simulation. Initially, all members are correct and are present in all the views. At time $T = 0$, 75% of the members crash, 20% of the remaining members become Byzantine. In order to generate a worst-case scenario, the correct members disable as many correct predecessors as possible, and the Byzantine members

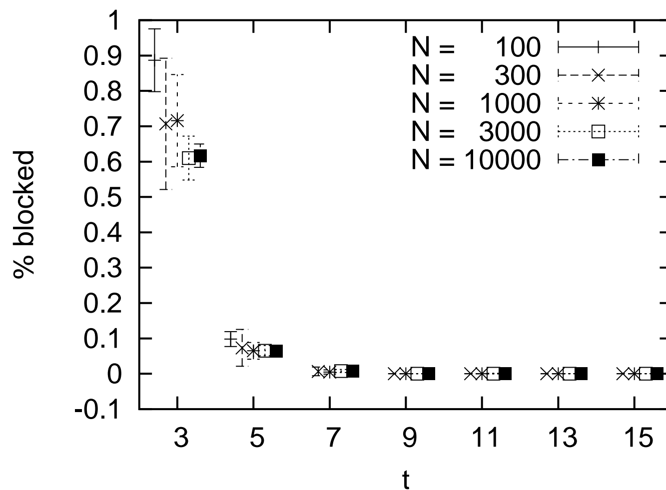


Figure 4.9: The likelihood of blocked accusations

do not emit any accusations. In Figure 4.9, we show the results of this simulation. The graph shows that even for large N, a relatively small value for t makes blocked accusations highly unlikely.

4.7 Failure Detection

To detect crashed members, members monitor one another. Essentially, a member m_1 monitors some other member m_2 by probing it at regular intervals. Each probe involves m_1 sending a ping message to m_2. If m_2 is correct, it returns a pong message. If m_2 is crashed, it will not answer. A probe is only successful if both the ping and the pong messages are delivered. In the time period between the ping and the pong message, we say that a probe is pending or running. After some period of time, a pending probe will time-out and will be considered failed.

Because the Internet is a best effort network, messages can be delayed or lost. Hence, if a probe fails, m_1 should repeat it. Only after τ consecutive probes have failed, then m_1 considers m_2 crashed.

Using a static global time-out for τ is, however, not a good choice as members might experience different packet-loss rates and end-to-end latencies. A poorly chosen time-out value will cause correct members to either accuse live members too often, resulting in unnecessary network traffic, or cause correct members to accuse failed members too rarely, allowing them to remain in the views. As such, the time-out value τ should be adapted to the

characteristics of each individual monitoring link.

4.7.1 Setting the Time-out Threshold τ

Bolot [19] shows that the loss of probe packets is essentially random when the probe traffic consumes less that 10% of the available network bandwidth. Also, Barford and Sommers [12] show that the overall loss-rate is stable. As such, we model probing as a *negative binomial experiment* with parameters $r = 1$ and the probability of a probe succeeding, $P_{success}$, reflected in the measured packet-loss rate.

A successful probe requires that both the ping and the pong message are delivered. Hence, the packet-loss probability rate P_{loss}, and the probability of a successful probe $P_{success}$ are related by $P_{success} = (1 - P_{loss})^2$.

Let X denote the random variable of the number of probes required to succeed. For instance, if a link has no packet loss, then $X = 1$. As a negative binomial experiment, the probability that the probe succeeds at x attempts is given by:

$$Pr(X = x) = (1 - P_{success})^{x-1} P_{success}, \quad x = 1, 2, \ldots$$

If m_1 repeats a probe τ times and m_2 is indeed alive, the probability that at least one probe succeeds is given by

$$\begin{aligned} Pr(X \leq \tau) &= \sum_{x=1}^{\tau} (1 - P_{success})^{x-1} P_{success} = P_{success} \sum_{y=0}^{\tau-1} (1 - P_{success})^{y} \\ &= P_{success} \frac{1 - (1 - P_{success})^{(\tau-1)+1}}{1 - (1 - P_{success})} = 1 - (1 - P_{success})^{\tau} \end{aligned}$$

Hence, if after τ failed probes, m_1 decides that m_2 has failed, the probability that m_1 is wrong, and the all probes have failed due to packet loss, is given by

$$P_{mistake} = 1 - Pr(X \leq \tau) = (1 - P_{success})^{\tau} = (2P_{loss} - P_{loss}^2)^{\tau} \qquad (4.6)$$

If m_1 wants to establish with certainty $P_{mistake}$ that m_2 has failed, then the number of consecutive probes it must submit is given by:

$$\tau = \frac{\log (P_{mistake})}{\log (2P_{loss} - P_{loss}^2)} \qquad (4.7)$$

As shown in Figure 4.10, the threshold τ increases exponentially with the P_{loss}. As such, we can not effectively determine a host failure with a high accuracy when packet loss is high.

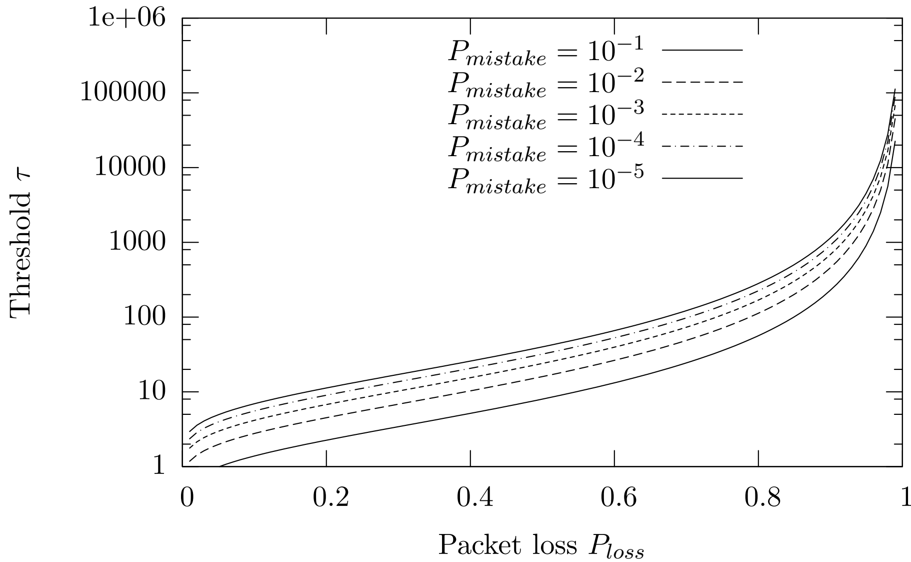


Figure 4.10: Failure detection threshold τ as a function of packet loss

4.7.2 Rounding Error

The output of Equation 4.7 will produce values that might contain fractions. For instance, if $P_{mistake} = 10^{-4}$ and $P_{loss} = 0.10$, then $\tau = 5.546$. Clearly, m_1 can not probe m_2 5.546 times. Instead, it must choose either 5 or 6. In either case, a rounding error is introduced. Because P_{loss} is determined by the packet-loss rate of the underlying network, it can not absorb this error. Hence, the error must be absorbed by $P_{mistake}$. Say that m_1 chooses τ to be 5. For this to occur, Equation 4.6 tells us that $P_{mistake} = 2.47 \times 10^{-4}$. In other words, even though m_1 configured $P_{mistake}$ to be 10^{-4} the observed $P_{mistake}$ will be 2.47×10^{-4}, which is 2.47 times higher. If m_1 had chosen τ to be 6, the observed $P_{mistake}$ would be 4.70×10^{-5}, which is 0.470 times higher.

Figure 4.11 shows the observed $P_{mistake}$ when converting τ to an integer using three different functions.

- *Rounding,* uses the integer that is numerically closest to τ. This causes the observed $P_{mistake}$ to oscillate around the configured $P_{mistake}$ as shown in Figure 4.11a.

- *Flooring,* uses the largest integer smaller than τ. As shown in Figure 4.11b, flooring also causes oscillations but the observed $P_{mistake}$ is always larger than the configured $P_{mistake}$.

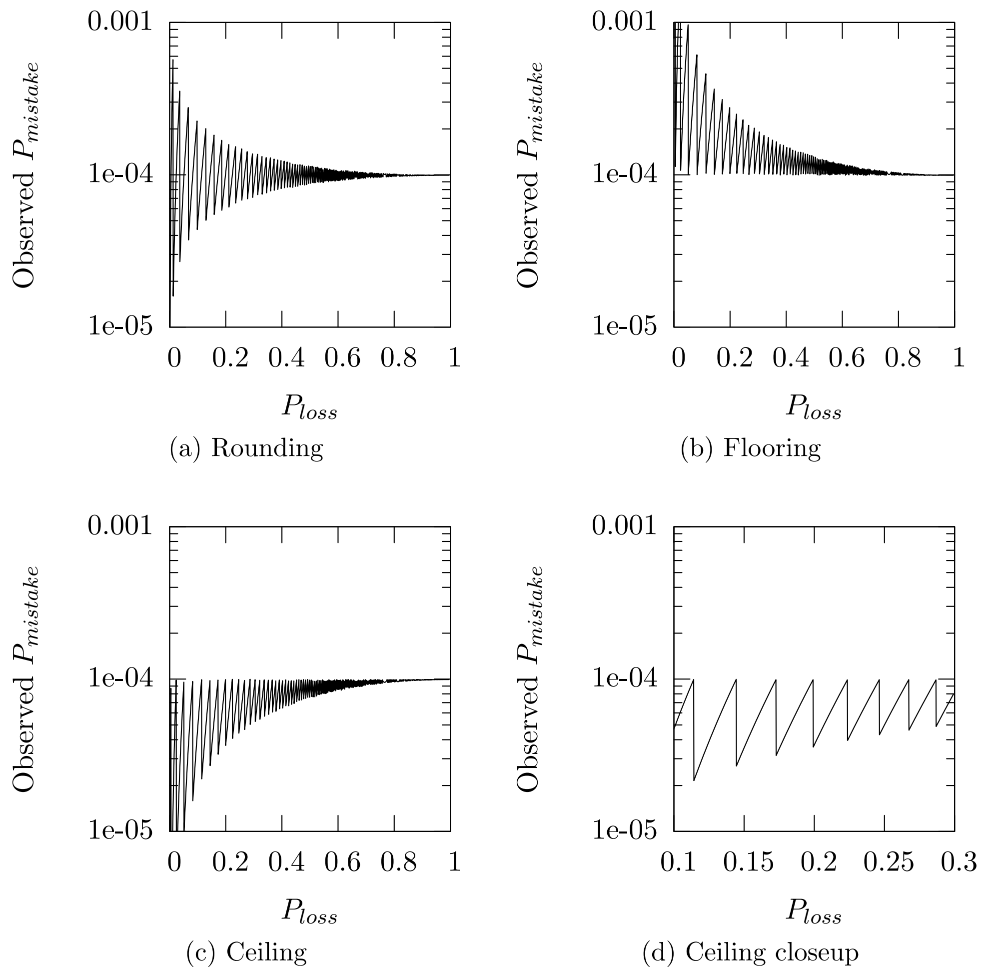


Figure 4.11: The effect of rounding error on adaptive pinging

- *Ceiling,* uses the smallest integer larger than τ. As shown in Figure 4.11c, ceiling behaves similarly to flooring except that it always remains below the configured $P_{mistake}$. Figure 4.11d shows the ceiling function in a shorter interval.

Although the choice of which of these functions to use can be left to higher level services, we will in the remainder of this dissertation take a conservative approach and use the ceiling function.

4.7.3 Estimating Packet-Loss Rate

The calculations above relies on knowing the packet-loss rate. For this, we estimate $P_{success}$, the probability of a probe succeeding, by measuring the number of probes that m_1 sends before it receives a response from m_2. For negative binomial experiments, the average number of trials required for before a success is given by $E(X) = \frac{1}{P_{success}}$. By substituting this into Equation 4.7 we get

$$\tau = \frac{\log(P_{mistake})}{\log\left(1 - \frac{1}{E(X)}\right)} \tag{4.8}$$

The value for $E(X)$ can be estimated by m_1 by recoding the difference in sent ping messages and received pong messages. For instance, if m_1 sends 6 pings to m_2, but receives a pong from m_2 only for the last ping, then m_1 concludes that 1/6 of those ping messages were lost in the network.

To estimate future loss rate, we use the *simple exponential smoothing model*. That is, if $E_i(X)$ is the current expected value, x is the number of pings sent before a pong is received and α is the smoothing factor, then

$$E_{i+1}(X) = \alpha E_i(X) + (1 - \alpha)x \tag{4.9}$$

To measure the effectiveness of our adaptive pinging protocol, we constructed a simulation where a member m_1 monitored some other member, m_2. The smoothing factor α was set to 0.99995 and the pinging interval was set to 1 second. Packet loss was random and both members were correct during the course of the experiment. Figure 4.12 shows the observed rate at which m_1 made pinging mistakes when the packet-loss rate, P_{loss}, varies stepwise between 5% and 40%. As expected, the protocol will adapt over time to quick changes in packet loss rate by adjusting the timeout threshold τ. The figure also shows the expected rate of mistakes after adjusting for the τ rounding error. Although in this particular experiment adaption is slow, quicker response time can be achieved by choosing a lower α value.

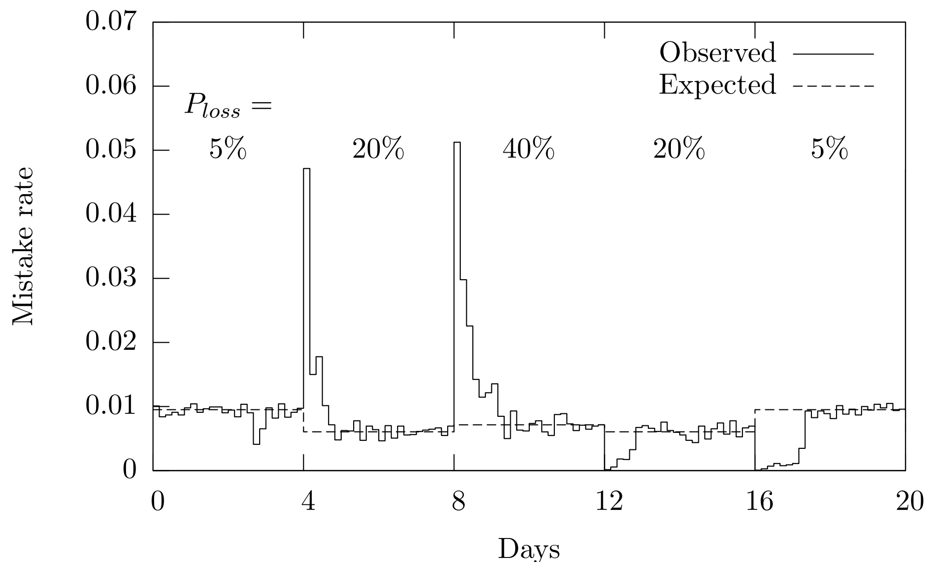


Figure 4.12: Adapting timeout threshold to packet loss rate

4.7.4 Threshold Limits

If packet loss is very low, τ would be set unrealistically low. With no packet loss ($P_{mistake} = 0$), τ would even be undefined. We address these issues by imposing a minimum threshold τ_{min}. Similarly, if packet loss is very high, then τ will be set unrealistically high. We therefore impose a maximum threshold τ_{max}.

4.7.5 Pinging Attacks

Byzantine members could potentially prevent detection of crashed members by forging pong messages. This is prevented by having each ping message contain a nonce that has to be signed by the monitored member and returned in the corresponding pong message. This strategy prevents both forging of pong messages and replay attacks.

Byzantine members can, however, generate a modest amount of overhead on the system by not responding to ping messages from correct members, and rebutting the ensuing accusations. Such "nuisance attacks" are easily identifiable, and such members can be removed by revoking their public key certificates.

4.8 Gossip

A gossip protocol is a simple group communication protocol whereby each member periodically picks a random member from its view and exchanges state information. Such protocols are known to be highly robust, as they are essentially flooding protocols. But unlike flooding protocols, they are efficient with probabilistic bounds on delivery latency [85]. In our particular situation, we have to concern ourselves with Byzantine members.

All notes and accusations are signed, and because we assume that Byzantine members can not break the cryptographic building blocks, we do not have to worry about impersonation attacks. We have also assumed that trivial DoS can be detected and suppressed. But Byzantine members can still attack the gossip protocol in the following two ways. In order to slow down dissemination, they can neglect to forward recent updates. This slowdown can be incorporated in the calculation of the upper bound on message dissemination, Δ. Byzantine members can also pretend that they have no information, causing correct members to transmit their entire state to them and thus causing unnecessary load on the correct members and on the network. In order to reduce the opportunity for Byzantine members to launch this attack, we will consider gossip protocols in which each member can only gossip with a small subset of the membership. Kermarrec et al. [85] show that it is possible to build effective gossip protocols if each member only has a small set of uniformly chosen members it gossips with. Each member m selects k gossip *neighbors* from its view uniformly at random. For such a scheme to work, k must be large enough to create a connected graph of correct nodes.

4.8.1 Ensuring Connectivity

A classic result of Erdös and Rényi [54] shows that in a random graph of n nodes, if the probability of two nodes being connected is $p_n = (\log n + c + o(1))/n$, then the probability of the graph being connected goes to $\exp(-\exp(-c))$.

The number of correct members, n, is expected to be at least $(1 - P_{byz}) \times N$, where P_{byz} is the configured upper bound on the probability that a live member is Byzantine, and N is the total of the correct and the Byzantine members. Then the probability that one member is connected to another is $1 - (1 - 1/N)^k \approx k/N$. Thus $p_n \approx 2k/N$. We assume that every correct member can connect to every other correct member. This assumption can be relaxed, but p_n has to be adjusted accordingly. In order for the correct

members to be connected with probability φ, we obtain

$$k \geq \frac{N}{2n} \cdot \left(\log \frac{-n}{\log \varphi} + o(1) \right)$$

4.8.2 Pseudo-Random Mesh

The analysis of the gossip protocol above tacitly ignores the possibility of a Byzantine member selecting more than k neighbors in order to increase the overall load on the correct members. Also, Byzantine members could "gang up" on a small set of correct members, overwhelming them with gossip load [11]. In order to fight such membership attacks, we introduce a rule that determines who can gossip with whom. We use the same technique that we used in Section 4.3 to assign monitors, except that we use a different number of rings.

On each ring, a member initiates gossip only with the first successor in its view. For ring r, a member m_1 sets up a secure mutually authenticated connection with its successor m_2 using their private and public keys. Member m_1 then sends $m_1.note$ and the ring identity to m_2, so that m_2 can add m_1 to its view if necessary and possible (existing accusations of $m_1.note$ might prevent this). Member m_2 checks that it is a successor of m_1 on the proposed ring using its local membership view.

One complication is that even when m_1 and m_2 are both correct, they may have different views. In particular, m_2 may know a "better" gossip neighbor m_3 for m_1 that is not in m_1's view. If such is the case, m_2 sends $m_3.note$ to m_1. Should m_1 have plausible accusations for m_3, then it returns those to m_2 and terminates the attempt to gossip. If no such accusations exist, then m_1 was unaware of m_3. In that case m_1 adds m_3 to its view and tries to gossip with m_3 instead.

If at any point in time m_1 should determine a better gossip neighbor for ring r than m_2, then m_1 terminates the existing connection. Note that newly joining and recovering members should gossip with at least $t + 1$ different members before they can be reasonably certain that they will be integrated into the "true" membership, as opposed to a fake membership created by Byzantine members [134]. Gossip neighbors are thus chosen from a convenient low-diameter mesh that connects the correct members.

4.8.3 Time-out value Δ

Next we determine the resulting Δ, the time to disseminate a message in a random graph. To better preserve resources, each member does not update

all its k outbound neighbors in each round, but instead selects one neighbor for each round in a round-robin fashion. Conservatively, we will assume that it takes k rounds to update all gossip neighbors, and thus the dissemination runs a factor k slower than if all neighbors were updated in each round. If d_n is the diameter of the graph of correct members, then the expected amount of time to disseminate an update reliably among the correct members is therefore $\Delta = k \times d_n$.

An asymptotic value for the diameter of the resulting graph d_n can be determined. A recent result of Chung and Lu [34] shows that if $np_n \rightarrow \infty$ (which in our case it does), then the expected diameter of our graph is given by

$$d_n = (1 + o(1)) \ \frac{\log n}{\log np_n}$$

Unfortunately, it does not provide the constants needed to tune the mesh. In order to find suitable constants, we ran simulation experiments with N ranging from 16 to 16,384 for varying P_{byz} and with k chosen as above (ignoring the $o(1)$ term), to determine if the resulting graphs of correct members are indeed connected and to obtain values for Δ. We ran each experiment 100 times. We encountered no disconnected graphs in any of our 3000 experiments. In Figure 4.13 we report the maximum number of required gossip rounds that we observed for each N and P_{byz} with $\varphi = 0.99999$.

