Fireflies: A Secure and Scalable Membership and Gossip Service

HÅVARD D. JOHANSEN, UIT The Arctic University of Norway
ROBBERT VAN RENESSE, Cornell University
YMIR VIGFUSSON, Emory University and Reykjavík University
DAG JOHANSEN, UIT The Arctic University of Norway

An attacker who controls a computer in an overlay network can effectively control the entire overlay network if the mechanism managing membership information can successfully be targeted. This article describes Fireflies, an overlay network protocol that fights such attacks by organizing members in a verifiable pseudorandom structure so that an intruder cannot incorrectly modify the membership views of correct members. Fireflies provides each member with a view of the entire membership, and supports networks with moderate total churn. We evaluate Fireflies using both simulations and PlanetLab to show that Fireflies is a practical approach for secure membership maintenance in such networks.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms: Design, Performance, Security

Additional Key Words and Phrases: Byzantine failures, gossip, membership management, overlay network

ACM Reference Format:
Håvard D. Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. 2015. Fireflies: A secure and scalable membership and gossip service. ACM Trans. Comput. Syst. 33, 2, Article 5 (May 2015), 32 pages.
DOI: http://dx.doi.org/10.1145/2701418

1. INTRODUCTION

Overlay networks are essential to several Internet services. For instance, Spotify [Kreitz and Niemelä 2010], a popular commercial music streaming application, uses an unstructured Gnutella-like [Chasin 2001] Peer-to-Peer (P2P) overlay to offload data distribution from its central music repositories to the client machines; BitTorrent, a popular file-sharing application, uses a P2P Distributed Hash Table (DHT) structure [Wolchok and Halderman 2010] to map file identifiers to swarms of peers sharing those files; and Tor [Dingledine et al. 2004] uses an overlay of relay servers to provide private and secure Internet communication.

This article is based on the work presented in the 1st ACM SIGOPS EuroSys Conference [Johansen et al. 2006] and Johansen's PhD dissertation [2007].
This work was supported in part by the Norwegian Research Council projects 231687/F20 and 174867, the DARPA/IPTO SRS program, the AFRL/Cornell Information Assurance Institute, NSF grant 0430161, DARPA grant FA8750-10-2-0238, grant-of-excellence #120032011 from the Icelandic Research Fund, and funds from Emory University.

Authors' addresses: H. D. Johansen and D. Johansen, Dept. of Computer Science, UiT The Arctic University of Norway, Hansine Hansens veg 18, 9019 Tromsø, Norway; emails: {haavardj, dag}@cs.uit.no; R. van Renesse, Dept. of Computer Science, 433 Gates Hall, Ithaca, NY 14853, USA; email: rvr@cs.cornell.edu; Y. Vigfusson, Dept. of Math and CS, Emory University, 400 Dowman Drive, W415 Atlanta, GA 30322, USA; email: ymir@mathcs.emory.edu.

ACM 0734-2071/2015/05-ART5 $15.00
DOI: http://dx.doi.org/10.1145/2701418

Several papers have discussed the problem of *Byzantine failures* within overlay networks, including those of Douceur [2002], Sit and Morris [2002], Srivatsa and Liu [2004], Bortnikov et al. [2008], and Urdaneta et al. [2011]. A key observation is that errors in the mechanism that maintains membership information can cripple any higher-level effort to provide fault tolerance, particularly if induced systematically by a hostile intruder attacking the system [Singh et al. 2004]. Possible faults or attacks include falsely reporting correct members as crashed, falsely reporting crashed members as live, and biasing the overlay topology so that correct clients unknowingly prefer communicating with the attacker, a so-called *eclipse* attack [Singh et al. 2004].

These security and reliability problems are a major obstacle to deploying new Internet applications based on overlay networks. For example, the DPoll protocol relies on an overlay network to provide decentralized polling for social networks [Guerraoui et al. 2012], but cannot be deployed without a secure overlay substrate. Many applications would benefit from deploying P2P key-value stores, such as Cassandra [Lakshman and Malik 2010] and Dynamo [DeCandia et al. 2007], but such systems cannot survive even simple attacks.

In this article, we describe an overlay network called Fireflies,[1] which combines full membership with a pseudorandom structure to provide a novel and practical tradeoff between tolerance to Byzantine faults and scalability. Fireflies provides its correct members with a membership view that includes all members that have been correct for sufficiently long and excludes all members that have stopped executing for sufficiently long. Fireflies also provides a low-diameter communication graph on the members that guarantees, with high probability, that the subgraph of correct members is connected. This graph is ideally suited for gossiping among the correct members.