4.8.4 Communication Efficiency

Even if both m_1 and m_2 are correct, communicating all information back and forth is inefficient, as most of the notes and accusations held by m_1 and m_2 are likely to be the same. Determining the differences of two set is a case of what is known as *set reconciliation*. There exist protocols that reconciles sets of information by only exchanging an amount of information that is on the order of the size of the difference between the sets [21, 102, 103]. Unfortunately, set reconciliation can be computationally expensive, and so we minimize its use.

Set reconciliation on the entire information of m_1 and m_2 must be done when a connection between these members is first created. From then the potential set of differences is greatly reduced, and it may be sufficient to send only updates along the connections. It is possible to reduce communication overhead further by performing set reconciliation iteratively on the updates that occurred since the last gossip exchange between the members.

Byzantine members can attack this protocol by alternately appearing correct and crashed, causing gossip connections to be set up and broken

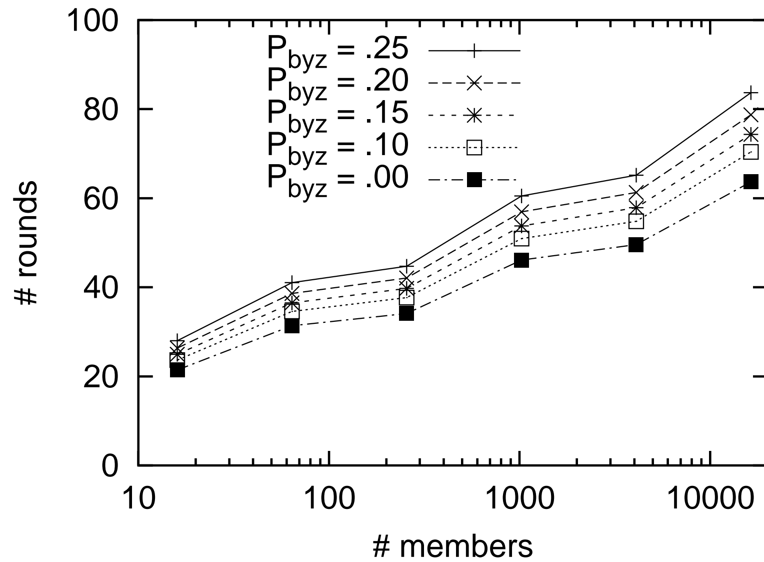


Figure 4.13: Number of rounds required to disseminate an update

repeatedly, necessitating new set reconciliations. In order to prevent correct members from falling into this trap, they maintain in each member's *info* structure a smoothed average of the rate of information sent to this member, and impose an upper limit on the rate using a leaky bucket flow control mechanism.

4.9 Protocol Steps

In this chapter, we have described *Fireflies*, a protocol that correct members follows in order to maintain up-to-date membership information in an intrusion-tolerant manner. There are four event types that trigger transitions in our protocol:

- **Member m_1 receives a note for member m_2.** If m_1 has a note for m_2 that is as recent as the one that arrived, then m_1 ignores it. Otherwise m_1 updates its note for m_2, removes any accusations that it has for m_2, cancels m_2's view removal timer if any, and includes m_2 in its view. In addition, the removal of accusations for m_2 may invalidate accusations that m_1 holds for other members. These accusations are removed as well.

- m_1 **suspects** m_2**.** On each ring, m_1 monitors the lowest ranked successor m_2 for which m_1 does not hold valid accusations (unless m_2 has disabled the ring, in which case m_1 does not monitor anybody on that ring). Should m_1 suspect that m_2 has crashed, then it creates an accusation of m_2 that is subsequently gossiped to the other members.

- m_1 **receives an accusation for** m_2**.** If m_1 does not consider the accusation valid, then m_1 ignores it. If $m_2 = m_1$, then m_1 replaces its note with a new one to act as a rebuttal, which is subsequently gossiped to the other members. If $m_2 \neq m_1$ and m_1 already has an accusation for m_2 on the same ring as the new accusation, then m_1 replaces its accusation only if the new one is from a higher ranked accuser. Otherwise m_1 accepts the accusation and sets m_2's view removal timer to 2Δ.

- m_1**'s view removal timer for** m_2 **expires.** m_1 removes m_2 from its view.

Using our protocol, an intrusion-tolerant overlay network that fulfills all design requirements stated in Chapter 3 can be constructed. Requirement 1 is fulfilled by the central CA, Requirement 2 is fulfilled by the *Fireflies* monitoring and gossiping scheme, and Requirement 3 is fulfilled by providing to each member a up-to-date view of all participating members.

Chapter 5

FiRE: The *Fireflies* Runtime Environment

This chapter describes FiRE, a toolkit and runtime environment that facilitates the construction of intrusion-tolerant overlay networks. FiRE uses the *Fireflies* protocol, which we described in Chapter 4, to maintain intrusion-tolerant membership information.

5.1 Overview

FiRE is a library that executes in-process with application processes. It is implemented in the Python programming language [147] using the Twisted event-based networking framework [96]. Twisted provides to FiRE an event loop, tools for scheduling and handling events, and structures for implementing low-level networking protocols. Using Python, FiRE can run on a large number of platforms, including: Windows, Linux/Unix, and Mac OS X.

The primary function of FiRE is to maintain the overlay-network structure in an intrusion-tolerant manner by managing membership information and neighbor selection as described in Chapter 4. FiRE is made up of the following four components:

- A *Certificate Authority (CA) component* that generates and manages group and member certificates.
- A *membership component* that implements the membership rings and the rules governing valid accusations and the selection of neighbor.
- A *failure detection component* that monitors neighbors using adaptive pinging.

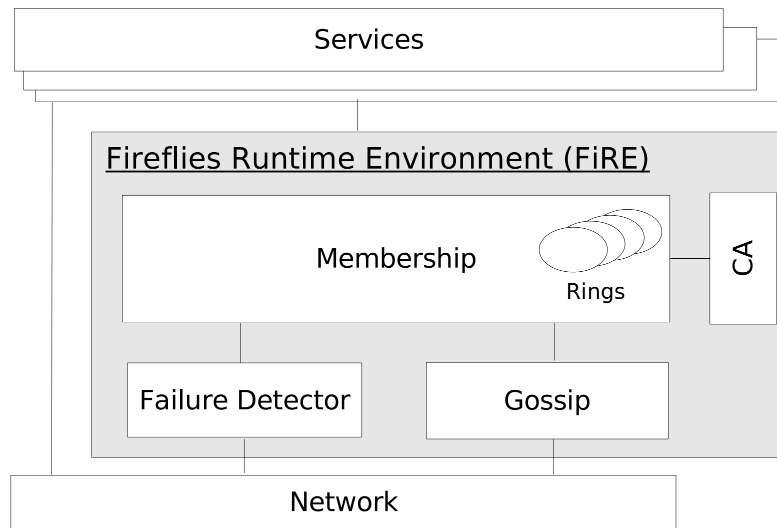


Figure 5.1: Architectural overview of FiRE

- A *gossip component* that distributes notes, accusations, and member certificates to all correct members.

Application-specific logic is implemented on top of FiRE as one or more *services*. Services are program objects that register themselves with FiRE for maintenance of membership information. For this, services are required to implement an Application Programming Interface (API) containing a few call-back functions. Other than this API, FiRE puts few restraints on how services are implemented and what actions they perform. For instance, a service may communicate with other in-process services using shared variables or interact with the underlying operating system in order to read and write files. Services may communicate over the Internet using both TCP or UDP, or higher-level protocols like HTTP. A service can provide generic messaging functionality, like application level multicast, or implement high-level application logic, like graphical user interfaces.

We have implemented several services. For instance, we have constructed a web service that inspects the membership state and generates HyperText Markup Language (HTML) pages upon HTTP requests. This service has been beneficial to us in order to debug our code and to monitor our experiments. We have also created several multicast services. In particular, we have created a patch distribution service, which will be described as a case study in Chapter 7. The relations between the components within FiRE and with services are summarized in Figure 5.1.

The membership component has a central role in the runtime. Its primary

function is to turn inbound certificates, notes, and accusations into high-level membership events as specified by the *Fireflies* protocol. For instance, the membership component will generate a failure event if an accusation has not been rebutted within the set timeout 2Δ.

Services use FiRe to maintain up-to-date membership information. For this, FiRe can either be pulled periodically, or a service can register itself to receive callbacks whenever there are membership changes. In particular, a service can register itself such that FiRe maintains its list of neighbors. In this case, the service receives callbacks only when it needs to establish or break overlay-network links. The service can instruct the runtime to maintain the neighbors such that it, for instance, has at least one correct neighbor with high probability. The failure detection component and the gossip components are implemented using this feature.

The membership component is not aware of the underlying network. All communication goes through the failure detection and gossip components. Upon notification from the failure detection protocol that a monitored member is suspected of having failed, the membership component will generate an accusation for that member. The gossip component maintains a set of notes, a set of accusations, and a set of member certificates. When gossiping, data sets are compared and missing data elements are exchanged. These are handed to the membership component and membership information is updated accordingly.

5.2 Global Data Structures

FiRe defines many data structures and constants. We will in this section describe those that are most important.

5.2.1 Configuration Options

FiRe can be tuned to meet particular application requirements using a wide range of configuration options. In particular, the number of rings can be adjusted to trade intrusion-tolerance for scalability. All configuration options are included in the group certificate as default values, as described in Section 4.2.1. Some values, like timeouts, are flexible in that members can, to some extent, override them without ill effect. Other values, like the number of monitoring rings, are absolute in that members that modify them are considered Byzantine. Although these constants are described in the previous chapters, we summarize them in Figure 5.2.

Symbol	Description
P_{byz}	Upper bounds on the probability that a member is Byzantine
N	Maximum number of members
Δ	Upper bound on the time to distribute a membership event
$P_{mistake}$	Target probability of making a pinging mistake
α	Smoothing factor for adaptive pinging
τ_{min}	Minimum number of pings
T_{ping}	Pinging interval in seconds
T_{gossip}	Gossip interval
$T_{timeout}$	TCP connection timeout
φ	Target probability of having a connected gossip mesh

Figure 5.2: Configuration options

5.2.2 Data Objects

FiRe defines and uses many data structures and objects. Important structures include those that are communicated to other members. These are member certificates, accusations, and notes. In addition, the *Fireflies* protocol defines a group certificate. Figure 5.3 summarizes the content of these structures as used by FiRe. We will in the following describe important details relating to the implementation and usage of these structures, and show how the FiRe structures differ from those described in Chapter 4.

Fireflies use public key encryption to accomplish its goals. For signatures FiRe use 160 bits SHA-1 [53] and RSA-1024 [123] with 1024 bit keys. Both algorithms are widely available and are commonly used [13]. Recently, SHA-1 has become obsolete as collisions can be found with less than 2^{69} operations [151]. Also, a mechanism that can break RSA-1024 keys within reasonable time is proposed in [133]. FiRe is constructed with such advances in mind and can easily make use of new cryptographic tools and larger key sizes.

An early FiRe prototype used Python's standard *pickle* functions for marshaling communicated data elements. The pickle functions are flexible and easy to use but do not marshal objects into compact structures. Pickled Python objects are represented in a printable ASCII data format and include type information. Using pickle, FiRe data structures measured in the order of several Kilobytes (kBs). By using handwritten binary marshaling routines, we gained a factor of 20 in network efficiency. Our data format conforms to the External Data Representation (XDR) standard [137] such that it can be parsed on many computer architectures.

Public keys and member certificates are large objects when compared to

Attribute	Description
name	Group name
public	Public group key
config	Configuration options (see Section 5.2.1)
version	Version number
signature	Signed with the private key of the CA

(a) Group Certificate

Attribute	Description
identity	Member identity
date	Valid to date
public	Member's public key
address	Member's network address
version	Version number
signature	Signed with the private key of the CA

(b) Member Certificate

Attribute	Description
identity	Member identity
version	Note version number
enabled	Bitmap mask of enabled/disabled rings
signature	Signed with member's private key

(c) Note

Attribute	Description
accuser	Identity of the accuser
accused	Identity of the accused
version	Version number of the accused's note
ring	The ring number on which the accusation is made
signature	Signed with the accuser's private key

(d) Accusation

Figure 5.3: FiRE data structures

hashes and signatures. All certificates created by the CA contain a 20 Byte (B) member identity that uniquely identify the certificate and its embedded public key. We have ignored this in Chapter 4, but in practice we can improve communication efficiency by replacing the larger certificates with this smaller member identity.

By replacing the full member certificate with the corresponding member identifier, the size of a note is reduced with 143 bytes. Also, because accusations have embedded the note of the accused member, their size will be reduced as well. However, a further reduction on the size of accusations can be accomplished by removing the notes from the accusations altogether. To identify the note of the member that is accused, *m.note*, it is sufficient for an accusation to contain only the accused member's identity, *m.id*, and the version number of the accused's note, $m.note.version$. An accusation also contains the public key of the accuser, which is replaced with the accuser's identity.

One possible complication of this optimization is that accusations and notes are not self-contained. The implication is that the validity of an accusation can not be established without previously having received the accused note. Similarly, a member can not ascertain the validity of a note without previously having received the corresponding member certificate. We address this problem by specifying that correct members must exchange data in the following order: first member certificates, then notes, and then accusations. The corresponding group certificate must be obtained prior to joining using an external trusted channel.

These above optimizations resulted in a public key certificate of 163 bytes, a note of 49 bytes, and an accusation of 52 bytes.

5.2.3 Member Object

Each group member is represented within FiRE by a member object that encapsulates known state for that member. Many of the provided functions take member objects as arguments. The attributes of a member object is shown in Figure 5.4.

Services typically extend the member object with application specific attributes. For instance, a broadcast service might associate a vector clock with each member. Member objects are created whenever a member certificate, with a previously unseen member identity, is added to FiRE. Hence, services do not create member objects directly.

Because notes and certificates are exchanged separately, member objects can exist without having a proper note set. In this case, the member has an implicit note with version number 0, and is considered crashed. Explicitly

Attribute	Description
certificate	The member's latest member certificate
note	The member's latest note
accusations	List of all current valid accusations
failed	True if the member is considered not live
nrPings	The number of unanswered pings
avgLoss	The measured average loss rate

Figure 5.4: FiRe member object

created notes have version numbers larger than 0. Hence, accusations for notes with version number 0 are considered as Byzantine behavior and are ignored by correct members. When creating a note, for instance in response to a false accusation, a member increases its version number with 1.

5.3 Main Functionality

In this section we describe the core functionality of FiRe. Most functionality relate to the maintenance of membership information and neighbor selection. Several event types and several functions are provided and enable FiRe and services to interact. The following sections will highlight important functions.

5.3.1 Joining a Group

To join a FiRe group, a member must obtain a group certificate (`gCert`), a member certificate (`mCert`), and the corresponding private key (`mPriv`). In addition, the joining member must have member certificates of a sufficiently large number of bootstrap members (`bootNodes`) such that at least one is correct with high probability. A service can then join a group by invoking:

`join(gCert, mCert, mPriv, bootNodes)`

Optionally, trusted bootstrap nodes can be provided in the group certificate, in which case the `bootNodes` argument is not needed. The CA component consists of a set of command line tools to create and sign group and member certificates. FiRe provides library functions to marshal these data structures to and from files.

5.3.2 Events

FiRe produces events in response to changes in the membership. There are three event classes: *data events*, *membership events*, and *neighbor events*. In our current prototype, events are implemented as function callbacks. For this, a function pointer and an argument list is put on an event queue. The event loop responsible for making callbacks is provided by the underlying Twisted networking framework [96]. To distinguish events from function calls, event names are prefixed with `ev`.

Data Events

Data events are associated with the reception of low-level membership events from other members over the Internet. There is one event for each of the three communicated data types: accusations, notes, and certificates. Each data event has a data object pointer `o` and a `valid` flag. If `valid` = `1`, the data object `o` is valid according to the membership rules. Otherwise, `o` is not considered valid and should be discarded. The runtime will not explicitly delete these objects because services might need to do their own garbage collection.

Although FiRe will not accept inbound data that is invalid, changes in the membership might, for instance, cause a previous valid accusation to become invalid. Notes and certificates become invalid when a higher numbered version is received. The data events are:

`evAccusation(o, valid)`: Contains an accusation `o` for the member `m` with `m.identity` = `o.accused`. If `valid` = `1`, the accusation is stored in the `m.accusations` list.

`evNote(o, valid)`: Contains a note `o` for the member `m` with `m.identity` = `o.identity`. The note is stored in `m.note` and will invalidate all accusations in the `m.accusations` list and the previous note in `m.note`.

`evCertificate(o, valid)`: Contains a member certificate `o` for member m with `m.identity` = `o.identity`. Object `o` is stored in `m.certificate`. The previous certificate is discarded.

Membership Events

Membership events are triggered due to reception of data events and, in one case, the passage of time. Membership events indicate changes within the overlay structure in that one of four possible state transitions has occurred: a new member joins, a member permanently leaves, a member crashes, and a

crashed member recovers. In FiRE these four state transitions are represented by two types of events:

`evCrashed(m, reason)`: Indicates that member `m` has transitioned from a live state to a crashed state. The `reason` argument indicates why this occurs. If `reason` = `0`, then `m` has permanently left the membership, which is due to `m.certificate` being revoked or that it has not been renewed before its expiration date. In this case, services can garbage collect data structures and adapt to a permanent membership change. If `reason` = `1` then `m` is considered failed due to it being accused and not having updated its note before the failure timeout 2Δ expired.

`evRecover(m, reason)`: Indicates that member `m` has transitioned from a crashed state to a live state. If `reason` = `0` then `m` is a newly joined member, which signifies a definite changes in the overlay structure. If `reason` = `1` then member `m` was previously considered crashed, but has updated its note such that all previous accusations are invalidated (i.e., `m` has rebutted the accusations). If `reason` = `2` then `m` was previously considered crashed, but all previous accusations have been invalidated through changes in the membership.

Neighbor Events

A neighbor event suggests that an overlay link should be established or broken. They are triggered as a consequence of a successor or predecessor crashing or recovering.

`evConnect(m, role)`: These events indicate that `m` is to be considered as a neighbor. If `role` = `0` then `m` is a successor. If `role` = `1` then `m` is a predecessor. If `role` = `2` then `m` is both a successor and a predecessor.

In practice, the `role` argument is used when symmetry needs to be broken. For instance, after receiving an `evConnect` event, the FiRE gossip component decides based on the `role` argument if it should establish a TCP connection to `m`, or if it should expect and accept inbound connections from `m`. If `role` = `2`, both inbound and outbound connections could be established. In this case, the gossip component breaks symmetry by comparing member identities.

`evDisconnect(m, role)`: Indicates that `m` should no longer be considered as a neighbor. The `role` argument has the same meaning as in `evConnect` events.

5.3.3 Functions

In addition to emitting events, FiRe exports several functions, including the following.

`register(o, k, mask)`: This function registers an object `o` as a sink for membership events. Which types of events `o` will receive depends on the provided `mask` argument. The `mask` contains three bits. Each bit corresponds to one of the three event types that was defined in Section 5.3.2. If a bit is enabled, then FiRe will deliver those types of events. The bits are as follows:

Bit	Symbol	Event Type
0	`evDATA`	Data events
1	`evMEMBER`	Membership Events
2	`evNEIGHBOR`	Neighbor events

The symbols in this table are constants that can be or'ed together to create the appropriate mask.

The value `k` states how many neighbors are required by `o`. The following values for `k` are calculated on startup based on the set configuration options:

Symbol	Meaning
K_{one}	At least one correct neighbor
K_{maj}	A majority of correct neighbors
K_{con}	Members form a connected sub-mesh
K_{all}	Indicates that the members should be fully connected

Note that if neighbor events are disabled (i.e., bit 2 of `mask` is disabled) then the `k` argument is ignored. Figure 5.5 illustrates a service that registers to receive neighbor events.

`deregister(o)`: Object `o` will no longer receive membership events from FiRe. An unregistered object is allowed to make library function calls.

`suspect(m, ring)`: Suspect member `m` of being failed on the specified `ring`. The function fails if `m` can not be accused in accordance to the membership rules. Otherwise, the function results in the generation of an accusation for the current note of `m`. If $\texttt{ring} = -1$, the first possible ring is used. If $\texttt{ring} > K_{maj}$, an error is produced.

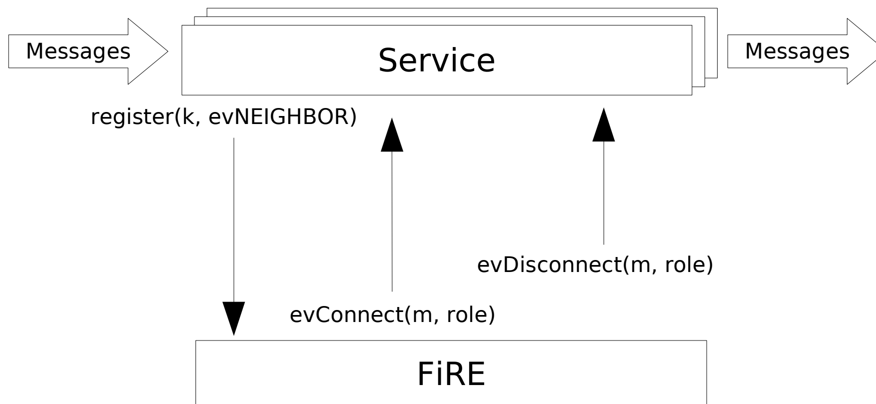


Figure 5.5: A FiRE service that registers to receive neighbor events

`members`(`all`): Returns a list of member objects. If `all` = 0, the returned list will contain all and only live members. If `all` = 1, the list will contain all members, both crashed and live. A service can either pull this function to maintain its view of the overlay, or register to receive membership updates through the event interface.

`getMemberByIdentity`(`id`): This function returns the member object m whose $m.identity$ = `id`. The function fails if no such member exist.

5.4 Internal Issues

In the previous section we described FiRE from an external perspective. In this section we will highlight important internal details. Key internal functions are those that relate to the maintenance of the membership rings, failure detection, and gossip.

5.4.1 Membership Rings

FiRE maintains k membership rings, where k is the largest number of rings requested by a subcomponent through the `register`$(k, \ldots)$ function.

Initially, $k = \mathtt{K_{maj}}$ rings are created, one for each of the required number of monitoring rings. The rings are numbered from 1 to k. Bit i of a note's enabled attribute controls monitoring on ring $i + 1$. If a service specifies a k smaller than the current number of rings, the first k existing rings will be

```
succ(m, ring): // Return successor of m in ring.
  idx = ring.index(m) // index of m in the ring array
  return ring[ (idx + 1) mod ring.length ]

pred(m, ring): // Return predecessor of m in ring.
  idx = ring.index(m)
  return ring[(idx - 1) mod ring.length]
```

Figure 5.6: Pseudo-code for ring operations

used. If k is larger, then new rings are created. The creation of a ring does not affect services that previously have registered.

To construct a ring structure from an unordered set of members, we keep the member objects in a sorted array r. The edges are implicitly defined from array element $r[i]$ to $r[i+1]$, except if $r[i]$ is the last element, in which case there is an edge from element $r[i]$ to $r[0]$. Successor and predecessor operations are implemented using array index subtraction and addition modulus the number of members, as illustrated in Figure 5.6. Rank operations are implemented using recursive successor operations.

Different rings are then implemented by sorting the member objects in different orders. For this, we associate with each ring an unique numerical identity $r.id$ in the range from 1 to k. Next, we construct a ring specific pseudo-random sort key for each member `m` using the SHA-1 output of `m.identity` concatenated with the ring identifier `r.id`. Because SHA-1 is CPU intensive, computed ring identities are kept as an attribute of member objects. Each ring array is kept sorted according to each sort key.

It is possible for a member to specify a number of rings that depends on the size of its view. Care should be taken to deal with Byzantine members specifying an *enabled* map in their note with a very large number of rings in order to try to consume all memory of correct members. Our current implementation of FiRe assumes a static number of monitoring rings, implying that a maximum membership size should be anticipated and enforced.

5.4.2 Gossip

The FiRe gossip component is responsible for the distribution of low-level data events. For this, the gossip component maintains three data sets: a set of valid accusations, a set of valid notes, and a set of valid certificates. On

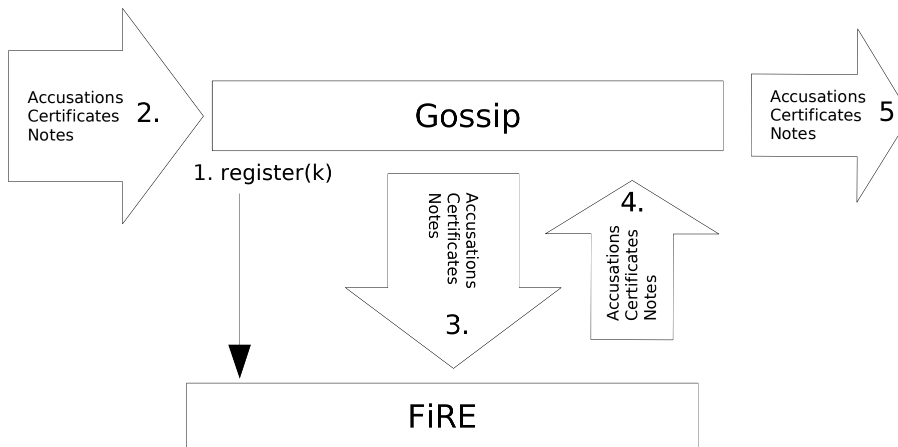


Figure 5.7: Gossip of accusations, notes, and certificates

startup, the gossip component `g` registers itself with FiRE by calling

$$\texttt{register}(\texttt{g},\ \texttt{K}_{\texttt{conn}}\ , \texttt{evData} \mid \texttt{evNeighbor})$$

After that, `g` will receive both data events end neighbor events from FiRE as shown in Figure 5.7. The data events are used to maintain the three data sets. By specifying $\texttt{k} = \texttt{K}_{\texttt{conn}}$, FiRE provides to the gossip component a sufficient number of randomly chosen neighbors such that the set of correct members will be connected with high probability, as described in Section 4.8.1.

The gossip component will listen for inbound TCP connections on the IP address specified in `mCert.address`, where `mCert` is the member certificate specified in the `join` function call. Upon receiving an $\texttt{evConnect}(\texttt{m}, \texttt{role} = \texttt{0})$ event, the gossip component will establish and maintain a TCP connection to `m`. Since the underlying membership protocol is responsible for failure detection, a broken TCP connection does not signify a failure. Broken or timed-out TCP connections are therefore reestablished such that interrupted data exchange can be restarted. Exponential back-off is used to avoid adding traffic to a congested network.

Although network traffic from multiple services can be multiplexed into a single TCP connection, FiRE currently does not implement this feature. There are two reasons for this. First, multiplexing data streams would complicate the implementation and add communication overhead. Second, application messages would block membership messages, thus increasing the required upper bound on dissemination time, Δ. If application messages are small, this would not necessarily be a problem, but we do not want to impose any limitations on message size. Although FiRE could be instrumented to

```
on time to ping m on ring r:
    τ = log(P_mistake)/ log(1 − 1/m.avgLoss); // calculate threshold
    if m.nPing > max(τ, τ_min)
        m.accusations.add( new Accusation(m.note, r, self.id) );
    else
        send(m, new Ping(self.id));
        m.nPing + +;

on receive Pong(m):
    m.avgLoss = α × m.avgLoss + (1 − α) × m.nPing;
    m.nPing = 0;
```

Figure 5.8: Adaptive Pinging Protocol

split large messages into smaller fragments, this would add complexity and overhead. Instead, each TCP-based service must set up a separate connection to each neighbor. Also, FiRE currently does not support multiplexing of inbound connections from a single listening port. Port number must either be agreed upon in advance or be included in member certificates.

Each time a member m_1 gossips with member m_2, they first exchange a collision-resistant hash of their sets of certificates, notes, and accusations. If they differ, a full state reconciliation is done using the algorithm described by Minsky et al. [102, 103]. Our current implementation of this algorithm is based on the reference implementation made by Agarwal and Trachtenberg [4]. The algorithm is optimal with regards to communication complexity but can be CPU intensive.

5.4.3 Adaptive Pinging Protocol

In the same manner as the gossip component, the failure detection component is managed by FiRE by having it call the `register` function upon initialization. It registers itself with $k = \mathtt{K_{maj}}$ such that at most t out of $k = 2t + 1$ neighbors are Byzantine as specified in Section 4.3.3.

In Figure 5.8, we present a simple, but effective probing protocol based on an adaptive failure detection mechanism as described in Section 4.7. Because packet-loss must be estimated, our protocol uses UDP for unreliable message passing.

In essence, m_1 probes m_2 at intervals of length $\mathtt{T_{ping}}$. If a probe has not succeeded before the next probe is scheduled, then that probe is considered failed. If more than τ consecutive probes fail, then m_1 considers m_2

as crashed. In this case, m_1 creates and gossips an accusation for m_2. The timeout value τ is adjusted to the estimated packet loss rate, P_{loss}, between m_1 and m_2, and the configuration target probability of making pinging mistakes. $P_{mistake}$. After creating an accusation for m_2, m_1 does not immediately consider m_2 as crashed. It waits until the timeout period 2Δ expires.

Chapter 6

Evaluation

To evaluate both the *Fireflies* protocol and its implementation in FiRe, we constructed a basic FiRe service, which only task is to initialize and run the membership management protocol. We refer to each running instance of this service as a FiRe *agent*. We ran experiments using FiRe agents in both simulated environments and on PlanetLab. This chapter describes important experimental findings. We describe in detail our experimental environments and our setup. We also highlight important problems that were encountered and describe possible sources of error.

6.1 Simulations

To produce repeatable experiments in a controlled environment, we constructed a simulated network environment in which FiRe agents could run. Simulated FiRe agents shares most of the code-base with their networked variants, but bypasses the TCP stack, the UDP stack, and the marshaling routines for efficiency reasons. We also avoid copying accusation, notes, and certificates structures by passing memory location pointers. Having members share these data structures is unproblematic as they are made immutable using digital signatures.

By using the same code as in the networked variant of FiRe, we get a detailed picture of how the protocol will behave. It also simplifies debugging as errors can be reproduced in a controlled simulated environment. Unfortunately, the level of detail which we simulate impacts the scalability of the simulator. On a 3 Gigahertz (GHz) Pentium 4, we were able to simulate 256 members with simulation time running close to real-time. Observed aggregate memory usage was never above 100 MB. The scalability limits of our simulator does *not* not indicate scalability limits of the *Fireflies* proto-

col since we expect the protocol to be network bound when running in the wide-area Internet.

With several hours of simulation time and several repetitions, a single data point took up to 24 hours to produce. To speed up data generation, we therefore ran simulations concurrently on a local cluster consisting of 36 3.2 GHz Intel Prescott 64 machines. As our simulator is deterministic, it is not affected by the underlying software and hardware platform. In particular, local timing issues and concurrent load are not factors in the output. Hence, running simulations concurrently on multiple machines does not introduce a bias.

In all experiments presented in this chapter, we used 25 membership rings, which, according to our calculations in Section 4.3.3, enables FiRe to tolerate 20% Byzantine members within a group of up to 1000 members. The ping interval, T_{ping}, was set to 30 seconds. For gossip we used 8 membership rings such that each member was assigned 16 neighbors. According to the calculations in Section 4.8.1, this ensures that FiRe can deliver membership events to all correct members with probability $\phi = 0.9999999$ within a group of 1000 members when 20% of those members are Byzantine. The gossip interval, T_{gossip}, was set to 3.75 such that each member gossiped on average once every minute with each neighbor. The intervals at which a member gossips were randomized in order to prevent synchronized "waves" of gossip. The probabilistic upper bound on the time for gossip to spread, Δ, was set to 2.5 minutes. We set the probability of making a mistaken failure detection in the pinging protocol, $P_{mistake}$, to 0.001 in order to trigger frequent false accusations.

To simulate churn, members would periodically crash and later rejoin. Both the Mean Time To Failure (MTTF) and the Mean Time To Recovery (MTTR) of all members were set to 6 hours. The intervals between crashing and starting were exponentially distributed. Although several works have argued that the exponential distribution does not accurately model churn [105, 142], it is unclear if the proposed alternatives are better suited for our purposes.

The total number of members, N, ranged from 16 to 256. Each experiment was run for seven simulation hours, including an one-hour warm-up period before measurements started. The warm-up period excludes from our measurements the extra load incurred by bootstrapping the system. Each experiment was repeated from nine to twelve times with different initial random seeds.

For each set of experiment, we calculated 95% confidence intervals. With 128 or more members, the calculated confidence intervals were on average within 2% of the measured value. With fewer than 128 members, confidence

intervals were higher. For instance, with 16 members, the average observed confidence interval size was 9% relative to the measured value. The maximum observed confidence interval was 12% in one experiment with 16 members. Larger confidence intervals are expected with smaller membership sizes because there are fewer data points. For clarity, the graphs presented in this section contain confidence intervals only where we consider them significant.

6.1.1 Overhead of Membership Maintenance

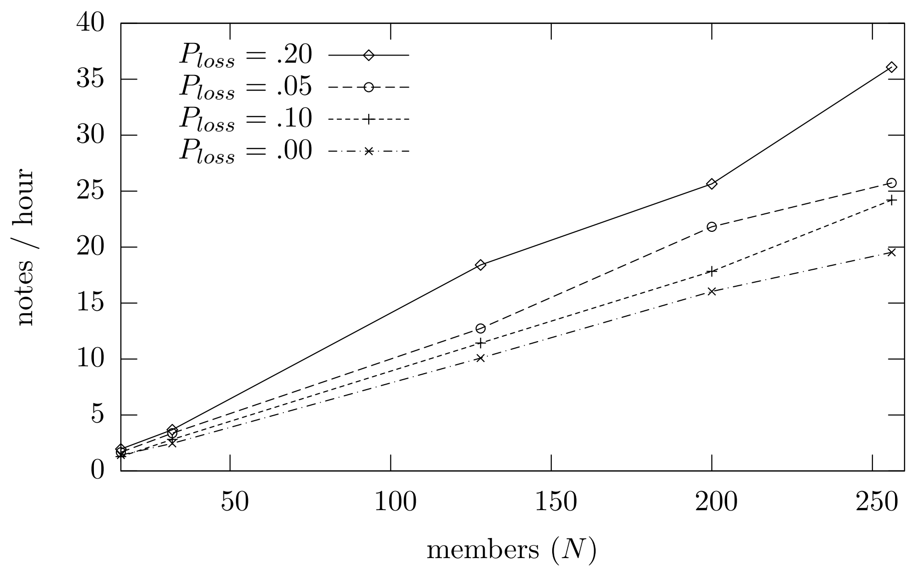
First we look at network overhead due to membership maintenance in absence of Byzantine members. For each member, we record the rate of sent and received notes and accusations. We do not record the rate of sent or received member certificates since we do not simulate adding or removing members.

Figure 6.1a shows the average number of notes sent (created or forwarded) per correct member per hour as a function of the number of members for various P_{loss}, the probability of message loss.

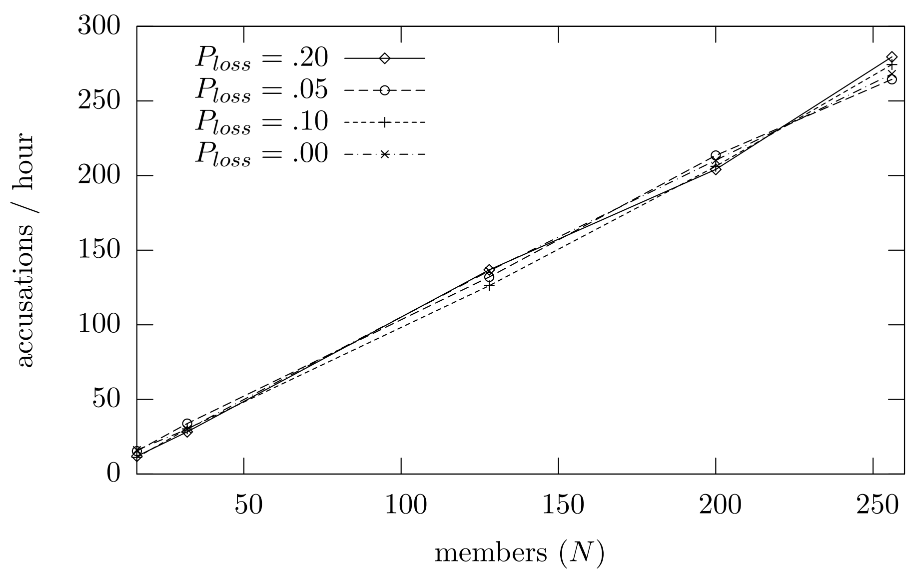
Without packet loss, the expected number of notes is $N/12$, as on average there is one recovery every 12 hours (i.e., MTTF + MTTR). With loss, pinging mistakes are made. With 256 members, packet-loss approximately doubles the number of notes generated as false accusations are rebutted. We see a clear *linear trend* in the rate of sent notes as a function of the number of members. This trend is expected as our simulated rate of churn increases linearly with the number of members

Due to our adaptive pinging protocol the rate of mistakes is almost, but not completely, independent of P_{loss}. In particular, a higher loss rate does not necessarily imply more pinging mistakes. For instance, in Figure 6.1a, for 5% packet loss the rate of notes sent is higher than for 10% packet loss. The differences are due to the rounding error that occurs when calculating the pinging threshold, τ, as explained in Section 4.7.2. Because we compute τ using the ceiling function, our protocol consistently makes fewer mistakes than what would be expected from the configured $P_{mistake}$.

Figure 6.1b shows the average number of accusations sent per correct member per hour as a function of the number of members for various P_{loss}. Because a failed member is accused on multiple rings, these rates are higher than for notes. They do not depend much on the packet-loss rates. This is partially due to our adaptive pinging protocol, but also due to the fact that a pinging mistake results in only one accusation, while a crash results in up to one accusation per enabled ring. In our case, each member has exactly 13 rings enabled at all times. Unlike a valid accusation, a false accusation might also not reach all members if the accused member is quick to issue a rebuttal.



(a) Note rate



(b) Accusation rate

Figure 6.1: Simulated network overhead for varying packet-loss rates

6.1.2 The Effect of Byzantine Members

Next, we introduced Byzantine members into our simulation. We implemented two types of membership attacks:

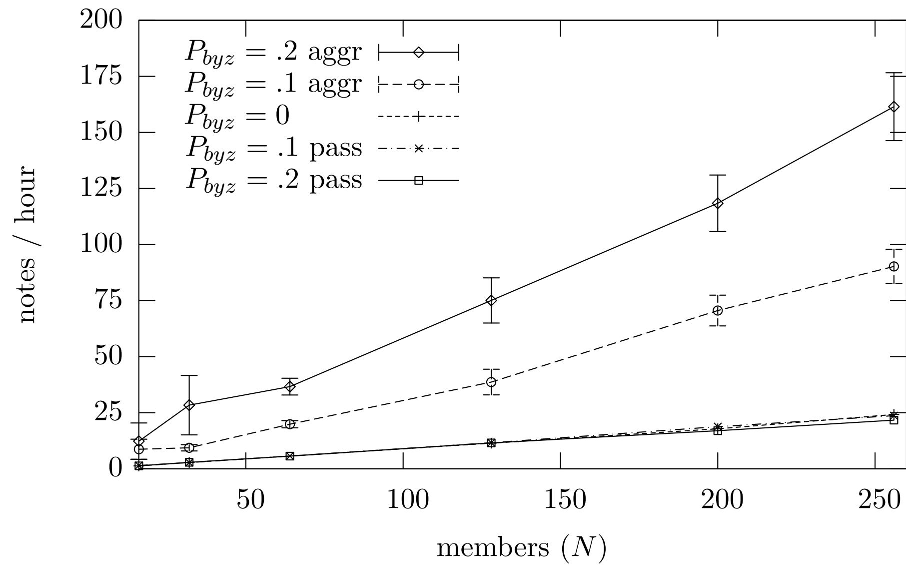
- An *aggressive attack,* where the goal of the attacker is to remove live members from the views of correct members and to induce extra network load. For this, the attacker accuses other members of being failed at any opportunity. The attacker will only create accusations that are valid in accordance to its view of the membership. In particular, an attacker will not accuse a member on a ring that is disabled since such accusations are in clear violation of the membership rules. Correct members will ignore any such invalid accusations and they might forward them to the CA, proving that the attacker is not following the protocol. The attacker will also refrain from forwarding a note if it invalidates any accusations. This in order to prevent correct members from rebutting false accusations made by either the attacker or by other members.

- A *passive attack,* where the goal of the attacker is to keep failed members in the views of correct members. For this, the attacker never accuses members, and does not forward accusations of crashed members.