Membership protocols that maintain full views have been shunned in the past as building blocks for P2P file-sharing networks and DHT services because the rate of membership events will likely grow linearly with the number of members, possibly leading to unmanageable volumes of network traffic. However, by maintaining full membership views, applications built on top of Fireflies can send messages directly to their destinations. We can thus avoid the complex techniques required for secure and reliable overlay routing [Urdaneta et al. 2011], and the overlay need not be restructured dynamically to runtime metrics like network proximity [Gummadi et al. 2003]. Many applications have relatively static membership and thus maintaining full membership views is both possible and desirable [Lakshman and Malik 2010; DeCandia et al. 2007; Rodrigues and Blake 2004; Kreitz and Niemelä 2010; Gupta et al. 2003].

The current article extends a prior publication by Johansen et al. [2006], providing significantly more details on the mechanics, analysis, and implementation of the protocol. The exposition benefits from practical experience with building applications on Fireflies.

2. BACKGROUND

An overlay network is a virtual packet processing and routing substrate built on top of some existing network infrastructure like the Internet or, recursively, on top of another overlay network. 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 and exist only as part of the overlay state. Overlay networks

[1] Fireflies, the bioluminescent family of winged beetles, model not only the on/off behavior of members, but like Byzantine members they are also known for their aggressive mimicry in order to dupe and devour related species.

are commonly represented as a graph where the vertices are member processes and the edges are communication links.

2.1. System Model

Overlays are dynamic and processes may *join* the overlay becoming *members*, and existing members may permanently *leave*. Each member is either *correct*, *crashed*, or *corrupt*. Correct members faithfully execute the specified overlay protocol, while crashed members do not execute any protocol steps. Corrupt members are not bound by the protocol and might execute arbitrary instructions. For convenience, we refer to members that are either correct or crashed as *honest*, and to members that are correct or corrupt as *live*. Members that are not correct are often called *Byzantine*. Note that crashed members are considered Byzantine, but not corrupt.

Members may switch between being live and crashed, which is commonly referred to as *churn*. Correct members may be unreachable and appear crashed to other members due to transient network outages. Corrupt members can disguise themselves as correct members by executing the protocol, or as crashed members by not executing at all. Generally, correct members cannot determine which members are corrupt unless they reveal themselves as such by sending messages that prove that they are not following the protocol.

Every correct member m has a unique identifier $m.id$, a view $m.view$ containing the identifiers for other members participating in the overlay network, and a set of members believed to be live, $m.live \subseteq m.view$. Assuming m is correct,

—$m'.id \in m.live$ means that m considers that m' was live, at least until recently. The converse,
—$m'.id \notin m.live$, implies that m considers m' to be crashed, at least until recently.

Also, m has a set of neighbors, $m.neighbors$, which is a subset of $m.live$. In this article, we assume that each correct member m can connect to every other correct member in $m.neighbors$. This assumption can be relaxed, but calculated tolerance thresholds have to be adjusted accordingly. The views and set of neighbors of correct members have the following properties with high probability:

Agreement. If a member m is in the view of some other correct member m', then, within bounded time, m will also be in the views of every other correct member.
Validity. The view includes the identifiers of all members that have been correct sufficiently long, and excludes the identifiers of all members that have been crashed sufficiently long (*sufficiently long* will be made more precise in the following).
Connectivity. The set of correct members form a connected subgraph of neighbors.
Scalability. The number of neighbors is logarithmic in the size of the membership, and the diameter of the neighbor graph is logarithmic in the size of the membership.

2.2. Attack Model

We make few assumptions on the capabilities of an attacker. Corrupt members can deviate arbitrarily from the Fireflies protocol. They can collude and share state, and they can also know the state of honest members. In order to create a protocol that can work in the presence of corrupt members, we make the following basic assumptions:

—Corrupt members do not have sufficient computational power to break cryptographic building blocks. In particular, we assume that they cannot forge public key certificates or public key signatures of honest members.

—Trivial Denial-of-Service (DoS) attacks like flooding can be detected and suppressed using techniques such as port randomization, careful resource management, and rate limiting [Badishi et al. 2006].
—Correct members have access to clocks running with a bounded difference to real time.
—Correct members can exchange and process messages within a known bounded time interval.