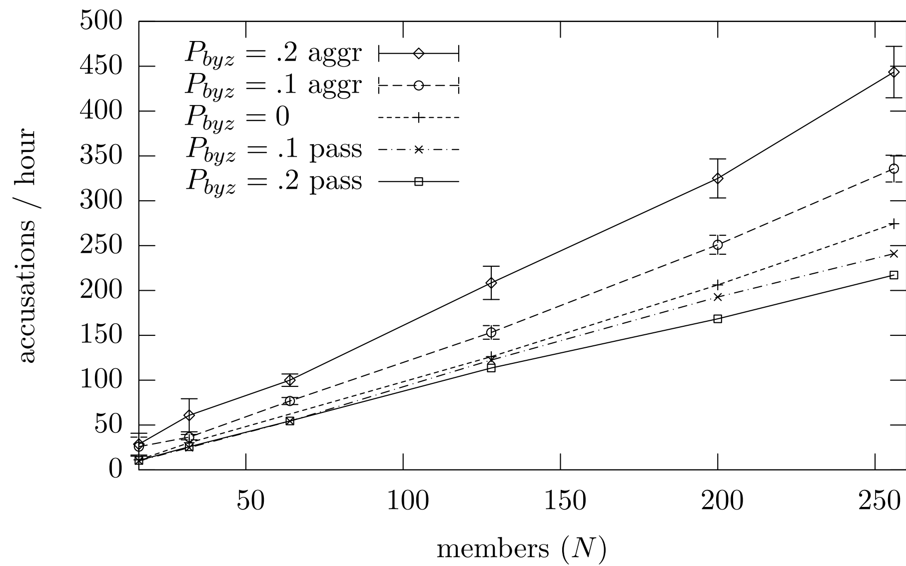
We varied the fraction of attackers, P_{byz}, from 0 to 0.2 for both aggressive and passive attacks. The attackers were chosen randomly from the set of all members. None of the attacks succeeded in our experiments, indicating that *Fireflies* can deal with a large fraction of Byzantine members. The aggressive attacks did, however, have a modest impact on the network load.

Figure 6.2a shows the average number of notes sent by correct members for various P_{byz} and styles of attack. Figure 6.2b shows the corresponding rate of accusations. Surprisingly, the passive attacks decreased the rate of notes and accusations. This observation is, however, reasonable because correct members make pinging mistakes, which results in a false accusation followed by a rebuttal from the falsely accused member. Because passive attackers refrain from making *any* accusations, they will decrease the aggregate rate of false accusations and subsequently decrease the network load.

Aggressive attacks had a moderate effect on traffic. With 20% of the members executing aggressive attacks, the rate of notes increased one order of magnitude compared to when there were no Byzantine members. The rate of accusations doubled. This indicate that an attacker will be able to increase load by creating false accusations, but the rate at which this can be done is bounded due to correct members disabling Byzantine monitors.



(a) Note rate



(b) Accusation rate

Figure 6.2: Simulated network overhead when under attack

6.2 PlanetLab

Our simulations indicate that the network load induces by FiRe increases linearly with the number of members. To gain a clearer understanding on how FiRe and *Fireflies* behave when deployed in the wide-area Internet, we deployed our code on PlanetLab [7, 112, 136]. In essence, PlanetLab is a world-wide collection of over 600 machines at over 275 sites connected to the Internet in 30 countries, including China, Norway, and USA. PlanetLab can be used to test new scalable protocols and to deploy novel distributed services.

We first deployed *Fireflies* on PlanetLab in early February 2005, and found the experience useful to find pragmatic problems and test our solutions. However, as argued in Section 1.4.2, the overheads we measured, some of which are presented below, are specific to PlanetLab only.

6.2.1 Experimental Setup

Our networked variant of the FiRe agent uses TCP for gossip but UDP for pinging as our adaptive pinging protocol needs to determine when messages get lost. Each FiRe agent is instrumented to write a checkpoint to a log every 10 seconds. Each checkpoint contains the current time and measurement data. During an experiment, logs are kept locally on disk at each individual PlanetLab machine. The checkpoint function is robust to delays and bugs in other parts of the FiRe code. A checkpoint contains approximately 100 bytes of data. Writing checkpoints will in most circumstances not noticeably affect the execution of FiRe.

The clocks for recording log times were synchronized using the Network Time Protocol (NTP), which can provide millisecond precision [101]. As we are measuring trends over time-periods of minutes, NTP synchronized clocks provide sufficient precision for our purpose. As a safeguard, all clocks were periodically checked for drift using a local reference clock. Only a few discrepancies were observed. For instance, two machines in France had wrongly set time zones, which were compensated for in our logs. In other cases, machines with irregular clock drift were observed. Such deviating machines were excluded from our experiments.

Each experiment generated about 1 Gigabyte (GB) of log data. After an experiment had ended, all logs were retrieved to and processed on a desktop computer located at the University of Tromsø. In some cases, we were unable to log into some machines after an experiment had ended. Such experiments were discarded because we could not retrieve the logs on those machines.

In some cases, FiRe agents would continue to run on unaccessible ma-

chines. Several experiments were discarded because such zombie agents would integrate themselves into the new membership, thus interfering with our experimental setup. Although changing the group certificate between experiments prevents zombie agents from participating in the new membership, they still interfere by establish TCP connections and sending handshaking messages to members of the new group. We addressed this by changing the TCP and UDP listening ports whenever we observed zombie agents.

6.2.2 Measurement Study

We will in the following describe the results of one of our PlanetLab experiments. The experiment started on February 24, 2006, and ended February 26, 2006.[1] The purpose of this experiment was to measure the behavior of FiRe under both low and high churn load and when, at the same time, under attack by Byzantine members. Configuration options were set the same as in the simulations above, except that $P_{mistake}$ was set to a more sensible 10^{-5}.

Our experiment started with FiRe agents running on 280 PlanetLab machines. During the experiment about 10 PlanetLab machines became unresponsive and fell out of the experiment. At approximately 20:11 on February 24, 2006, we terminated 80 of the agents, chosen randomly. At about 08:02 on February 25, 2006, those agents were restarted.

If an agent has not written a checkpoint to its log during a 4 minute period, it is considered crashed in that period. The number of live agents per time period is shown in Figure 6.3. As expected, we observe a large drop in the number of members followed by an equally large increase corresponding to when we terminated and when we restarted the 80 members.

Terminating the agents involved a script that logged into each individual PlanetLab machine and issued an Unix kill signal. Hence, agents would crash abruptly and without prior warning. The low-level signalling protocols in FiRe are instrumented to tolerate such failures. Starting agents involved a similar script. The scripts ran from our desktop computer located at the University of Tromsø. Running each script on all 80 agents took several minutes.

To measure the impact of Byzantine members, 10% of the FiRe agents were configured to mount aggressive attacks, creating accusations at any opportunity. Another 10% were configured to mount a passive attack, not accusing and not forwarding accusations for failed members. Byzantine members were chosen randomly from the set of all members.

In an early version of FiRe, members chosen to execute aggressive attacks

[1]Dates are in Eastern Standard Time (EST) using 24-hour clock notation.

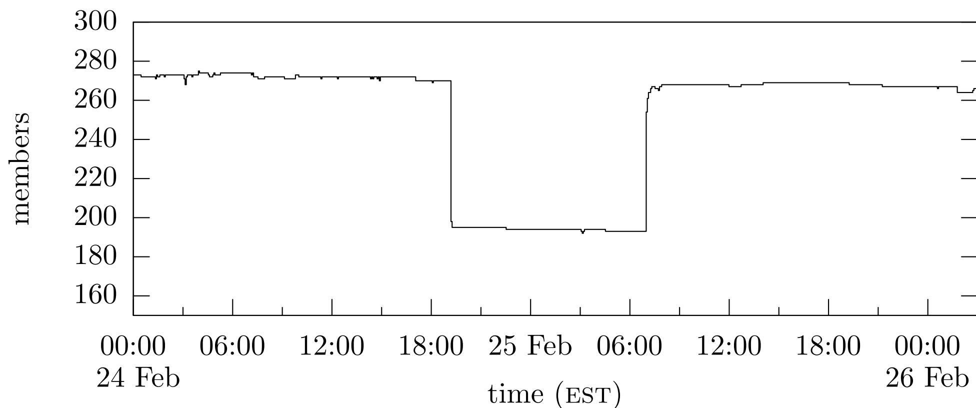


Figure 6.3: Live members

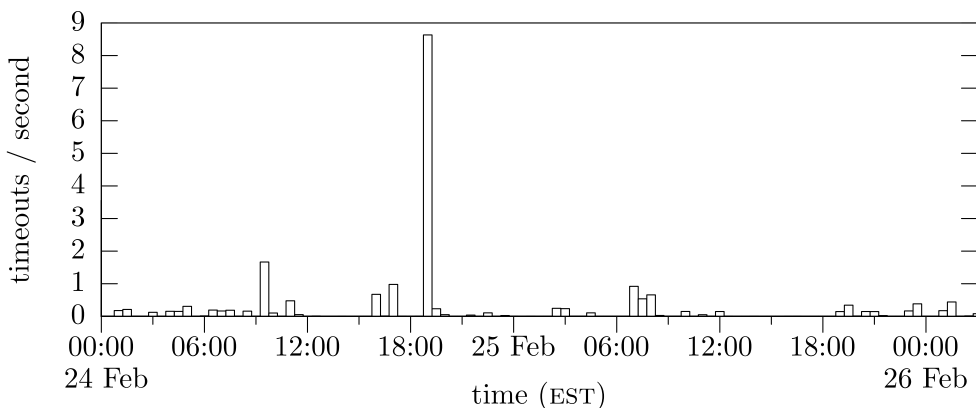


Figure 6.4: Rate of timeouts

would start attacking immediately after startup, accusing other members at any opportunity. We observed several instances where such behavior resulted in Byzantine members partitioning themselves out of the membership without causing much harm to other members. To have more effective attacks, we therefore modified our code such that Byzantine members do not start attacking until they had integrated themselves fully into the membership.

Figure 6.4 shows the aggregate rate of `evCrashed` events (i.e., failure timeouts as defined in Section 5.3.2) emitted as a consequence of valid accusations not being rebutted. To show trends over time, we average data over 30 minute time intervals. A clear peak of approximately 8.8 failures per second can be distinguished when members are terminated. The peak lasts for 30 minutes. Hence, over that period a total of $15,840$ `evCrashed`

events are registered. This corresponds well with the 16,000 such events that were expected as a consequence of the remaining 200 members emitting an `evCrashed` event for each of the 80 terminated members. The missing events are registered either before or after the peak period. This indicates the remaining agents adopt to the outage within 30 minutes.

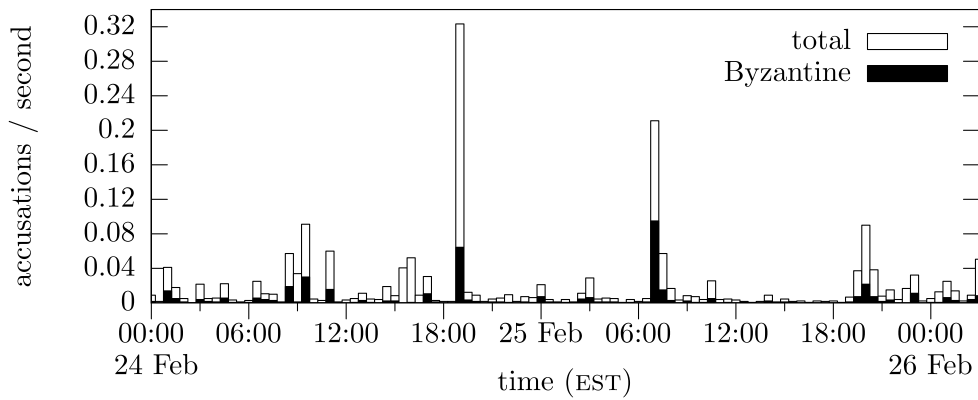
Membership Events

To analyze how the membership changes during our experiment, members log a wide range of membership events. Figure 6.5a plots the observed aggregate rate of accusations created per second, divided into total and false accusations from Byzantine members. Two peaks can be clearly distinguished: when the agents are terminated, and when they are restarted. The first peak is obvious: the terminated agents are accused by the remaining correct members. The second peak was unexpected. It turned out to be caused by several rejoining agents becoming unresponsive due to some heavily loaded PlanetLab machines' inability to accommodate the extra CPU and network overhead incurred when re-integrating the recovering agents into the membership. Accidental accusations from correct members gives aggressive Byzantine members opportunities to issue new false accusations, adding to the temporary flurry of communication.

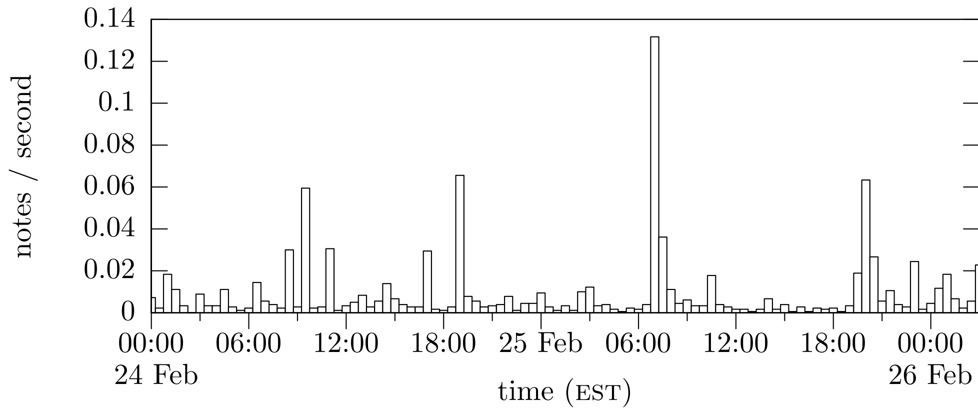
Figure 6.5b shows the aggregated rate of created notes (rebuttals). A large peak at 08:02 February 25, 2006, was expected as the recovering agents need to generate and disseminate new notes when re-integrating themselves into the membership. We see a clear correlation with the rate of Byzantine accusations shown in Figure 6.5a. For instance, the size of the 08:02 peak is explained as follows. The 80 crashed agents whom, upon restart, all generate a note within a period of 30 minutes, would incur an average of 0.04 notes per second within that period of time. In addition, Figure 6.5a shows that Byzantine members are generating approximately 0.09 false accusations per second, which are all rebutted. In total, the expected rate of notes in that period of time should be $0.04 + 0.09 = 0.13$, which corresponds well with what we observed in Figure 6.5b.

Uncontrollable Factors

All our measurements indicate that there is a fair bit of membership churn not under our control. For instance, in Figure 6.5a, we see this as a constant rate of new accusations being created. Because the liveness of a member is only determined by its ability to write a checkpoint to file, the churn, seen as wiggles in Figure 6.3, is not due to network outages or delays in the network.

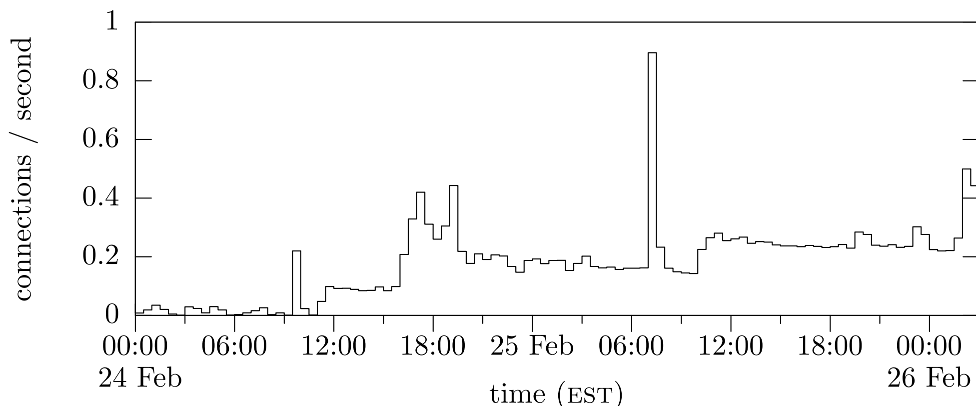


(a) Accusations

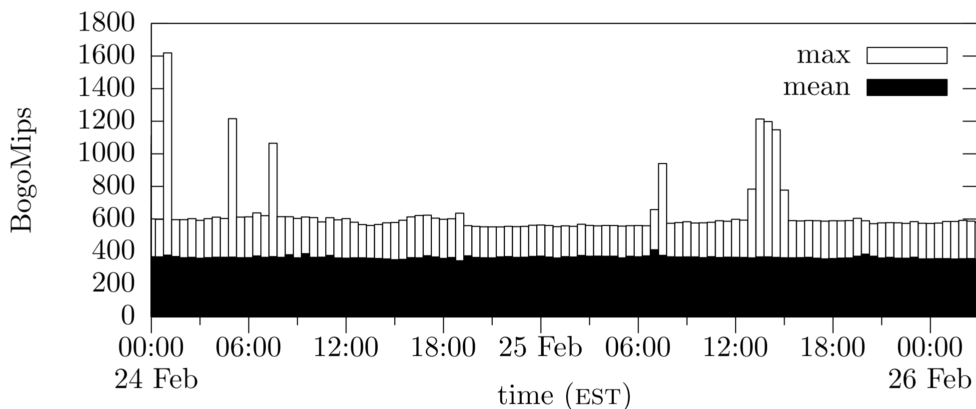


(b) Notes

Figure 6.5: Aggregate rate of membership events on PlanetLab



(a) TCP Connection



(b) CPU load

Figure 6.6: Observed churn on PlanetLab

In practice, we have observed that the majority of PlanetLab nodes tend to be fairly well-connected, although some of the nodes are heavily loaded, to the point of making some of these nodes effectively unreachable.

We have observed nodes that are only partially reachable, either due to configuration problems or due to heavy packet loss. For example, some nodes could not send or receive UDP messages, although TCP would work. This has two consequences for *Fireflies*. First, a node that can not receive UDP packets will accuse its successors, even if they are correct. This is not a problem, as these successors will use their *enabled* bitmaps to disable the corresponding rings. Second, such a node will be accused by its predecessors. The accusations are effectively rebutted, and this accused member is not removed from the views as long as it is able to gossip new notes (using TCP). Unfortunately, the member can not disable all rings, which would have

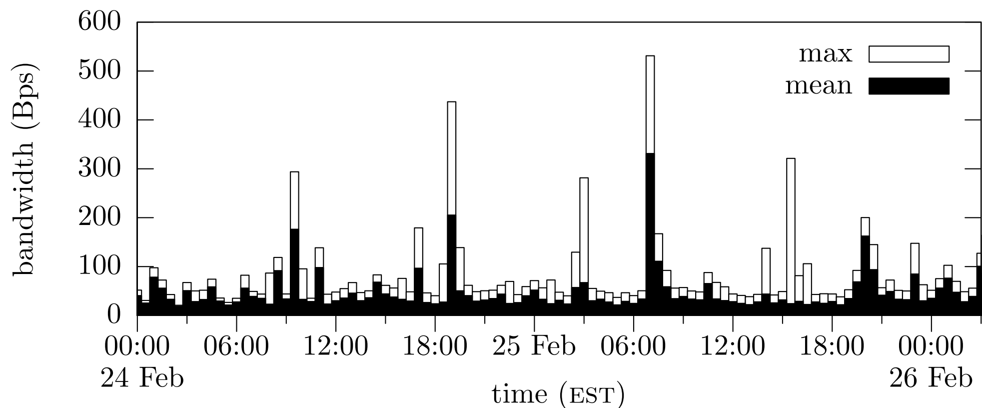
its own problems, leading to the observed continuous background gossip of accusations and notes. Besides a communication overhead on the network, these superfluous messages increase the load on machines.

We also observed nodes that had problems communicating through TCP. Members will time-out and try to restore a TCP connection to a neighbor as long as the membership protocol considers that member live. Figure 6.6a shows the aggregate rate of opened TCP connections per second. As expected, there is a clear peak in the graph when the terminated members rejoin. We also see a constant rate of new connections being opened, which indicate that some members had connectivity problems.

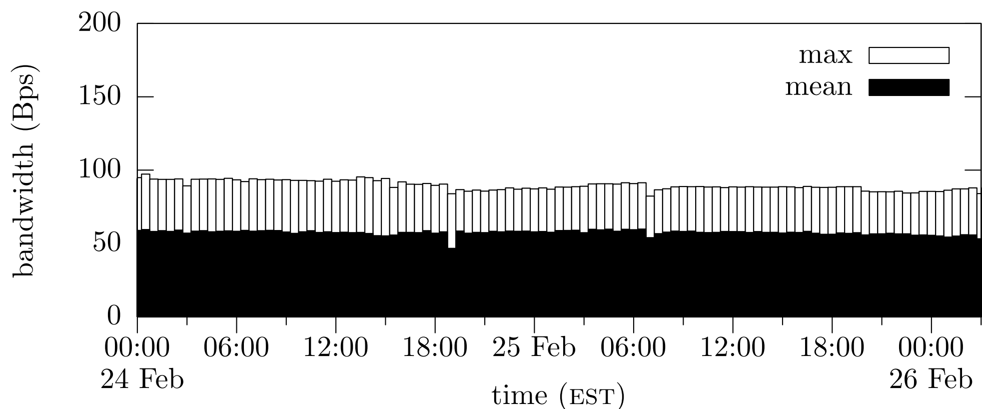
A limitation of the *Fireflies* protocol is that the correct nodes must form a connected gossip graph. In particular, *Fireflies* does not handle network partitions. Some network partitions have been observed in our PlanetLab deployment when an individual member became disconnected from the rest of the network. Such a member is generally not able to accuse every other member, and the partition prevents the member from receiving accusations. It is then stuck with a view that includes members that it can not reach until the partition is resolved. Occasionally, however, there are "successful" partitions. For example, we have observed two members in China form their own membership.

Because the set reconciliation algorithm that we use in our gossip protocol is CPU intensive (see Section 5.4.2), we suspected that agents might be CPU bound. To check our CPU utilization, agents recorded their effective CPU user time, as reported by the underlying operating system. These values were then multiplied with the machines' *BogoMips* rating [49], which is a coarse-grained estimate of the Million Instructions Per Second (MIPS) that a processor can execute. The resulting value is an estimate of the MIPS used by each agent.

Figure 6.6b plots both the observed mean and max BogoMips values. As can be seen, the mean is fairly stable at around 300 BogoMips. The highest recorded value lies around 1600 BogoMips. The average number of BogoMips available on the machines that we used was 4270. The minimum was 1145. With up to 25 concurrently running experiments (also known as slices) and a median CPU utilization of about 50% [111], it is likely that some agents did not operate correctly due to being CPU bound. This observation indicates that the use of a CPU intensive set reconciliation algorithm might be inappropriate in a heavily loaded PlanetLab environment. In environments with more CPU capacity, this might not be an issue. For instance, we have been running as many as 20 FiRE agents on a single 3 GHz Intel Pentium 4 without them being CPU bound. Such a processor provides approximately 6000 BogoMips.



(a) Outbound TCP traffic



(b) Outbound UDP traffic

Figure 6.7: Network performance on PlanetLab

6.2.3 Network Performance

Next we examine the load that our system incurs on the network. We instrumented our code to record the number of bytes sent and received trough both TCP and UDP, and included those numbers in the logs. The instrumentation was done as far down in the protocol stack as possible such that all signaling and protocol overhead were captured. We do not include overhead from TCP and IP headers. Figure 6.7a shows the mean and maximum network per member due to gossiping of certificates, notes, and accusations. The bandwidth follows the rate of accusations, but mostly remain below 50 bytes per second (Bps) per member. The various peaks are caused by the issues described above. The largest observed peaks are when the 80 agents are terminated and restarted, in which case, members sent as much as 520 Bps.

Figure 6.7b shows the mean and maximum number of bytes sent using UDP (i.e., pinging traffic). The mean rate is fairly stable at approximately 50 Bps per member with a maximum of 100 Bps. The stable rate indicates that members have a constant number of neighbors to monitor. There is a small drop in the average when we terminate the members. This is expected as monitoring assignments change.

Chapter 7

Case Study: Disseminating Software Updates

To show that FIRE can be used to construct useful overlay networks, this chapter will describe the design, implementation, and evaluation of FirePatch, a secure software dissemination mechanism. FirePatch combines encryption, replication, and sandboxing with the intrusion-tolerant membership management mechanism provided by FIRE.

7.1 Background and Related Work

Automatic software updates for bug fixes are essential for Internet applications. It is particularly important when a software update fixes a security hole. Software vendors, for fear of liability, release patches for security holes as soon as possible. They do so without publicizing what the bug is, for fear that hackers will exploit the vulnerability before end-users have an opportunity to install the patch.

In practice the time between when a patch is released to the time that it is installed is long and typically measured in days [9, 62]. A counterintuitive observation is that a long patching cycle is worse than no patching cycle at all. This paradox stems from the fact that a security patch can be reverse-engineered to reveal the vulnerable code. In other words, if the software vendor can not provide the mechanism to distribute *and* install a patch quickly, the end user might be better of if the patch is not released at all.

Even if users are notified about a vulnerability and are able to download a patch in time, installing a patch is an inconvenience and might lead to downtime of critical services. Patches might also contain bugs that break system configuration or introduce new vulnerabilities. It has even been sug-

gested that patch installation should be delayed until the risk of penetration is greater than the risk of installing a broken patch [20].

Fortunately, protection against security vulnerabilities can be done in the network layer by installing stateful packet filters like Shields [150], Self-Certifying Alerts (SCAs) [37], or vulnerability-specific predicates [82] that inspect and modify incoming packets. Such patches do not interrupt the execution of applications and are a viable intermediate solution until the user is able to install a permanent fix to the software. Also, automatic patching infrastructures have emerged that greatly reduce the time software is left vulnerable. For instance, a recent study on the Microsoft Windows Update mechanism [62] shows that the automation of notification, downloading, and installation of patches ensures that as much as 80% of the end-clients are updated within one day of patch release.

This still gives a malicious agent ample time to construct and execute an attack. For instance, by examining the binary difference between a vulnerable version of Microsoft's Secure Socket Layer (SSL) library and a corresponding patch, Flake [57] constructed a program that reliably exploited this vulnerability within 10 hours. Marketplaces for buying and selling exploits already exist [141]. It is therefore imperative that software vendors disseminate patches with low end-to-end latency. Such a patch dissemination service must be resilient to DoS attacks and intrusions as hackers might target the service to increase their opportunity to exploit the vulnerabilities exposed by the patches.

A study done on several software vulnerabilities appearing in the last half of the 1990's [9] found that almost all intrusions can be attributed to vulnerabilities known by both the software vendor and by the general public and to which patches existed. The study found that vulnerable software remained unpatched for months or even years. The primary reason for such long patching cycles was, the authors claim, that the studied software was not enrolled with an automatic updating service. Instead, end-users were required to discover the existence of both vulnerabilities and patches on their own by browsing the vendors web-sites, visiting bulletin-boards, etc.

With approximately 300 million clients, Microsoft Windows Update is currently the world's largest software update service [62]. The service consists of a (presumably large) pool of servers that clients periodically pull for updates. Other commercial patch management products like ScriptLogic's Patch Authority Plus[1] and PatchLink Update[2] enable centralized management of patch deployment on the Windows platform. However, it is unclear

[1] http://www.scriptlogic.com/products/patchauthorityplus/

[2] http://www.patchlink.com/

how any of these systems protect themselves from intrusion and if they address the possibility that hackers reverse-engineer patches into exploits.

Open-source communities, like the Debian GNU/Linux Project[3], organize their software update services similarly to Windows Update as a pool of servers that clients periodically pull for updates. Clients can freely choose which server to pull. The servers are organized into a hierarchy with children periodically querying their parent for updates. As these communities rely on donated third party hosting capacity, an attacker can easily intrude into the server pool.

The ratio of how often a patch is released and how quickly it must be received by clients implies substantial overhead for pull-based retrieval mechanisms like those used in the above systems. Pushing is better suited for this type of messaging, but incurs overhead to maintain an up-to-date list of clients. Peer-to-peer content distribution systems, like SplitStream [24], Bullet [90], and Chainsaw [109] approach this by spreading both maintenance and forwarding load to all clients. Although the elimination of dissemination trees in Chainsaw makes it more robust to certain failures than SplitStream and Bullet, these systems do not tolerate Byzantine failures.

A promising approach to detecting vulnerabilities in existing software is to use machine clusters that emulates a large number of independent hosts in order to attract attacks. Such "honeyfarms" have been shown to be able to emulate the execution of real Internet hosts in a scalable manner [149] and can be used to generate SCAs [37] automatically upon detection of intrusion.

7.2 Architecture and Assumptions

We distinguish three roles: *patchers*, *clients*, and *mirrors*. Patchers are typically software providers that issue patches. For simplicity, we will assume a single patcher, although any number of patchers is supported. Clients are machines that run software distributed by the patcher, mirrors are servers that store patches for clients to download, and notify clients when a new patch is available.

We assume that the patcher is correct and is trusted by all correct clients. In particular, using public key cryptography clients can ascertain the authenticity of patches. In our system, clients are passive participants, and in particular do not participate in the dissemination system. Thus we do not have to assume that clients are correct.

In order to increase the patcher's upload capacity and ability to fight attacks, we employ a distributed network of mirror servers. The more mirrors,

[3] `http://www.debian.org/`

the harder it is to mount a DoS attack against the network. However, the easier it is to compromise one or more mirrors. We allow a subset of mirrors to become compromised, but assume that individual compromises are independent of one another, and that the probability that a mirror is compromised is bounded by a certain P_{byz}. However, we do allow compromised mirrors to collude when mounting an attack.

The patcher publishes (and signs) the list of servers that it considers mirrors for its patches. This list contains a version number so the patcher can securely update this list when necessary.

We assume that all communication goes over the Internet, the shortcomings of which are well-known. In order to deal with spoofing attacks, all data from the patcher is cryptographically signed, and we assume that the cryptographic building blocks are correct and the private key is securely kept by the patcher.

7.3 Two-Phase Dissemination

We refer to the time from when a software vulnerability is first made public to when the number of exploitable systems shrinks to insignificance as the Window of Vulnerability (WoV). We have devised a dissemination protocol that, when layered on top of a secure broadcast channel, makes the WoV independent of message size. The net result of such an invariant is that the WoV can be kept fixed and small despite the fact that voluminous data has to be transferred over the wire.

We disseminate patches (or any data) in two phases. In phase one, we distribute an encrypted patch, and in the second phase, we disseminate the small fixed size decryption key. More formally, our general applicable protocol is specified as follows. Let d be a message that a source s wants to disseminate to a set of clients. In the first phase, s generates a symmetrical encryption key K and a unique identifier UID, and broadcasts a ⟨ENVELOPE, UID, K(d)⟩ message, signed by s. Upon receipt and verification of the signature, a client stores this message locally. In the second phase, s broadcasts ⟨KEY, UID, K⟩ to all clients. Upon receipt, clients can decrypt the ENVELOPE message. The UID contains a version number so clients can distinguish newer from older versions of patches.

If t_0 is the time when the first client receives a patch p, if t_1 is the time when the last client receives p, and if Δ_{attack} is the time needed by an attacker to reverse engineer p into a workable exploit, then, as illustrated in Figure 7.1, the WoV opens at time $t_0 + \Delta_{\text{attack}}$ and closes at time t_1. In traditional dissemination the size of the patch determines the length of the

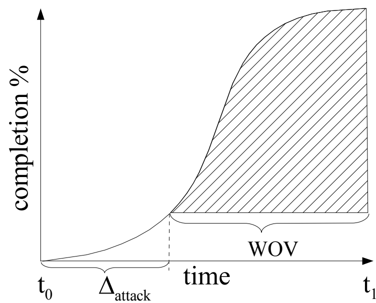


Figure 7.1: Cleartext dissemination

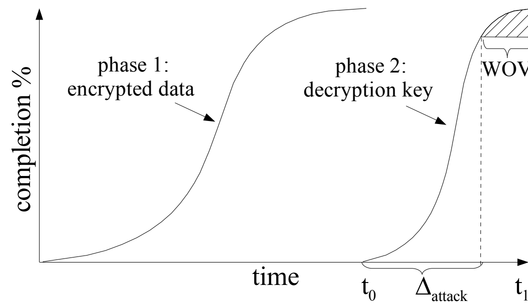


Figure 7.2: Two-Phase dissemination

WoV. The advantage of the two-phase dissemination scheme is, as illustrated in Figure 7.2, that the WoV only depends on phase two. That is, the dissemination of a small fixed size decryption key.

The time between the two phases is a policy decision. One extreme is to do the second phase immediately when the first phase completes. This would require a mechanism by which the patcher detects when all recipients have received the encrypted patch and are ready to install it. However, this is not a viable approach as disconnected clients can delay the completion arbitrarily. More alarmingly, malicious clients can prevent the second phase from happening by never acknowledging receipt. A better scheme is to start phase two some configured time after phase one is initiated. For instance, in the Windows Update system, a 24 hour time period between the phases would allow at least 80% of the clients to receive the encrypted patch [62].

7.4 Secure Dissemination Overlay

As mentioned before, FirePatch employs a network of mirrors to increase the patcher's upload capacity and to fight DoS attacks. The mirrors form a superpeer-like network structure [153] to which clients connect. Thus, the patcher does not broadcast patches and keys directly to the clients, but instead to the collection of mirrors. The mirrors forward this information to all clients that are currently connected to the Internet, and provides it on demand to clients that connect to the Internet at a later time. Each client connects to a minimum number of mirrors such that at least one mirror is correct with high probability.

7.4.1 Mirror Mesh

An attacker might be in control of one or more mirrors. Such Byzantine mirrors are not bound to any overlay protocol and might display arbitrary and malicious behavior. Although cryptographic signatures prevent Byzantine mirrors from modifying or inserting patches, they can still mount a DoS attack by neglecting to forward data.

Our approach to fight such attacks is to ensure that the dissemination overlay contains sufficient link redundancy and link diversity such that, with high probability, there exists at least one path of only correct mirrors from the patcher to each correct mirror and to each correct client. For this, we build on FiRe.

7.4.2 Data Dissemination

FirePatch reliably disseminates patches by an efficient flooding protocol on the neighbor mesh created by *Fireflies*, much like ChainSaw [109]. First, a patch is split into a set of fixed sized blocks that are individually signed by the patcher and disseminated through the mesh. A mirror m_1, upon receiving block b, notifies all of its neighbors by sending them a ⟨BLOCK-NOTIFY, block-id⟩ message, where block-id is the signature of the block. Upon receiving this notification, m_2 can request this block by issuing a ⟨BLOCK-REQUEST, block-id⟩ message to m_1. m_1 then responds with a ⟨BLOCK, block⟩ message containing the requested block. Upon receiving the block, m_2 verifies the signature and stores the block locally. m_2 then notifies all its neighbors, except m_1 that it has received the block.

To enable clients to reassemble the patch from the blocks, the patcher disseminates a signed ⟨PATCH, UID, block-id list⟩ message, where UID is the unique patch identifier. Upon receiving such a message for the first time, a mirror forwards it immediately to all its neighbors except the neighbor from which the message was received. Finally, after some time, the patcher reveals the content of the patch by disseminating a signed ⟨KEY, UID, key⟩ message. These messages are disseminated similarly to the BLOCK-NOTIFY and PATCH messages. Figure 7.3 summarizes the FirePatch dissemination protocol.

To run this protocol, each mirror maintains a TCP connection to each of its neighbors. Mirrors strive to keep all connections busy downloading missing blocks while trying to minimize the number of redundant blocks that they both send and receive. For this we use two techniques. The first technique is to randomize the order in which BLOCK-NOTIFICATION messages are sent. This helps disperse the block randomly upstream from the patcher such that mirrors are able to request different blocks from different neighbors.

```
on receive ⟨BLOCK, block⟩ from m:
  blockid = block.signature
  if  blockid in missingBlocks:
    blockStore.add(blockid, block)
    missingBlocks.remove(blockid)
    for patch in patches:
      if patch.completed(): decrypt_and_install(patch)
    for n in neighbors:
      if n != m: send ⟨BLOCK-NOTIFY, blockid⟩ to n
  schedule_next_request(m)

on receive ⟨BLOCK-NOTIFY, blockid⟩ from m:
  if not blockid in blockStore:
    availableBlocks[m].add(blockid)

on receive ⟨BLOCK-REQUEST, blockid⟩ from m:
  if blockid in blockStore:
      send ⟨BLOCK, blockStore[blockid]⟩ to m

on receive ⟨PATCH, uid, blockList⟩ from m:
   if not uid in patches:
     patches.add(uid, blockList)
     for blockid in blockList: missingBlocks.add(blockid)
     for n in neighbors:
       if n != m: send ⟨PATCH, uid, blockList⟩ to n

on receive ⟨KEY, uid, key⟩ from m:
   patches[uid].setKey(key)
   if patches[uid].completed():
     decrypt_and_install(patches[uid])
   for n in neighbors:
       if n != m: send ⟨KEY, uid, key⟩ to n

proc schedule_next_request(m)
  queue = randomize( missingBlocks ∩ availableBlocks[m])
  next_request = queue[0]
  for blockid in queue:
    if blockid not requested:
      next_request = blockid
      break
  send ⟨BLOCK-REQUEST, next_request⟩ to m
```

Figure 7.3: Pseudo-code for the FirePatch dissemination protocol

This is particularly important during the initial phase of the dissemination. The second technique is to schedule block requests randomly such that a request for the same block is not made to more than one neighbor unless some timeout has expired and the other connections are not busy.

7.4.3 Disconnected Nodes

A problem with the approach so far is that not all clients may be up and connected to the Internet at the time that the patch is being disseminated. When at some later time such a client connects to the Internet, it is vulnerable as hackers have now had ample time to create an exploit and may be lurking on such clients. We thus need a protocol for connecting clients to get the patches thay are missing without being compromised.

Our approach is as follows. When running, clients store the list of all mirrors (disseminated by the patcher just like patches and keys) on disk. When a client connects, a local firewall is initially configured to block all network traffic except certain message formats to and from the mirrors selected at random from the stored list. A client connects to a minimum number of mirrors in order to make it likely that at least one of the mirrors is correct. If all clients connect to all mirrors an unreasonable load might ensue on the mirrors.

First, the client sends a $\langle$RECOVER, $v\rangle$ message to each selected mirror, where v is the version of the latest installed patch at the client. Each mirror responds with notifications of the missing patches as in the protocol described above for connected clients, and the client proceeds to download the necessary patches and keys while all other messages are ignored and dropped. When completed, the client reconfigures its firewall to allow arbitrary communication.

7.5 Evaluation

Our prototype implementation, which is written in Python on top of FiRe, has been evaluated on a local cluster consisting of 36 3.2 GHz Intel Prescott 64 machines with 2 GB of RAM. The machines were connected by a 1 Gigabits per second (Gbps) Ethernet network. We ran 10 mirrors on each machine for a total 360 mirrors. In addition, one dedicated machine was used to run a mirror that acted as the patcher. To limit the effect of network congestion, the outbound bandwidth of each agent was, using a hierarchical token bucket, limited to a rate of 500 Kilobytes per second (kBps) with a max burst size of 1 MB. In addition, each agent divided its total bandwidth equally amongst

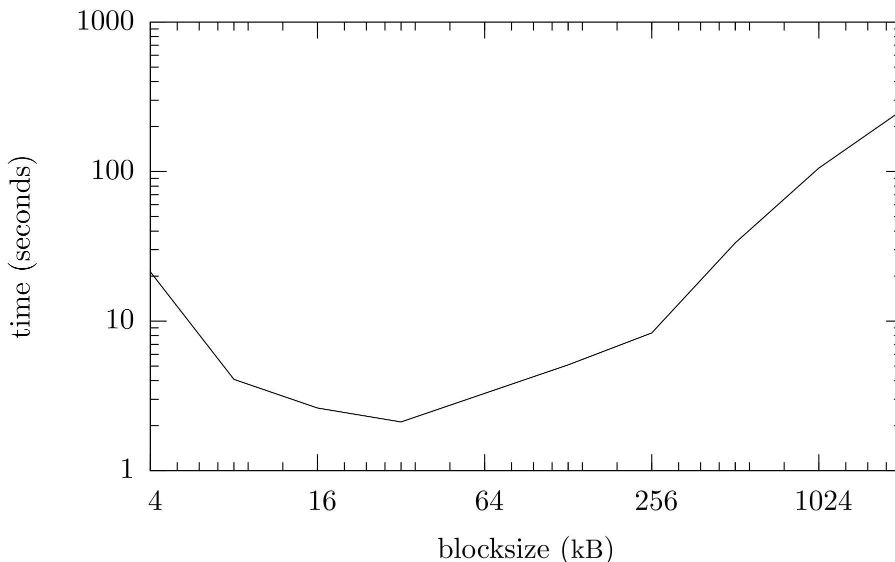


Figure 7.4: Effect of the block size on dissemination

all its active neighbors. In all experiments we used $k = 9$ *Fireflies* rings, resulting in each mirror having 18 neighbors. Hence, bandwidth between two mirrors was approximately 28 kBps.