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

If an attacker is able to acquire control over a large number of identities, an overlay is at risk of being compromised. The forging of multiple identities in order to gain control of a system is often referred to as a *Sybil attack* [Douceur 2002]. Overlays are susceptible to such attacks unless we limit the fraction of corrupt members within the overlay network as a whole, as well as within any subset of the members selected for particular tasks. For instance, in OceanStore [Kubiatowicz et al. 2000], the number of Byzantine members (that is, crashed or corrupt) within each primary replication group must be less than one-third.

Various approaches have been proposed for dealing with Sybil attacks, including packet latency triangulation [Bazzi and Konjevod 2005], credit payment schemes on top of social networks [Viswanath et al. 2012], client puzzles [Juels and Brainard 1999], and physical artifacts like smart cards [Druschel and Rowstron 2001]. Although these systems make it harder for an attacker to accumulate and control a large number of overlay-network identities, they cannot prevent an attacker from joining the overlay.

In this article, we assume some oracle Certificate Authority (CA) service that assigns identities to members so that there is a bounded probability, p_{corrupt}, that any randomly chosen live member is corrupt. This is a stronger condition than a bound on the probability that *any* member is corrupt. The weaker condition is not sufficient as the situation where most honest members are crashed, while most corrupt members remain live is similar to a Sybil attack. Nonetheless, the assumption that among all live members only a fraction is corrupt is reasonable, particularly since we do not limit the fraction of crashed members among all members.

3. MEMBERSHIP MAINTENANCE

Fireflies organizes members into a pseudorandom mesh structure that dictates neighbor selection. To maintain this structure, members monitor one another using an *adaptive crash detection* protocol and issue *accusations* whenever a member is suspected to have crashed. When a member m receives an accusation for some other member m', m waits a time period of length 2Δ before removing m' from $m.live$, where Δ is a probabilistic upper bound on end-to-end latency of notices sent on the overlay. Should m' receive an accusation about itself, then m' has the opportunity to issue a *rebuttal* before the timeout of 2Δ expires, which will invalidate any previous accusations for m'. Fireflies strives to make the set of accusations empty for correct members and nonempty for crashed members.

Members broadcast recent accusations and rebuttals using a secure gossip channel. Correct membership depends on the ability of this channel to deliver messages to correct members within Δ time, even in the presence of corrupt members. In turn, Fireflies depends on the correct membership views to ensure that gossip reaches all correct members with high probability. To navigate the narrative complexity of this circular

$\mathcal{C}_m$ (Certificate)	$\mathcal{N}_m^i$ (Note)	$\mathcal{A}[m]_{m'}^i$ (Accusation)	$\mathcal{T}[m]_{m'}^i$ (Timeout)
public key	$\mathcal{C}_m$	$\mathcal{C}_{m'}$ (accuser)	$\mathcal{C}_{m'}$ (observer)
network address	ith epoch	$\mathcal{N}_m^i$ (accused)	$\mathcal{N}_m^i$ (accused)
	mask		time stamp
CA sign.	$\mathcal{C}_m$ sign.	$\mathcal{C}_{m'}$ sign.	

Fig. 1. Data types.

dependency, the remainder of this section describes the mechanisms and rules governing membership maintenance, assuming the existence of some appropriate broadcast channel. Section 4 describes how Fireflies constructs its gossip mesh.

3.1. Data Structures

Each correct member m maintains three local datasets: *m.notes*, *m.accusations*, and *m.timeouts*. Correct members exchange notes and accusations with their neighbors through gossip so that every member eventually will have the same set. The set of timeouts is kept local to each member. The following data structures are used:

Certificate. Public-key certificates are used to uniquely identify member processes. A certificate $\mathcal{C}_m = (\textit{public key}, \textit{address}, \textit{signature})$ binds m's *public key* to the *network address* where it can be reached. In practice, other fields like start and expiry dates are also included in certificates, but their functions are not required for the description of our protocol and will therefore be omitted here. To be valid, the binding must be signed by the trusted CA oracle. By assumption, honest members never reveal their private keys.