In our first experiment we measured the effect of the block size on the end-to-end latency. Our experiment consisted of injecting 2 MB patches with block sizes varying between 4 kB and 2 MB. We used a 240 second delay between consecutive patches to prevent interference. A 20 B decryption key was released after a fixed delay of 180 seconds after each patch. To achieve acceptable 95% confidence intervals, we repeated each experiment 20 times.

Figure 7.4 shows the resulting average total dissemination time[4]. As can be seen from the figure, the block size has a significant impact on the end-to-end latency. As expected, the messaging overhead increases with the number of blocks. Also, as the block size increases, the efficiency of our randomized block selection algorithm decreases, producing more duplicate messages and hence a longer dissemination time. We observe that in our set-up the optimal block size is between 16 kB and 64 kB.

Next we tested FirePatch's resilience to attacks from an increasing fraction of Byzantine mirrors in both phase-one and in phase-two of our dissemination protocol. We fixed the block size at 32 kB and repeated the previous experiment with the fractions of Byzantine mirrors varying between 0% and 20%. Each Byzantine mirror was configured to execute omission attacks by

[4]The measured 95% confidence intervals were small and are left out for clarity.

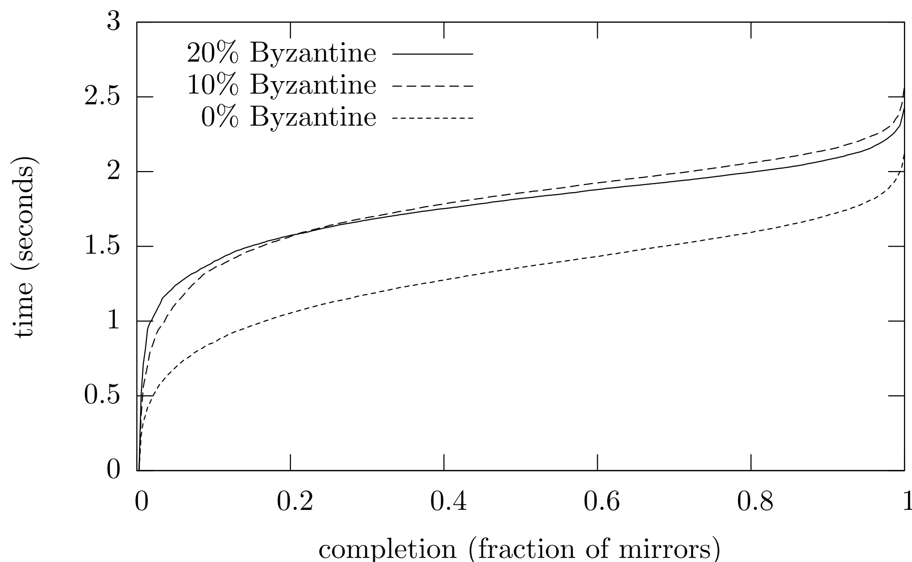


Figure 7.5: Time to complete phase-one

notifying block arrivals but not responding to block requests from neighbors. Byzantine mirrors were chosen randomly from the list of all mirrors. In all our experiments Byzantine mirrors were not able to prevent correct mirrors from completing either phase-one or phase-two.

Figure 7.5 shows the resulting average time for an increasing fraction of the mirrors to complete phase-one of our protocol. As expected, the graph displays a clear gossip-like behavior by starting slow, speeding up, then ending slow. When under attack by 20% of the mirrors, we observed a delay of less than 1 second compared to when all mirrors were correct. This indicates that FirePatch is highly resilient to omission attacks. Note that for larger systems we expect the dissemination time to grow logarithmically in the number of mirrors because the diameter of the *Fireflies* mesh grows logarithmically.

Figure 7.6 shows a similar graph of the completion of phase-two. As expected, the dissemination of the smaller decryption key in phase-two is significantly faster than for the larger sized patch in phase-one. Also, omission attacks had little impact. Figure 7.7 shows the reduction of the WoV size due to our two-phase dissemination protocol when the patch size varies between 128 kB and 4 MB.

Next we compare our phase-one dissemination protocol with naïve push and pull mechanisms. For the push mechanism we modified our code such that mirrors transmitted the blocks instead of block notifications. To implement a pull mechanism we modified our block request scheduler such that it

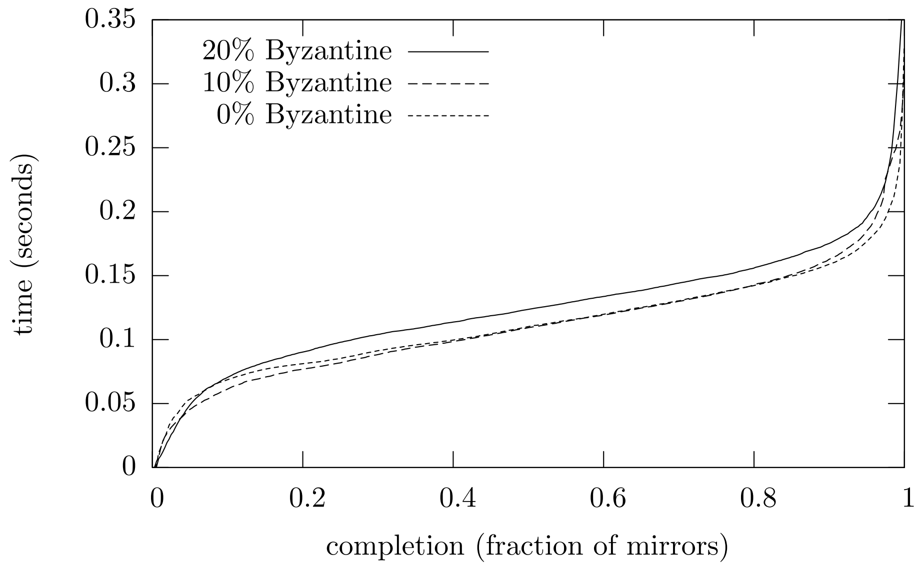


Figure 7.6: Time to complete phase-two

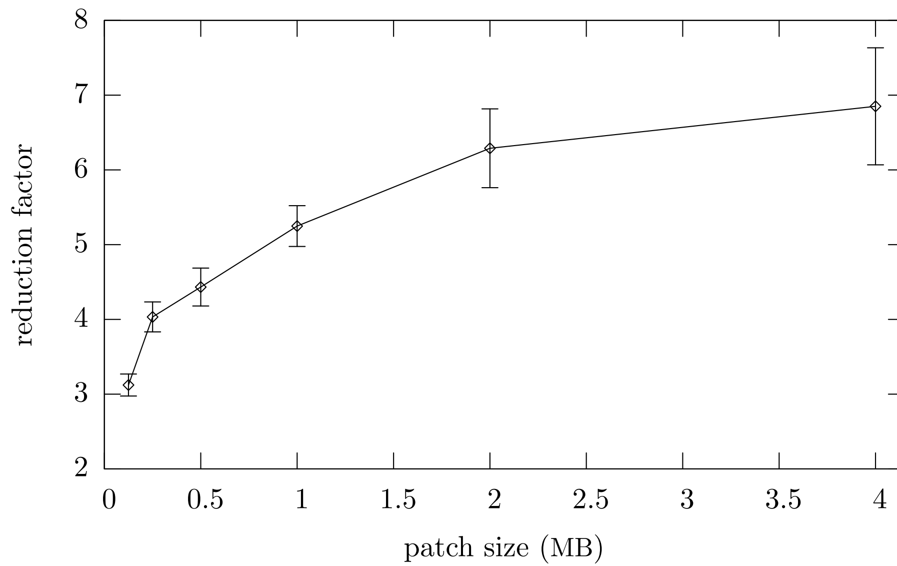


Figure 7.7: Reduction in the Window of Vulnerability due to two-phase dissemination

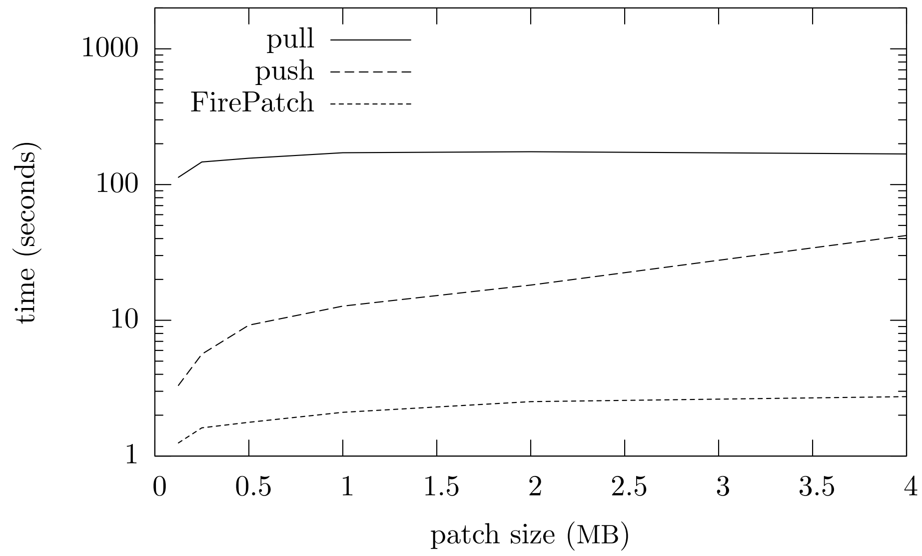


Figure 7.8: Comparison with naïve pull and push

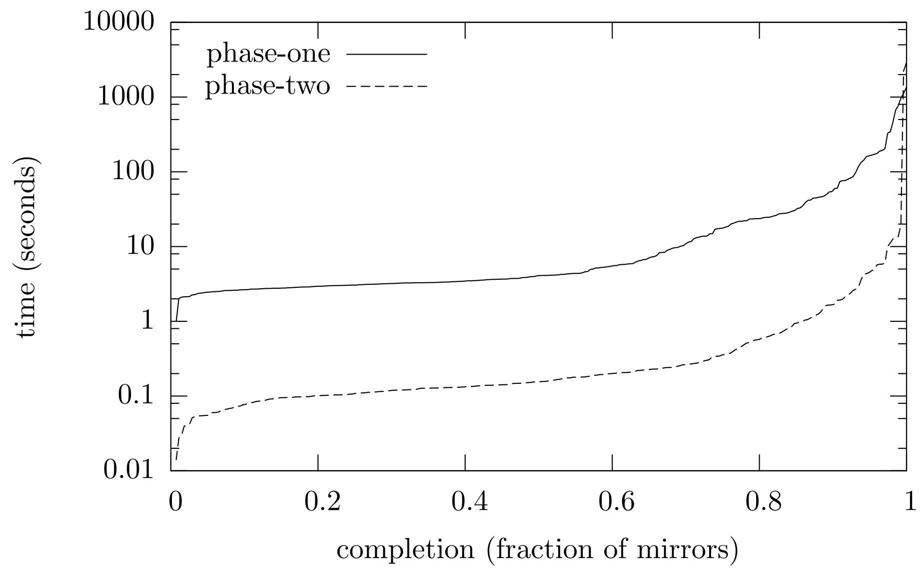


Figure 7.9: Dissemination on PlanetLab

would not make more than one request for a block unless a static timeout of 20 seconds had expired. The performance of pull, push, and FirePatch dissemination for varying patch sizes is shown in Figure 7.8.

To test FirePatch in a more realistic environment, we deployed our code on PlanetLab. We set the fraction of Byzantine mirrors to 20% and removed the bandwidth limitation.

Figure 7.9 shows the result of one experiment that we ran on the 30th of October 2006 where a 2 MB patch and a 20 B key were disseminated in a mesh of 279 mirrors. In this particular setup 80% of the mirrors had completed phase-one within 24 seconds and phase-two within 0.58 seconds. However, we also observed that a few mirrors used a significantly longer time. It turned out that these mirrors had become unresponsive due to heavy CPU and network load from other projects. This was particularly noticeable during phase-two where all but two mirrors received the key within 19 seconds. The last two mirrors became unresponsive between the phases but reintegrated themselves into the mesh and completed phase-two one hour later. Because each client connects to multiple mirrors, such outages will not prevent clients from receiving updates.

Chapter 8

Discussion

This chapter discusses key issues that have not been addressed earlier in this dissertation. We start by exploring alternative designs for membership management. Next, we evaluate our assumption of synchrony. Finally, we show that our solution is applicable for constructing intrusion-tolerant overlay networks other than the one we described in the previous chapter.

8.1 Membership Management

The problem of maintaining membership has been extensively researched. A wide range of algorithms and protocols exist. Membership maintenance is closely related to the theoretical concept of *unreliable failure detection oracles*, which output hints about which processes in a system have crashed. There exist an extensive body of theoretical work on the subject, starting with the seminal papers by Chandra et al. [28, 30]. Many of the algorithms and protocols that have been proposed solve consensus or leader election in all executions. Although, consensus can be used to solve the problem of maintaining membership, the proposed solutions are often impractical, inefficient, or insecure.

8.1.1 No Membership

In broadcast and multicast enabled networks, group communication primitives can be implemented without maintaining membership information. If multiple concurrent groups are to coexist on a single broadcast network, the underlying system must provide some mechanism to isolate messages between each group. For instance, the V system [32] solves this by having each V kernel maintain a local process table with mappings between group iden-

tities and process identities. Messages include the identity of the group that they are addressed to. Each V kernel will then use its local process table to deliver inbound broadcast messages to those processes that have previously registered themselves as members of the destination group.

If the network provides multicast, the V system leverages this by assigning each group identity to a multicast address. If there are more groups than multicast addresses then some groups will share the same address. As with broadcast, groups sharing a multicast address are isolated from one another by filtering messages based on the local process tables.

Currently, neither network level multicast nor network level broadcast are generally available in the Internet [47]. Also, some overlay-network operations, like lookup operations in DHTs, require explicit membership information. Although membership information can be retrieved by broadcasting probe messages, such operations are expensive and, if frequently used, might warrant the use of a group membership protocol.

8.1.2 Partial Membership

An alternative to providing views containing all members or not to provide views at all is for a membership protocol to provide views containing a small subset of all members. We refer to such protocols as *partial membership protocols*. Although the number of members contained in each view might vary from protocol to protocol, the view sizes are usually small compared to the total number of members. As a minimum, a member m's view will contain those members that are to be m's neighbors. The view sizes can grow with the number of members, but the growth rate is usually slow.

Scamp [59] is an epidemic-style membership protocol that, like *Fireflies*, uses a small number of gossip partners in order to increase scalability. Unlike *Fireflies*, Scamp members do not forward membership events to all members but will stop forwarding them when certain criteria are met. The Scamp protocol is not intrusion-tolerant. A Scamp member can become isolated, in which case, that member is required to rejoin. Also, the Scamp mesh can converge to a non-random structure. Cyclon [148] provides an improvement over Scamp by using a distributed shuffling algorithm to maintain randomness even with high node churn. Unlike Cyclon, *Fireflies* maintains a pseudo-random overlay structure by assigning neighbors based on multiple pseudo-random permutations of the membership views. Other partial membership protocols include those of the overlays that are described in Section 2.1.

One reason for providing partial membership rather than full membership is increased scalability. In a full membership protocol, memory requirements

per member will grow linearly with the number of members. Given the availability of cheap memory this is not necessarily a problem. For instance, with member certificates of 163 bytes and a notes of 49 bytes, FiRe will be able to fit approximately 100000 members within 20 MB of memory.

Full membership protocols also require that all group members receive notification of all membership changes. By requiring members to only receive notifications about a subset of the members, partial membership protocols offer a potential increase in scalability due to reduction in network load. This difference might be significant because the churn rate, and hence the subsequent aggregate rate of membership events, tends to grow linearly with the size of the membership.

Although advantageous with regard to scalability, providing members with only a partial view has several disadvantages compared to providing them with full membership information. First, partial membership requires messages to be routed through the overlay structure. As such, messages are more likely to get lost along the way and encounter higher end-to-end latency.

Secondly, existing overlay routing protocols built on top of a full membership protocol can be more efficient than if built on a partial membership protocol. For example, a full membership protocol provides Application-Level Multicast (ALM) protocols with a large candidate set of router nodes for building routing trees, which can significantly increase efficiency and robustness [113].

Third, maintaining strict structures like DHTs requires complex and expensive coordination when having only partial membership information [127, 139, 155].

Fourth, full membership protocols and one-hop routing are seemingly easier to secure than partial membership protocols [22]. Also, as argued in Section 3.4, multi-hop routing is more expensive to make tolerant to omission attack than direct messaging.

8.1.3 View-Synchronous Membership

View-synchronous membership protocols are widely used in Group Communication Systems (GCSs) to facilitate multicast with total or causal ordered delivery semantics in partially synchronous networks. Example of such GCSs are Horus [145], Totem [104], and Transis [48]. A comprehensive survey of these and others, can be found in the study by Chockler et al. [33]. A survey of multicast algorithms is provided by Défago et al. [44].

In a dynamic environment with churn, GCSs must take into consideration the possibility that the group composition can change during message transmission. If this is not addressed, incomplete delivery or missing mes-

sages in the causal ordering might occur. Most GCSs approach this problem by implementing some variant of the *virtual synchrony* property by ordering view changes along with multicast messages such that communication appears synchronous from an external point-of-view [17]. Programming models that provide this property include the *strong* [58], *weak* [58], *optimistic* [143], and *extended* [104] virtual synchrony models.

All variants of virtual synchrony provide the *same view delivery* property, which states that all members that receive a message m must do so in the same view v, and that v is the same or a later view in which m was sent. Strong view synchrony provides the stronger property of *send view delivery*, which adds the requirement that a message must be delivered in the same view in which it is sent. The extended virtual synchrony model allows all components of a partitioned group to continue operating independently and later rejoin. The weak and optimistic virtual synchrony models mitigate the inefficiency of blocking or discarding messages during view changes, as required by the strong virtual synchrony model.

In all these models, it is the responsibility of some membership protocol to ensure that correct and connected members will agree upon the sequence of views such that messages can be delivered correctly. Such view-synchronous membership protocols are run whenever a change in the group has been detected, for instance, due to a suspicion by a failure detector or upon notification of a new member joining. After the network stabilizes, the protocols terminate with a new view installed at each member.

View-synchronous membership can trivially be implemented using consensus. Although agreement on the views is a weaker problem than the problem of consensus, Chandra et al. [29] show that the weaker conditions do not circumvent the impossibility of consensus in asynchronous networks when processes may fail [56].

Scalability Issues. The Totem system [6] organizes members in a logical token ring. Upon detection of a change in the membership, the Totem single ring membership protocol enters a *gather phase* where each member m broadcasts a join message containing m's current view of all group members and their apparent status as either live or crashed. If m receives a join message containing a different view than its own, then m merges that view with its own and restarts the gather phase. If m times out waiting for a join message from one of the members in its view, then that member will be tagged as crashed and m restarts the gather phase. If m receives the same view from all live members within its view, then m considers consensus to be established. In this case and if m has the lowest process identity

of all live members within the proposed view, then m will start circulating a commit token containing the proposed view. The commit token is circulated twice. In the first circulation, members establish that they all have the same view. The second circulation commits that view. If some member m detects that the commit token is lost, or other inconsistencies are detected, then m restarts the gather phase. By recursively removing slow and unstable members from the membership, the Totem single ring membership protocol will terminate in bounded time with an agreed upon view. Due to their serial nature, ring-based token passing protocols are generally not suited for wide-area Internet environment where packet latencies can be high.

The SecureRing protocol [86] is similar to the Totem single ring membership protocol and thus suffers from the same problems. Unlike Totem and *Fireflies*, the SecureRing protocol is augmented with a Byzantine failure detector. Detectable Byzantine behavior that SecureRing can tolerate includes mutant messages, improperly formed messages, and missing messages. A SecureRing member m considers the gather phase completed when m has received join messages containing the same view as its own from at least 2/3 of the members. The SecureRing protocol uses an expensive multicast abstraction where each member will multicast each message at least once before it is delivered locally.

Our approach to intrusion-tolerant membership management is different than that of SecureRing in that *Fireflies* does not try to detect and remove Byzantine members. However, unmistakable evidence of Byzantine behavior, like a malformed and signed message, can be distributed by the *Fireflies* gossip protocol to exclude misbehaving members.

In an extension of the Totem single ring membership protocol, scalability is improved by interconnecting multiple Totem rings using a small set of gateway processes [3, 104]. Each individual ring runs the single ring membership protocol, as previously described, in order to decide on a view. The multiple ring variant of the Totem membership protocol does not tolerate Byzantine failures. It is not clear if the SecureRing protocol can be extended in the same way.