Note. A note $\mathcal{N}_m^i = (\mathcal{C}_m, \textit{epoch}, \textit{mask}, \textit{signature})$ is the ith signal from member m that it is alive, where $\mathcal{C}_m$ is the certificate for m, *epoch* is a monotonically increasing number, and *mask* is a bitmap used to prevent repeated false accusations, which will be described later. The epoch numbers impose a total ordering of the notes from each member such that $\mathcal{N}_m^i > \mathcal{N}_m^j \Leftrightarrow i > j$. In this case, we say that $\mathcal{N}_m^i$ is more recent than $\mathcal{N}_m^j$. When clear from the context, we will use the notation $\mathcal{N}_m$ to denote the most recent note of m observed by some member. The set *m.notes* of a correct member m eventually holds the most recent note for each participating honest member in the overlay.

Accusation. An accusation $\mathcal{A}[m]_{m'}^i = (\mathcal{C}_{m'}, \mathcal{N}_m^i, \textit{signature})$ states that member m' suspects that some other member m has crashed, and $\mathcal{N}_m^i$ is the most recent note for m known to m'. The accusation is signed with the private key of m'. The set *m.accusations* of a correct member m eventually holds entries for each accused member in the overlay.

Timeout. A timeout $\mathcal{T}[m]_{m'}^i = (\mathcal{C}_{m'}, \mathcal{N}_m^i, \textit{time stamp})$ indicates the time when member m' first observed an accusation for note $\mathcal{N}_m^i$ of member m. The set *m.timeouts* of a correct member m contains at most one entry for each member in the overlay. A summary of these data structures is provided in Figure 1.

3.2. Data Validity Rules

Fireflies defines the following set of rules that a correct member follows to determine the validity of each data item it has:

RULE 1 (CORRECT SIGNATURES). *Note $\mathcal{N}_m^i$ or accusation $\mathcal{A}[m']_m^i$ is only valid if it is signed correctly with the private key for $\mathcal{C}_m$, and $\mathcal{C}_m$ is correctly signed by a common trusted CA.*

RULE 2. *Note $\mathcal{N}_m^i$ is only valid if it is the most recent observed note from m.*

RULE 3. *Accusation $\mathcal{A}[m]_{m'}^i$ is only valid if the contained note $\mathcal{N}_m^i$ is valid.*

RULE 4. *Timeout $\mathcal{T}[m]_{m'}^i$ is only valid if there exists a valid accusation $\mathcal{A}[m]_{m'}^i$.*

In addition, members adhere to the following decrees:

RULE 5 (REBUTTALS). *A correct member m, upon receiving a valid accusation $\mathcal{A}[m]_{m'}^i$ for its own note, will immediately create and gossip a new note $\mathcal{N}_m^{i+1}$. This note will eventually invalidate any previous accusations for m at all other correct members.*

Due to the probabilistic upper bound Δ on broadcasting messages in our gossip protocol, as will be discussed in Section 4, a correct member m will receive any valid accusation $\mathcal{A}[m]_{m'}^i$ within Δ time and, following Rule 5, issue a rebuttal $\mathcal{N}_m^{i+1}$. With high probability, this rebuttal will be received by all correct members within at most 2Δ time since $\mathcal{A}[m]_{m'}^i$ was issued. Hence, no correct member m' will have a timeout $\mathcal{T}[m]_{m'}^i$ that is more than 2Δ old. If m had indeed crashed, no rebuttal to $\mathcal{A}[m]_{m'}^i$ would be issued. This gives us the following definition for determining crashes:

Definition 3.1 (*Crashes*). A correct member m considers member m' crashed if, and only if, m has a local timeout $\mathcal{T}[m']_m^i$ that is valid according to Rule 4 and older than 2Δ. Otherwise, m considers m' live.

To attack this protocol, a corrupt member might refrain from issuing accusations for crashed members in an attempt to keep them in views of correct members. Consequently, we must make sure that all members are monitored by at least one correct peer member, a so-called *monitor*. However, there is a network overhead associated with monitoring so we also want to minimize the number of monitors assigned to each member. We also have to prevent corrupt members from increasing network load by submitting frequent accusations about correct members. This is a complicated issue because correct members might also accidentally accuse other correct members due to, for instance, transient link failures. Thus, not every false accusation is from a corrupt member. In the following sections we will describe how Fireflies implements monitoring to resolve these issues.

3.3. Membership Rings

To assign monitoring responsibilities, Fireflies organizes all members in a pseudorandom mesh structure made up of k membership rings. Each such ring is a subgraph of the mesh in which each member m has exactly two other neighbors (assuming there are at least three members in the overlay). More formally, a membership ring r is characterized by the pair $(\mathcal{M}, id)$ where $\mathcal{M}$ represents the set of members and id is a unique ring identifier known to all members. The set of edges connecting the members in r is derived deterministically from $\mathcal{M}$ and id. For this, we impose a total ordering, $\prec_r$, on the members, which is specific to each ring r. The ordering function $\mathcal{H}$ is specified by applying a Secure Hash Algorithm (SHASH) on the concatenation (∥ symbol) of the members' identities and ring identity in the following manner:

$$\mathcal{H}(m, r) = \texttt{SHASH}(m.id \parallel r.id). \tag{1}$$

The SHASH 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 and m', the ordering is defined as

$$m \prec_r m' \Leftrightarrow \mathcal{H}(m, r) < \mathcal{H}(m', r). \tag{2}$$

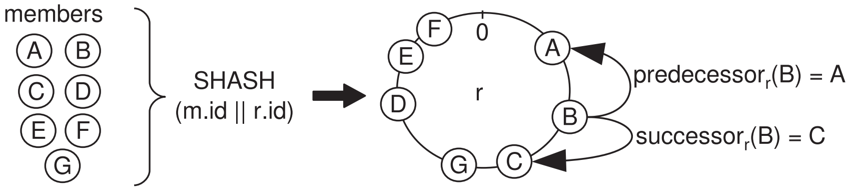


Fig. 2. A Fireflies membership ring.

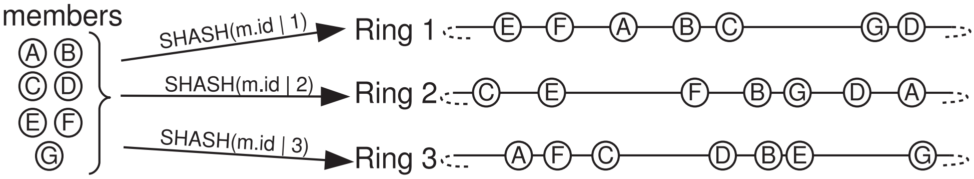


Fig. 3. Fireflies mesh with three rings.

There is an edge in r between all members that are adjacent by this ordering. Also, because $\prec_r$ is not circular, we include an edge between the highest member and the lowest member. More formally, there exists an edge from m to m' if and only if $m \prec_r m'$ and there exists no member m'' such that $m \prec_r m'' \prec_r m'$. If $m \succ_r m'$, then there exists an edge between them if and only if there is no member m'' such that $m'' \succ_r m$ and no member m''' such that $m''' \prec_r m'$. This results in a 2-connected Harary graph [Harary 1962], or a ringlike structure as seen in Figure 2.

In each ring r we define the following relations:

—*successor*$_r(m) = m'$. We say that m' is the *successor* of m in ring r if there exists an edge between m and m' and either $m \prec_r m'$ or there exists no m'' such that $m'' \succ_r m$. Each member has exactly one successor in r.

—*predecessor*$_r(m) = m'$. We say that m' is the *predecessor* of m in ring r if there exists an edge between m and m' and either $m \succ_r m'$ or there exists no m'' such that $m'' \prec_r m$. Each member has exactly one predecessor in r.

—$\mathrm{rank}_r(m, m') = x$. The rank relation adds transitive properties to the successor relationships so that there are exactly x successor edges connecting m and m' in ring r. In this case, we may also say that m' is m's xth successor.

As an example, consider the seven members A through G in Figure 2, each mapped by the secure hashing function to a pseudorandom position in the circular address space of the ring r. Then the successor of B is C since C is the next clockwise member from B. We also have $\mathrm{rank}_r(B, D) = 3$ since there are exactly three successor edges between B and D.

By combining k rings, each with a different ring identifier, each member m will be assigned up to k pseudorandom predecessors and up to k pseudorandom successors. By having each member monitor its successors in this set of rings, we can assign up to k pseudorandom monitors to each member. The number of rings, k, can be adjusted to trade attack resilience with network overhead. As an example, consider the seven members A through G in Figure 3, securely hashed into three rings. The successors of B, one for each ring, are $\{C, G, E\}$, and its predecessors are $\{A, F, D\}$.

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 a uniform and independent probability p_{corrupt} of being corrupt. Hence, the probability on the number of corrupt monitors of m has a *binomial distribution*.