The Horus membership protocol [58] avoids the high-latency problems relating to token ring passing by electing for each view change the process with the lowest identifier as a coordinator. Members exchange views in join messages similarly to Totem, although in Horus they are sent point-to-point to the designated coordinator and not broadcasted to all members. To establish agreement the coordinator broadcasts two views. First, it broadcasts a suggested view. If that view is acknowledged by all members, then the coordinator broadcasts a final view, which all members accept as the next view.

The BFT protocol [26] adds tolerance to Byzantine failures to a Horus like membership protocol by having members broadcast view messages to all members instead of only to the coordinator. Each member computes the coordinator based on the received view messages and sends a view-ack message to that member. After receiving view-ack messages from a member quorum, the primary broadcasts the new view. BFT can tolerate up to 1/3 Byzantine members but does not scale well as each member is involved in several rounds of communication. *Fireflies* provides only probabilistic guarantees.

Network Instability. Although view synchronous membership protocols like Totem, Horus, and SecureRing provide agreement on membership views, they can not guarantee that the delivered views do not contain stale entries since members might crash at any point in time. In particular, the Totem protocol might terminate in a singleton membership if sufficient network instability occurs, which is a highly undesirable situation.

It is also inefficient to require view-synchronous membership protocols to terminate during periods of instability because the delivered views will likely be obsolete before or shortly after delivery, thus requiring a rerun of the expensive agreement protocol. Although more aggressive timeout values will ensure quicker response-time to crashes, such a policy is counter productive with regards to throughput because the system will spend more time running the membership protocol [69]. To reduce load during periods of network instability, the Moshe membership protocol [84] delays view delivery until the network has stabilized. In overlay networks, where frequent and long periods of instability are expected, Moshe might be prevented from delivering updated views for long periods of time. Delaying view delivery does not prevent services from seeing stale view entries.

An alternative approach, used in Horus [58], is the weak virtual synchrony model. This model allows applications to continue sending messages using a temporary view while agreement is established. The temporary view must, however, be a superset of the agreed view. The integration of recovered and added members must therefore be delayed until a new view can be established. The optimistic view synchrony model [143] also uses temporary views but does not pose any restriction on how the temporary view relates to the later agreed view. Instead, applications are allowed to send optimistic messages that will be buffered at the receivers until an agreed view is established. Depending on the outcome of the membership agreement protocol, optimistically delivered messages are either delivered to the application or discarded.

Applicability in Overlay Networks. Although view-synchronous membership protocols are widely used in the context of GCSs, they have not gained popularity as building blocks for overlay networks. The main reason is the inherent lack of scalability in virtual synchronous type of communication, which is due to the cost associated with view changes and packet loss [69]. Birman et al. [16] argue from a practical point-of-view that the virtual synchronous communication is only suited for groups with less than 100 members. Adding tolerance to Byzantine failures through Byzantine agreement only increases overhead. Since most overlay networks do not use strong semantic on message delivery, they can avoid the overhead and complexity of view-synchronous membership protocols.

8.1.4 Weakly-Consistent Membership

Many membership protocols, including *Fireflies*, strive to provide to each member an up-to-date view of all group members, but do not provide agreement on the views. For instance, the epidemic membership protocol described by van Renesse et al. [146], provide only eventual view agreement. Unlike *Fireflies*, this protocol does not distribute suspicion events. Instead, each member increments a *heartbeat* counter in each gossip round and gossips the new counter. Each member can then individually detect the absence of heartbeat increments of other members using timeouts. Although highly robust, overhead is substantial and the protocol does not scale well as each member must receive updates to the heartbeat counters of all live members.

The GulfStream system [55] reduces the load of heartbeat style failure detection by organizing members in a logical ring. Each member then monitors its successor and predecessor in the ring. To reduce the rate of false positives, a member is only considered failed if suspected by both its monitors.

Congress [8] provides membership service for wide-area network environment with weak semantics similar to *Fireflies*. The protocol is made scalable by organizing the members in a hierarchical manner. The system is vulnerable to attack as a malicious member can efficiently prevent communication between subtrees. Congress does not include a failure detection mechanism but requires members to send failure notifications when stopping. Alternatively, it can be extended with an external failure detection mechanism.

The Swim protocol [42] is perhaps most similar to *Fireflies* in that it combines an accusation-rebuttal scheme with a pinging protocol and epidemic dissemination. Unlike *Fireflies*, in each pinging round, a Swim member m_i picks from its view a random member m_j for pinging. If m_i does not receive an acknowledgment from m_j within a specified timeout, m_i selects k

random members from its view as delegates and ask those to ping m_j instead. If the delegates succeed in pinging m_j, then they relay the received acknowledgement back to m_i. If unsuccessful, m_i will issue an accusation for m_j.

Although the Swim pinging scheme will prevent a Byzantine member from keeping failed members within the views of correct members, Swim can not prevent an attacker from repeatedly making false accusations. Also, the delegation of pinging adds to the time it takes for a crashed member to be removed from the views of the correct members. More alarmingly, the Swim protocol allows members to issue failure messages. Upon m_i receiving a failure message for m_j, m_i will immediately remove m_j from its view. There are no restriction on who can generate failure messages for whom. An attacker can therefore use failure messages to falsely claim that a correct member has failed. Unlike *Fireflies*, Swim piggybacks membership events on ping messages, which prevents Swim from taking advantages of set-reconciliation mechanisms to reduce the number duplicate events sent and received. Swim does not impose restrictions on member neighbor selection like *Fireflies*. This makes the Swim protocol more susceptible to DoS attacks.

8.2 Timing Attacks

Making assumption on timings is risky because an attacker might break expected timing bounds by slowing down the system [26]. Because this dissertation assumes synchrony, our solution is subject to such attacks.

First, we assume that correct members have access to accurate clocks. This assumption is unproblematic as most modern computers have accurate hardware clocks embedded on their motherboard. Also, because a member certificate is only valid before its expiry date, clocks must be synchronized with an accuracy in the order of minutes. This is also unproblematic as external sources of time that provide this level of accuracy are readily available (e.g., GPS). Because hardware clocks are local at each member they are highly resilient to DoS attacks.

Secondly, we assume a known upper bound on message delivery between correct members. This assumption on communication synchrony is used in two places within *Fireflies*. First, we assume an upper bound Δ on the time for a message to be disseminated to all members using gossip. Secondly, we assume that a correct member can respond to at least one ping message before the failure detection timeout period, $\tau \times T_{ping}$, expires. If an attacker can slow-down the network sufficiently for messages to take longer than the specified upper bounds, *Fireflies* might not operate correctly. We will in the

following describe the consequences of a successful DoS attack.

8.2.1 Violation of Timing Bounds

For *Fireflies*, a successful DoS attack can lead to the following:

- a correct member is incorrectly considered crashed,
- a crashed member is incorrectly considered live, and
- two members disagree on the status of some member.

Such inconsistencies are unproblematic on a small scale because *Fireflies* does not provide agreement on the views and does not attempt to exclude erratic members. Services built on top of *Fireflies* can consider a member that is under a DoS attack as faulty. The tolerance to Byzantine failures can then be adjusted according to the expected strength of the attacker.

However, if the attacker is sufficiently strong he might slow down a sufficient number of members such that the remaining correct members will have views and neighborhoods containing too many Byzantine members. *Fireflies* can not withstand such massive all-out timing attacks. Due to our design, we conjecture that the resources needed to execute such attacks is outside the reach of most people and organizations. In practice, *Fireflies* might not be the weakest link against such an adversary.

8.2.2 Weaker Models of Synchrony

There are many possible system models weaker than the one assumed in this dissertation and that would make it harder for an attacker to break assumed timing bounds. Of particular interest are those that are sufficiently strong to implements a leader election oracle, Ω. Having Ω, consensus, and hence membership, can be solved.

One way to weaken the model of synchrony is to assume timing bounds, but not assume that those bounds are known. In such a model, Ω can be implemented using a heartbeat approach, as described in Section 8.1.4, but modified such that timeout values are gradually increased until no member is falsely suspected [94].

Several recent works have shown that Ω can be implemented in environments where only some links are *eventually timely*. Such links have the property that there is some time after which all messages sent take a bounded time to be received. For instance, Aguilera et al. [5] show that in a network of N processes where f can fail, an Ω oracle is possible if there exists some

member, m, with at least f outbound links that are eventually timely. Hutle et al. [76] weaken this model further, requiring that the condition on m only eventually occurs.

Although these results show that consensus can be implemented under surprisingly weak assumptions of synchrony, implementing Ω under these assumptions would be impractical and prohibitively expensive. Hutle et al. argue that there is a clear tradeoff between message complexity and relaxed synchrony [76]. For instance, the model of Aguilera et al. requires f links to carry messages forever in some runs. The weaker model of Hutle et al. increases this number to $nf/2$.

Although the Internet is commonly considered asynchronous, we argue that our stronger assumption on message delivery bounds are reasonable because our timing bounds, which are in the order of minutes, are large in comparison with the expected Internet end-to-end latencies and packet-loss rate.

8.3 Applicability

Although FiRe maintains membership information in an intrusion-tolerant manner, services built on top of it do not automatically inherit this property [129]. For instance, to ensure safe storage of files, a service built on top of FiRe must still ensure that each file is replicated to a sufficiently large number of members.

Numerous approaches address Byzantine failures in distributed systems, starting with [93, 110]. One popular approach is to implement services using State Machine Replication (SMR) [92, 131]. In essence, SMR works by having each server start in the same state, then they execute the same set of deterministic instructions with the same input. Hence, all correct replicas produce the same output. Clients mask invalid responses from faulty replicas using majority voting. Generally, SMR systems are not considered scalable because each write operation requires involvement of all member processes. Also, SMR limits a system's aggregate storage capacity to the smallest storage capacity available at a member. Systems and protocols based on SMR include BFT [26], SecureRing [87], and Rampart [118].

Byzantine quorums system [98, 99, 100] offer an alternative to SMR. A quorum system divides members into multiple overlapping subsets. Each quorum can operate on behalf of all servers. The intersection between the quorums guarantees consistency of replicated data. By having a sufficiently large overlap, Byzantine failures can be masked.

By combining optimistic execution with quorum operations, the Q/U pro-

tocol [1] is shown to produce a peak throughput that is four times higher than the BFT protocol [26] when five faults can be tolerated. However, the throughput of the protocol drops when there is contention among concurrent update operations. To resolve such contentions, the Q/U protocol employs exponential back-off. The HQ protocol [38] improves upon the Q/U protocol by resolving contention using the BFT protocol.

Although both quorums systems and SMR systems can be built on top of an intrusion-tolerant membership protocol like *Fireflies*, they might be to expensive to run on a large number of members. Rodrigues et al. [125] suggest that the scalability limitations of these protocols can be mitigated using a hybrid P2P architecture where smaller sub-groups of members specialize at certain tasks. For instance, they propose a scheme where a small group of members are selected to provide a configuration service that maintains membership information. Similar schemes have been implemented within P2P file systems like OceanStore [120] and FARSITE [2]. How general hybrid P2P architectures can be constructed on top of FiRE is left as future work, although we have already approached the issue through the separation of clients and mirror roles within FirePatch, as described in Chapter 7.

We will in the remanding of this section outline two other overlay networks that can be constructed using FiRE.

8.3.1 One-Hop Distributed Hash Table

Intrusion-tolerant DHT messaging can be trivially implemented on a full membership protocol. Assuming object and member identifiers are chosen from the same identifier space, a member can simply consult its view to find the member whose identity is closest to the object identity. That member is then the destination, and messages can be sent directly to it. Such an implementation is called an One Hop Distributed Hash Table (OHDHT)[1] [68], as messages are not routed through intermediate members. If replication is required to, for instance, securely store a file, multiple *Fireflies* member rings can be used to assign to each message multiple destinations.

8.3.2 Multimedia Streaming

SecureStream [73] is an intrusion-tolerant multimedia diffusion protocol that layers a push-pull messaging scheme [109] on top of FiRE, in a similar manner as FirePatch. Like security patch distribution, multimedia dissemination is

[1]Not to be confused with an $O(1)$ hop DHT, although OHDHTs are members of that class.

sensitive to delay. However, within a multimedia stream, late packets are considered to be permanently lost and will not be recovered. For instance, SecureStream members only request data that are within a moving window of interest. To reduce overhead of packed authentication, SecureStream groups hashes of multiple packets into a special linear digest message. The system ensures that digest messages are delivered to members before they receive the corresponding data messages.

By not forwarding data messages and digests, and by over requesting, an attacker might try to delay the reception of a multimedia segment such that it is no longer usable for the receivers. With a sustained rate of 300 Kilobits per second (kbps), SecureStream is shown to deliver a higher ratio of packets within acceptable time than SplitStream [24] when under attack.

Chapter 9

Conclusions

For an Internet service to accommodate increasing load and complexity, hardware resources must be added to it. P2P overlay networks offer an alternative by utilizing existing hardware resources available in the commodity class computers that are owned by those that use the service. Because an attacker can gain control of overlay-network components simply by joining, overlay network should be constructed in an intrusion-tolerant manner such that they are able to fight maliciously induced Byzantine failures.

9.1 Results

This dissertation identifies intrusion-tolerant membership management as a key function of an intrusion-tolerant overlay network. As stated in Section 1.2.3, the thesis of this dissertation is:

> *Using epidemic techniques it is possible to build overlay networks and peer-to-peer systems that strike a useful balance between intrusion-tolerance and resource usage.*

To evaluate our thesis we designed *Fireflies*, a weakly consistent membership management protocol. Next, we implemented FiRe, an overlay-network framework based on the *Fireflies* protocol. As a case study, we implemented a software dissemination network, FirePatch, using FiRe.

Our solution involves a central CA that is entrusted to assign to each member a random identity. The CA is also required to do background checks on each member such that the fraction of Byzantine members among all members is probabilistically upper bounded by a certain P_{byz}. Although having a centralized CA is not ideal for overlay networks, we are currently not aware of other suitable mechanisms to bound the fraction of Byzantine

members. The CA is considered an off-line component in that it is only involved in signing and revoking member certificates. In particular, the CA is not involved in maintaining up-to-date membership information. Instead, this is done by having members monitor one another and issue *accusations* (failure notices) whenever a member is suspected to have crashed.

If a member is falsely accused, it has the opportunity to issue a *rebuttal* before it is removed from the views of correct members. For scalability reasons, a member can only monitor a subset of the other members. The challenge is to guarantee that each member has at least one correct monitor and, at the same time, prevent Byzantine members from frequently falsely accusing correct members. We address this by organizing members in k circular address spaces, or member rings. Each ring is a pseudo-random permutation of the membership list and is calculated deterministically from the secure hash of the member identities in combination with a ring identifier. Each ring imposes successor and predecessor relationships on the members such that, with k rings, each member has a total of k successors and k predecessors. By picking a sufficiently large number of rings, each member will be able to disable all Byzantine monitors while, at the same time, ensuring that all members have at least one correct monitor.

Accusations and rebuttals are disseminated to all members using a secure broadcast channel, which we implemented by having members gossip with one another. To limit the ability of an attacker to launch an all-out DoS attack, a member is only allowed to gossip with a small subset of the members. We assign gossip partners by organizing the members in a strict pseudo-random mesh structure based on a set of member rings, similarly to how we assign monitoring responsibilities. By adjusting the number of gossip rings, correct members form a connected sub-mesh with high probability, rendering omission attack ineffective.

In Section 1.2.3 we listed three properties that would measure the success of our solution: scalability, intrusion-tolerance, and applicability. We will in the following show to what extent each of these attributes are fulfilled.

- *Intrusion-tolerance.* Chapter 3 listed two key design requirements for intrusion-tolerant membership management. As shown in Figure 9.1, *Fireflies* fulfills Requirement 1 using a central CA that safely assigns member identities. Requirement 2 is fulfilled by imposing a strict pseudo-random structure on the overlay-network topology, which is facilitated by the *Fireflies* membership rings. Our claims are supported by mathematical and statistical reasoning and simulations. Our evaluations have shown that our implementation of *Fireflies* in FiRe can withstand passive and active membership attacks from as many as 20%

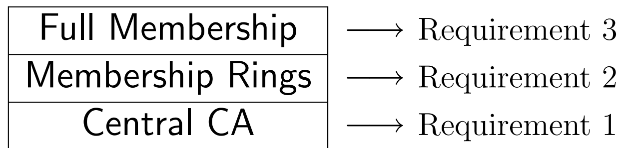


Figure 9.1: The *Fireflies* overlay-network stack

of the members. We have also observed in our experiments with FiRe on PlanetLab various unexpected behavior on certain nodes. Sometimes the local file system on a node disappears or runs out of disk space, preventing logs to be written. Sometimes nodes are wrongly configured with a non-routable IP address. Sometimes nodes become unaccessible due to network outages, CPU overload, or rarely, actual crashes. There have even been bugs in FiRe agents causing them to crash or behave erratically. But the FiRe infrastructure as a whole has survived all of these problems.

- *Scalability.* We have shown that, in a dynamic PlanetLab like environments, FiRe can maintain membership information in overlay networks with 280 members. Measured overhead in this setting is in the order of 50 Bps under normal load, and up to 500 Bps when approximately 1/4 of the members crash at the same time. Our simulations have shown that the rate of membership events grow close to linearly with the number of members. This indicates that FiRe is able to support group sizes in the order of thousands of members on current wide-area Internet network technology. By providing full membership information, applications can avoid the cost of multi-hop overlay routing, thus Requirement 3 is fulfilled as shown in Figure 9.1.

- *Applicability.* To show the applicability of our solution, Chapter 7 describes and evaluates FirePatch, an intrusion-tolerant software security patch distribution mechanism, built using FiRe. By combining encryption, replication, and sandboxing with the intrusion-tolerant membership service provided by FiRe, FirePatch enables end-users to fight attacks by hackers that exploit software vulnerability by reverse engineering software patches. Section 8.3 discusses other areas of applicability, including a multimedia-streaming network that has been built using FiRe.

There are clear limitations to what our solution can offer. Byzantine members can disguise themselves as correct members by executing the pro-

tocol, or as crashed members by not executing at all, and so a correct member can not determine which members are Byzantine unless they reveal themselves as such by sending messages that prove they are not following the protocol. Also, views trail membership changes, and might be stale at any time. *Fireflies* does not provide virtual synchrony properties like agreement on the views. Instead, it provides eventual and probabilistic consistency among the views of the correct members. Given constant churn, members might never reach agreement on the state of the membership. Although, with high probability, correct members' views will agree on the set of long time correct members and on the set of long time crashed members. It is, however, possible that long time crashed members are temporarily included in the views of correct members due to cascading invalidation of accusations. The *Fireflies* protocol makes such inconsistent states infrequent, with probabilistic guarantees.

While our solution is not as scalable as Pastry, not as secure as BFT, and not as applicable as Horus, we have found a novel combination of these attributes that is useful for constructing intrusion-tolerant overlay networks.

9.2 Future Work

Although this dissertation has focused on intrusion-tolerance, the masking of Byzantine faults is, as argued in Section 1.2, only one of many overlapping and complementary tools for increasing the level of security in a system. The integration of a larger set of such techniques within FiRe is an interesting topic of future work. For instance, mechanisms for host-based intrusion detection [52, 126], recovery [35, 63], and pro-active reboots [25, 26] could be used to limit the ability of an attacker to break in and maintain control of computers that are not his own. Here, one interesting approach is to minimize the inconvenience of downtime by using high-level application mobility to move running applications to a temporary location while the underlying system is rebooting. We observed in an earlier paper that many legacy application could be instrumented with such mobility without any modification to their code [77].

An important approach to fighting attacks is to detect and exclude members that are misbehaving [70, 86, 88]. For instance, FiRe should be extended with a rate monitoring module that detects and excludes members that generate excessive amounts of notes and false accusations. Also, incorrectly signed or badly formatted messages are clear indications of misbehaving members, which should lead to exclusion of the sender. Future work should extend FiRe with this capability.

Another interesting direction for future work is, as described in Section 8.3, to increase the scalability and efficiency of FiRe by using a hybrid P2P structures where members are specialized to perform certain tasks. Members can, for instance, be dynamically allocated to specialization subgroups based on certain properties like their disk, CPU, or network capacity. Such structures have already been proposed [2, 120, 125] and seem to be a promising direction for P2P overlay networks.

Appendix A

Publications

This dissertation is based on work presented in the following four publications:

Paper I

> Håvard Johansen and Dag Johansen. Improving object search using hints, gossip, and supernodes. In *Proceedings of the 21st Symposium on Reliable Distributed Systems: Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 336–340. IEEE, October 2002.