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

$$P[X = t \mid k] = \binom{k}{t} p_{\text{corrupt}}{}^{t}(1 - p_{\text{corrupt}})^{k-t}, \quad t = 0, 1, \ldots, k. \tag{3}$$

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

$$P[X = k \mid k] = \binom{k}{k} p_{\text{corrupt}}{}^{k}(1 - p_{\text{corrupt}})^{k-k} = p_{\text{corrupt}}{}^{k}. \tag{4}$$

For example, if $k = 7$ rings are used and $p_{\text{corrupt}} = 0.10$, 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 corrupt monitors assigned to it. Indeed, the probability of this happening increases with the number of rings:

$$P[X \geq 1 \mid k] = 1 - P[X = 0 \mid k] = 1 - (1 - p_{\text{corrupt}})^{k}. \tag{5}$$

In the preceding example with $k = 7$ and $p_{\text{corrupt}} = 0.10$, the probability of m having a corrupt monitor becomes 0.523.

3.4. Disabling Corrupt Monitors

Any false accusations of correct members will be rebutted and therefore does not alter the views of correct members. Nevertheless, a corrupt member can repeatedly accuse those members that it is assigned to monitor in order to increase system load and execute a DoS attack. To deal with this, Fireflies allows each member m to disable rings with misbehaving predecessors using the mask field in its note. This must, however, be done in such a manner that m cannot, intentionally or unintentionally, disable all its correct monitors, or m could end up having only corrupt monitors.

To solve this, we impose an upper limit on the number of disabled rings. Let $k = 2t+1$ where t is the *maximum* number of corrupt monitors that some member m can tolerate. Next, allow m to disable monitoring in t rings. Then, m can disable all of its corrupt monitors. At the same time, even if m disables t correct monitors, at least one correct monitor remains. For instance, given seven rings, m can tolerate having up to three corrupt monitors. After disabling three monitors, m still has four active monitors, whereof at least one is correct. This gives us the following additional rules:

RULE 6. *A note $\mathcal{N}_m^i$ is only valid if the contained mask bitmap is of length $k = 2t + 1$ and at most t of the bits are disabled.*

RULE 7. *An accusation $\mathcal{A}[m]_{m'}^i$ is only valid in ring $r = (\mathcal{M}, id)$ if $m \in \mathcal{M}$ and the bit corresponding to the identifier id in the mask field of $\mathcal{N}_m^i$ is enabled.*

In general, given $k = 2t+1$ rings, a natural question is to estimate the likelihood that m winds up with a majority of corrupt monitors. We assume that all of m's neighbors have been assigned independently and that each is corrupt with probability p_{corrupt}. Let $X_{mi} \in \{0, 1\}$ for $i = 1, 2, \ldots, k$ be random indicator variables such that $X_{mi} = 1$ if neighbor i of m is corrupt, and $X_{mi} = 0$ otherwise. Define $X_m = \sum_{i=1}^{k} X_{mi}$ to count the number of corrupt neighbors of m.

THEOREM 3.2. *Let $0 < p_{corrupt} < 1/2$. The probability that m has a majority of corrupt monitors is $P[X_m \geq t+1] < \exp(-O(k))$. If $k = \Omega(\log N)$, then the expected number of nodes that have a majority of corrupt neighbors is $O(1)$.*

PROOF. Let $\mu_m = E[X_m] = k \times p_{\text{corrupt}}$. We will use the following version of the Chernoff bound on the upper tail of sums of independent random variables:

$$P\left[X_m \geq (1+\delta)\mu_m\right] \leq \exp\left(-\frac{\delta^2 \times \mu_m}{2+\delta}\right) \text{ for } \delta > 0. \tag{6}$$

Note that $\frac{k}{2} < t+1$, and so that $P[X_m \geq t+1] < P[X_m \geq \frac{k}{2}]$. By setting

$$\delta = \frac{1}{2p_{\text{corrupt}}} - 1,$$

we have $k/2 = (1+\delta)\mu_m = (1+\delta)k \times p_{\text{corrupt}}$.

Note that the assumption $0 < p_{\text{corrupt}} < 1/2$ in our theorem implies $\delta > 0$ as required by the Chernoff bound. We can therefore substitute for δ and μ_m in (6) to obtain

$$\begin{aligned} P[X_m \geq t+1] < P[X_m > k/2] \ &\leq \ \exp\left(-\frac{\left(\frac{1}{2p_{\text{corrupt}}} - 1\right)^2 k \times p_{\text{corrupt}}}{2 + \frac{1}{2p_{\text{corrupt}}} - 1}\right) \\ &= \ \exp\left(-k \times \frac{(2p_{\text{corrupt}} - 1)^2}{4p_{\text{corrupt}} + 2}\right). \end{aligned}$$

Let us call a node that has a majority of corrupt neighbors an *unfortunate* node. The preceding bound shows that the probability of a node being unfortunate is exponentially small in k, the number of rings. We proceed to bound the expected number of unfortunate nodes Z as we vary k. This calculation will indicate how our system should be configured at large scales. Let $Z_m \in \{0, 1\}$ for $m = 1, \ldots, N$ be a random variable denoting whether node m is unfortunate ($Z_m = 1$) or not. Then $Z = \sum_{m=1}^{N} Z_m$. It follows that

$$E[Z] \ = \sum_{m=1}^{N} E[Z_m] = \sum_{m=1}^{N} P[X_m > k/2] < N \times \exp\left(-k \times \frac{(2p_{\text{corrupt}} - 1)^2}{4p_{\text{corrupt}} + 2}\right).$$

Thus, if we impose the condition that

$$E[Z] < c, \tag{7}$$

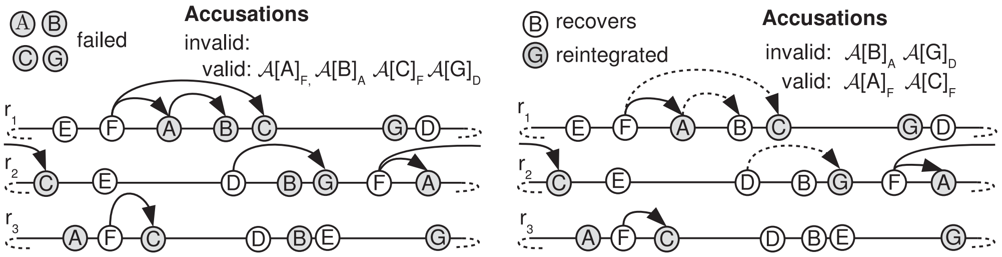
for some configuration constant $c > 0$, we obtain that

$$\begin{aligned} \exp\left(-k\frac{(2p_{corrupt} - 1)^2}{4p_{corrupt} + 2}\right) &< \frac{c}{N}, \\ k\frac{(2p_{corrupt} - 1)^2}{4p_{corrupt} + 2} &> \ln\left(\frac{N}{c}\right), \end{aligned}$$

and thus

$$k > \ln\left(\frac{N}{c}\right)\frac{4p_{\text{corrupt}} + 2}{(2p_{\text{corrupt}} - 1)^2} = \Omega(\log N). \tag{8}$$

In other words, the expected number of unfortunate nodes is constant with c when the number of rings k is logarithmic in N.


A B C G failed
Accusations
invalid:
valid: $\mathcal{A}$[A]$_F$, $\mathcal{A}$[B]$_A$ $\mathcal{A}$[C]$_F$ $\mathcal{A}$[G]$_D$
r1
r2
r3
E F A B C G D
C E D B G F A
A F C D B E G
B recovers
G reintegrated
Accusations
invalid: $\mathcal{A}$[B]$_A$ $\mathcal{A}$[G]$_D$
valid: $\mathcal{A}$[A]$_F$ $\mathcal{A}$[C]$_F$

(a) Valid accusations with crashed members

(b) Recovery may invalidate accusations

Fig. 4. Example of valid and invalid accusations.

The case where no node is unfortunate is of particular interest. Based on the preceding calculations, we can obtain the following lower bound on k. Since

$$P[Z=0] \;=\; \prod_{i=1}^{N}(1-P[Z_m>k/2]) > \left(1-\exp\left(-k\times\frac{(2p_{\text{corrupt}}-1)^2}{4p_{\text{corrupt}}+2}\right)\right)^N,$$

we obtain that $P[Z=0] > \varepsilon$ when

$$1-\exp\left(-k\times\frac{(2p_{\text{corrupt}}-1)^2}{4p_{\text{corrupt}}+2}\right) > \varepsilon^{1/N}$$

or

$$k \geq \frac{4p_{\text{corrupt}}+2}{(2p_{\text{corrupt}}-1)^2}\ln\left(\frac{1}{1-\varepsilon^{1/N}}\right) \geq \frac{4p_{\text{corrupt}}+2}{(2p_{\text{corrupt}}-1)^2}\varepsilon^{1/N}. \quad \square \tag{9}$$

3.5. Skipping Crashed Members

As we have not bounded the probability that a member is crashed, all predecessors of a member may be crashed with nonnegligible probability. In order to allow such members to be accused in case they crash, a member must be able not only to accuse its immediate successor in each ring, but must also be able to make accusations skipping over potentially crashed successors.