In this paper we present a P2P object search protocol that reduces network load by caching past queries as hints for future searches. The idea is that if some member m submits a query, it is likely to find and download matching objects. Member m is therefore a likely candidate to match similar queries. Such queries should therefore be forwarded to m in order to improve forwarding accuracy. A hint gossiping scheme is added to prevent objects from becoming isolated and to increase efficiency. The paper show that hint caching increases efficiency while providing similar lever of recall compared to a Gnutella type of query flooding. The paper does not address intrusion-tolerance and membership maintenance. The protocol is described in Section 2.1 as part of our survey of existing systems.

Paper II

> Dag Johansen, Håvard Johansen, and Robbert van Renesse. Environment mobility—moving the desktop around. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 150–154. ACM, October 2004.

In this paper we outline how high-level application mobility can be used to move a user's computational environment from one computer to another. We observed that many legacy applications could be instrumented with such mobility by adding an external signalling and state-transfer process. The paper motivates this dissertation because it identified the need for an intrusion-tolerant overlay network that makes marshalled application state available to a user as he moves from one computer to another. This dissertation lists the use of high-level application mobility as a possible support function for pro-active reboots in Section 9.2.

Paper III

> Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proceedings of the 1th EuroSys Conference*, pages 3–13. ACM, October 2006.

In this paper we present *Fireflies* as described in this dissertation. The bulk of the paper is contained within Chapter 4 and Chapter 6 of this dissertation. A few items are also contained in Chapter 5.

Paper IV

> Håvard Johansen, Dag Johansen, and Robbert van Renesse. FirePatch: Secure and time-critical dissemination of software patches. In *Proceedings of the 22nd International Information Security Conference*, pages 373–384. IFIP, Springer-Verlag, May 2007.

In this paper we present FirePatch as described as a case study in Chapter 7 of this dissertation. The papers shows that FiRe and *Fireflies* can be used to solve a real and important problem.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 59–74. ACM, October 2005.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14. USENIX, December 2002.

[3] Deborah A. Agarwal, Louise E. Moser, Peter M. Melliar-Smith, and Ravi K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[4] Sachin Agarwal and Ari Trachtenberg. Practical set reconciliation implementation. Software Version Beta 0.998, Boston University, `http://ipsit.bu.edu/programs/reconcile/`, June 2004.

[5] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23rd Symposium on Principles of Distributed Computing*, pages 328–337. ACM, July 2004.

[6] Yair Amir, Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, and Paul Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[7] Thomas Anderson, Larry Peterson, Scott Shenker, and Jonathan Turner. Overcoming the Internet impasse through virtualization. *IEEE Computer*, 38(4):34–41, April 2005.

[8] T. Anker, G. Chockler, D. Dolev, and I. Keidar. Scalable group membership services for novel applications. In *Networks in Distributed Computing*, volume 45 of *DGIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–42. American Mathematical Society, January 1998.

[9] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of Vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, December 2000.

[10] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[11] Gal Badishi, Idit Keidar, and Amir Sasson. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. *IEEE Transactions on Dependable and Secure Computing*, 3(1):45–61, March 2006.

[12] Paul Barford and Joel Sommers. Comparing probe- and router-based packet-loss measurement. *IEEE Internet Computing*, 8(5):50–56, October 2004.

[13] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management–part 1: General (revised). Special publication 800-57, National Institute of Standards and Technology, May 2006. Draft.

[14] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[15] Rida A. Bazzi and Goran Konjevod. On the establishment of distinct identities in overlay networks. In *Proceedings of the 24th Symposium on Principles of Distributed Computing*, pages 312–320. ACM, July 2005.

[16] Kenneth P. Birman, Bob Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *Proceedings of the 2000 Information Survivability Conference and Exposition*, pages 149–161. DARPA, January 2000.

[17] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 123–138. ACM, November 1987.

[18] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[19] Jean-Chrysostome Bolot. Characterizing end-to-end packet delay and loss in the Internet. *Journal of High Speed Networks*, 2(3):305–323, December 1993.

[20] Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen. A trend analysis of exploitations. In *Proceedings of the 2001 Symposium on Security and Privacy*, pages 214–229. IEEE, May 2001.

[21] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Fast approximate reconciliation of set differences. Technical report 2002-019, Boston University, July 2002.

[22] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*. USENIX, December 2002.

[23] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical report MSR-TR-2003-52, Microsoft Research, 2003.

[24] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 298–313. ACM, October 2003.

[25] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*. USENIX, October 2000.

[26] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002.

[27] Olivier Chalouhi, Alon Rohter, and Paul Gardner. Azureus. Software version 2.5.0.0, SourceForge, `http://sourceforge.net/projects/azureus/`, August 2006.

[28] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[29] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*, pages 322–330. ACM, 1996.

[30] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[31] Andrew Chasin. The Gnutella protocol specification. Specification Version 0.41, Clip2 Distributed Search Solutions, June 2001. Document revision 1.2.

[32] David R. Cheriton and Willy Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[33] Gregory V. Chockler, Idid Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[34] Fan Chung and Linyuan Lu. The diameter of random sparse graphs. *Advances in Applied Math*, 26(4):257–279, May 2001.

[35] Tzi cker Chiueh and Dhruv Pilania. Design, implementation, and evaluation of a repairable database management system. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 179–188. IEEE, December 2004.

[36] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: Proceedings of the 2000 International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes on Computer Science*, pages 46–66. Springer-Verlag, July 2001.

[37] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end containment of Internet worms. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 133–147. ACM, October 2005.

[38] James Cowling, Daniel Myers, Barbara Liskov Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 2006 Symposium on Operating System Design and Implementation*, pages 177–190. USENIX, November 2006.

[39] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using a peer-to-peer lookup service. In *Peer-to-Peer Systems: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes on Computer Science*, pages 155–165. Springer-Verlag, March 2002.

[40] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 202–215. ACM, October 2001.

[41] Yogen K. Dalal and Robert M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, December 1978.

[42] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 303–312. IEEE, June 2002.

[43] Susheel Daswani and Adam Fisk. Gnutella UDP extension for scalable searches (GUESS). Specification Version 0.1, Lime Wire LLC, `http://www.limewire.org/`, August 2002.

[44] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[45] Peter J. Denning. Is computer science science? *Communications of the ACM*, 48(4):27–31, April 2005.

[46] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, January 1989.

[47] Christophe Diot, Brian N. Levine, Bryan Lyles, Hassan Kassem, and Doug Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, February 2000.

[48] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, April 1996.

[49] Wim Dorst. The quintessential Linux benchmark. *Linux Journal*, 21:online, January 1996.

[50] John R. Douceur. The Sybil attack. In *Peer-to-Peer Systems: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes on Computer Science*, pages 251–260. Springer-Verlag, March 2002.

[51] Peter Druschel and Antony Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 75–80. IEEE, May 2001.

[52] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th Symposium on Operating System Design and Implementation*, pages 211–224. USENIX, December 2002.

[53] Donald E. Eastlake, III and Paul E. Jones. US secure hash algorithm 1 (SHA1). RFC 3174, The Internet Society, September 2001.

[54] Pál Erdös and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[55] Sameh A. Fakhouri, Germán Goldszmidt, Michael Kalantar, John A. Pershing, and Indranil Gupta. GulfStream–a system for dynamic topology management in multi-domain server farms. In *Proceedings of the 3rd International Conference on Cluster Computing*, pages 55–62. IEEE, October 2001.

[56] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[57] Halvar Flake. Structural comparison of executable objects. In *Proceedings of the 2004 Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pages 161–173. German Informatics Society, July 2004.

[58] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in Horus. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, pages 140–149. IEEE, October 1996.

[59] Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, February 2003.

[60] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 193–206. ACM, October 2003.

[61] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The digital distributed system security architecture. In *Proceedings of the 12th National Computer Security Conference*, pages 305–319. NIST, October 1989.

[62] Christos Gkantsidis, Thomas Karagiannis, Pablo Rodriguez, and Milan Vojnović. Planet scale software updates. *ACM SIGCOMM Computer Communication Review*, 36(4):423–434, October 2006.

[63] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 163–176. ACM, October 2005.

[64] Lawrence A. Gordon, Martin P. Loeb, William Lucyshyn, and Robert Richardson. 2006 CSI/FBI computer crime and security survey. Annual survey, Computer Security Institute, 600 Harrison Street, San Francisco, CA, USA, July 2006.

[65] Karl Taro Greenfeld. Meet the Napster. *TIME Magazine*, 156(14), October 2000.

[66] Saikat Guha and Paul Francis. Characterization and measurement of TCP traversal through NATs and firewalls. In *Proceedings of the 5th Internet Measurement Conference*, pages 199–211. USENIX, October 2005.

[67] Krishna P. Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and*

Protocols for Computer Communication, pages 381–394. ACM, August 2003.

[68] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. One hop lookups for peer-to-peer overlays. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 7–12. USENIX, May 2003.

[69] Indranil Gupta, Kenneth P. Birman, and Robbert van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International*, 18(3):165–184, June 2002.

[70] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. The case for Byzantine fault detection. In *Proceedings of the 2nd Workshop on Hot Topics in System Dependability*. USENIX, November 2006.

[71] Frank Harary. The maximum connectivity of a graph. *Proceedings of the National Academy of Sciences of the United States of America*, 48(7):1142–1146, July 1962.

[72] Tom Hargreaves. The FastTrack protocol. Specification Revision 1.19, giFT-FastTrack, `http://gift-fasttrack.berlios.de/`, July 2004.

[73] Maya Haridasan and Robbert van Renesse. Defense against intrusion in a live streaming multicast system. In *Proceedings of the 6th International Conference on Peer-to-Peer Computing*, pages 185–192. IEEE, September 2006.

[74] Nikki Hemming. KaZaa. Software Version 3.2.5, Sherman Networks, `http://www.kazaa.com/`, November 2006.

[75] John B. Horrigan. Home broadband adoption 2006. Report, PEW Internet & American Life Project, 1615 L Street, NW, Suite 700, Washington, DC 20036, USA, May 2006.

[76] Martin Hutle, Dahlia Malkhi, and Ulrich Schmid Lidong Zhou. Chasing the weakest system model for implementing Ω and consensus. Research report 74/2005, Technische Universität Wien, July 2006.

[77] Dag Johansen, Håvard Johansen, and Robbert van Renesse. Environment mobility—moving the desktop around. In *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*, pages 150–154. ACM, October 2004.

[78] Dag Johansen, Robbert van Renesse, and Fred Schneider. WAIF: Web of asynchronous information filters. In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes on Computer Science*, pages 81–86. Springer-Verlag, April 2003.

[79] Håvard Johansen, André Allavena, and Robbert van Renesse. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proceedings of the 1th EuroSys Conference*, pages 3–13. ACM, October 2006.

[80] Håvard Johansen, Dag Johansen, and Robbert van Renesse. FirePatch: Secure and time-critical dissemination of software patches. In *Proceedings of the 22nd International Information Security Conference*, pages 373–384. IFIP, Springer-Verlag, May 2007.

[81] Håvard D. Johansen and Dag Johansen. Improving object search using hints, gossip, and supernodes. In *Proceedings of the 21st Symposium on Reliable Distributed Systems: Workshop on Reliable Peer-to-Peer Distributed Systems*, pages 336–340. IEEE, October 2002.

[82] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 91–104. ACM, October 2005.

[83] Ari Juels and John Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium*, pages 151–165. The Internet Society, February 1999.

[84] Idit Keidar, Jeremy Sussman, Keith Marzullo, and Danny Dolev. Moshe: A group membership service for WANs. *ACM Transactions on Computer Systems*, 20(3):191–238, August 2002.

[85] Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, March 2003.

[86] Kim Potter Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, pages 317–326. IEEE, January 1998.

[87] Kim Potter Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, November 2001.

[88] Kim Potter Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, January 2003.

[89] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using random subsets to build scalable network services. In *Proceedings of the 4th Symposium on Internet Technologies and Systems*. USENIX, March 2003.

[90] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, and Amin Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of the 19th Symposium on Operating Systems Principles*, pages 282–297. ACM, October 2003.

[91] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 190–201. ACM, November 2000.

[92] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[93] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[94] Mikel Larrea, Antonio. Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proceedings of the 19th Symposium on Reliable Distributed Systems*, pages 52–59. IEEE, October 2000.

[95] Richard J. Larsen and Morris L. Marx. *An Introduction to Mathematical Statistics and Its Applications*. Prentice Hall, 3rd edition, 2001.

[96] Glyph Lefkowitz. Twisted Core. Software version 2.2, Twisted Matrix Labs, `http://twistedmatrix.com/`, February 2006.

[97] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[98] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, October 1998.

[99] Dahlia Malkhi, Michael Reiter, and Avishai Wool. The load and availability of Byzantine quorum systems. In *Proceedings of the 16th Symposium on Principles of Distributed Computing*, pages 249–257. ACM, August 1997.

[100] Dahlia Malkhi and Michael K. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, April 2000.

[101] David L. Mills. A brief history of NTP time: memoirs of an Internet timekeeper. *ACM SIGCOMM Computer Communication Review*, 33(2):9–21, April 2003.

[102] Yaron Minsky and Ari Trachtenberg. Practical set reconciliation. Technical report 2002-01, Boston University, 2002.

[103] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2212–2218, September 2003.

[104] Louise E. Moser, Peter M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: a fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[105] Kenneth E. Murphy, Charles M. Carter, and Steven O. Brown. The exponential distribution: the good, the bad and the ugly. A practical guide to its implementation. In *Proceedings of the 2002 Annual Reliability and Maintainability Symposium*, pages 550–556. IEEE, January 2002.

[106] Musiclab, LLC. BearShare. Software version 6, Musiclab, LLC., `http://www.bearshare.com/`, November 2006.

[107] Rafael R. Obelheiro and Joni da Silva Fraga. A lightweight intrusion-tolerant overlay network. In *Proceedings of the 9th International*

Symposium on Object and Component-Oriented Real-Time Distributed Computing, pages 496–503. IEEE, April 2006.

[108] Andy Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, March 2001.

[109] Vinay S. Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Peer-to-Peer Systems IV: Revised Papers from the 4th International Workshop on Peer-to-Peer Systems*, volume 3640 of *Lecture Notes on Computer Science*, pages 127–140. Springer-Verlag, February 2005.

[110] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[111] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of the 7th Symposium on Operating System Design and Implementation*, pages 351–366. USENIX, November 2006.

[112] Larry Peterson and Timothy Roscoe. The design principles of PlanetLab. *ACM SIGOPS Operating Systems Review*, 40(1):11–16, January 2006.

[113] Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. Evaluating DHT-based service placement for stream-based overlays. In *Peer-to-Peer Systems IV: Revised Papers from the 4th International Workshop on Peer-to-Peer Systems*, volume 3640 of *Lecture Notes on Computer Science*, pages 275–286. Springer-Verlag, February 2005.

[114] Charles G. Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Symposium on Parallel Algorithms and Architectures*, pages 311–320. ACM, June 1997.

[115] Venugopalan Ramasubramanian and Emin Gün Sirer. The design and implementation of a next generation name service for the Internet. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 331–342. ACM, September 2004.

[116] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 161–172. ACM, August 2001.

[117] Thomas Reidemeister, Klemens Bohm, Paul A. S. Ward, and Erik Buchmann. Malicious behaviour in content-addressable peer-to-peer networks. In *Proceedings of the 3rd Annual Communication Networks and Services Research Conference*, pages 319–326, Washington, DC, USA, May 2005. IEEE.

[118] Michael K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd Conference on Computer and Communications Security*, pages 68–80. ACM, November 1994.

[119] Michael K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[120] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore prototype. In *Proceedings of the 2nd Conference on File and Storage Technologies.* USENIX, April 2003.

[121] John Risson and Tim Moors. Survey of research towards robust peer-to-peer networks: Search methods. Technical report UNSW-EE-P2P-1-1, University of New South Wales, Australia, September 2004.

[122] Jordan Ritter. Why Gnutella can't scale. No, really. Report, Darkridge Security Solutions, `http://www.darkridge.com/`, February 2001.

[123] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[124] Rodrigo Rodrigues and Charles Blake. When multi-hop peer-to-peer routing matters. In *Peer-to-Peer Systems III: Revised Papers from the 3rd International Workshop on Peer-to-Peer Systems*, volume 3279 of *Lecture Notes on Computer Science*, pages 112–122. Springer-Verlag, February 2004.

[125] Rodrigo Rodrigues, Barbara Liskov, and Liuba Shrira. The design of a robust peer-to-peer system. In *Proceedings of the 10th ACM SIGOPS European Workshop*, pages 117–124. ACM, September 2002.

[126] Martin Roesch. Snort—lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on System Administration*, pages 229–238. USENIX, November 1999.

[127] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 2001 International Conference on Distributed Systems Platforms*, volume 2218 of *Lecture Notes on Computer Science*, pages 329–350. IFIP/ACM, Springer-Verlag, November 2001.

[128] Antony I. T. Rowstron and Peter Druschel. Storage management and caching in PAST, a large scale, persistent peer-to-peer storage utility. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 188–201. ACM, October 2001.

[129] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[130] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of the 10th Multimedia Computing and Networking Conference*. SPIE, January 2002.

[131] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[132] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 1–16. ACM, October 2005.

[133] Adi Shamir and Eran Tromer. Factoring large numbers with the TWIRL device. In *Advances in Cryptology–CRYPTO 2003*, volume 2729 of *Lecture Notes on Computer Science*, pages 1–26. Springer-Verlag, October 2003.

[134] Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. Defending against Eclipse attacks on overlay networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*. ACM, September 2004.

[135] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems: Revised Papers from the 1st International Workshop on Peer-to-Peer Systems*, volume 2429 of *Lecture Notes on Computer Science*, pages 261–269. Springer-Verlag, March 2002.

[136] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using PlanetLab for network research: Myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, January 2006.

[137] Raj Srinivasan. XDR: External data representation standard. RFC 1832, Sun Microsystems, August 1995.

[138] Mudhakar Srivatsa and Ling Liu. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, pages 252–261. IEEE, December 2004.

[139] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 149–160. ACM, August 2001.

[140] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[141] Brad Stone. A lively market, legal and not, for software bugs. *The New York Times*, online, January 30 2007.

[142] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th Internet Measurement Conference*, pages 189–202. ACM, October 2006.

[143] Jeremy Sussman, Idit Keidar, and Keith Marzullo. Optimistic Virtual Synchrony. In *Proceedings of the 19th Symposium on Reliable Distributed Systems*, pages 42–51. IEEE, October 2000.

[144] Marvin Theimer and Michael B. Jones. Overlook: scalable name service on an overlay network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 52–61. IEEE, July 2002.

[145] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Communications of the ACM*, 39(4):76–83, April 1996.

[146] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the 1998 International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware)*, pages 55–70. IFIP, Springer-Verlag, September 1998.

[147] Guido van Rossum. Python. Software Release 2.3.5, PythonLabs, `http://www.python.org`, February 2005.

[148] Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured P2P overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[149] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th Symposium on Operating Systems Principles*, pages 148–162. ACM, October 2005.

[150] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 193–204. ACM, September 2004.

[151] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology–CRYPTO 2005*, volume 3621 of *Lecture Notes on Computer Science*, pages 17–36. Springer-Verlag, August 2005.

[152] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, July 1999.

[153] Beverly Yang and Hector Garcia-Molina. Designing a super-peer network. In *Proceedings of the 19th International Conference on Data Engineering*, pages 49–60. IEEE, March 2003.

[154] Xinyan Zhang, Jiangchuan Liu, Bo Li, and Tak-Shing Peter Yum. CoolStreaming/DONet: A data-driven overlay network for for peer-to-peer live media streaming. In *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 2102–2111. IEEE, March 2005.

[155] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical report UCB/CSD-01-1141, University of California, Berkeley, April 2001.

[156] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

Abbreviations

ALM Application-Level Multicast

API Application Programming Interface

Bps bytes per second

B Byte

CAN Content-Addressable Network

CA Certificate Authority

CDN Content-Distribution Network

DHT Distributed Hash Table

DNS Domain Name System

DoS Denial-of-Service

EST Eastern Standard Time

GB Gigabyte

GCS Group Communication System

GHz Gigahertz

Gbps Gigabits per second

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

ICMP Internet Control Message Protocol

IP Internet Protocol

kBps	Kilobytes per second
kB	Kilobyte
kbps	Kilobits per second
MB	Megabyte
MIPS	Million Instructions Per Second
MTTF	Mean Time To Failure
MTTR	Mean Time To Recovery
NAT	Network Address Translation
NTP	Network Time Protocol
OHDHT	One Hop Distributed Hash Table
P2P	Peer-to-Peer
SCA	Self-Certifying Alert
SHA	Secure Hash Algorithm
SMR	State Machine Replication
SSL	Secure Socket Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
WAIF	Wide-Area Information Filtering
WoV	Window of Vulnerability
XDR	External Data Representation