We therefore allow members not only to accuse their immediate successors, but also the lowest ranked live member in each ring. This gives us the following rule:

Rule 8. *An accusation $\mathcal{A}[m']^i_m$ is only valid in ring r if there is no other live member m'' such that* $\text{rank}_r(m', m'') > \text{rank}_r(m', m)$.

In Figure 4(a), we illustrate how one of the members observes a group with seven members, A through G, using $k = 3$ rings. For simplicity, we ignore ring deactivation. Given that the four members A, B, C, and G have crashed so that they cannot issue rebuttals, then the accusations $\mathcal{A}[A]_F$, $\mathcal{A}[B]_A$, $\mathcal{A}[C]_F$, and $\mathcal{A}[G]_D$, shown as solid arrows in the figure, will all eventually become valid for the following reasons:

—$\mathcal{A}[A]_F$ is valid because F is the immediate predecessor of A in rings 1 and 2.
—$\mathcal{A}[B]_A$ is valid because A is the immediate predecessor of B in ring 1. Note that accused members are not excluded from issuing accusations, so $\mathcal{A}[B]_A$ is valid even though A is considered crashed.
—$\mathcal{A}[G]_D$ is valid when the timer $\mathcal{T}[B]_m$ for B expires at member m, making D the highest ranked live monitor for G in ring 2.
—$\mathcal{A}[C]_F$ is valid since F is the immediate predecessor of C in ring 3. Also, when T_A expires then F is the highest ranked live successor of C in ring 2, and when both T_A and T_B expires it is similarly the highest ranked successor in ring 1.

One complication from Rule 8 is that the reception of a note $\mathcal{N}_{m'}^{j}$ might not only invalidate any previous accusations $\mathcal{A}[m']_m^i$ where $i < j$. Since the rebuttal implies that m' is no longer considered crashed, then any previous accusations $\mathcal{A}[m'']_m$ valid in ring r are invalidated in that ring if $\text{rank}_r(m'', m) < \text{rank}_r(m'', m')$. If $\mathcal{A}[m'']_m$ is no longer valid in any rings, then it must be discarded.

To illustrate this, consider the situation in Figure 4(b) where member B recovers and issues a rebuttal. Then the following invalidation occurs, shown as dashed arrows in the figure:

—$\mathcal{A}[B]_A$ becomes invalid because of the new note from B.
—$\mathcal{A}[G]_D$ becomes invalid since B is now considered live and is higher ranked than D in ring 2 (i.e., $\text{rank}_2(G, D) < \text{rank}_2(G, B)$) and the accusation is not valid in any other ring. Consequently, there are no valid accusations for G, and it is considered live until it is correctly accused by either B or E.
—$\mathcal{A}[C]_F$ is valid as it is still valid in rings 2 and 3. However, the accusation becomes invalid in ring 1 due to B recovering.

This process of rechecking accusations must be conducted whenever a member m' transitions from a crashed to a live state or is added to the overlay. The invariant of Rule 8 limits the set of potential invalid accusations $\mathcal{A}[m'']_m$ to any successors m'' of m' up to and including the first live one in each ring r. Furthermore, only those accusations where $\text{rank}_r(m', m'') > \text{rank}_r(m, m'')$ need to be considered. This effect cascades if the invalidation of $\mathcal{A}[m'']_m$ results in m'' transitions from a crashed to a live state. In this case, the process of rechecking accusations must be conducted in the context of m'' as well.

Allowing members only to skip over crashed members rather than accused members limits the rate at which corrupt members can make false accusations to $k/2\Delta$. However, it will also make data propagation between correct members more complicated and less efficient as agreement on which accusations are valid will depend upon the 2Δ timeout. In practice, gossip-based dissemination schemes as used in Fireflies will take care of this complication, but can result in accusations being sent multiple times between members.

3.6. Protocol Summary

We can summarize the Fireflies overlay maintenance protocol in the following steps:

—**Member m suspects member m' of having crashed.** On each ring, m monitors the lowest ranked live successor m' for which m can issue a valid accusation according to Rule 8 and Rule 7. Should m suspect that m' has crashed, then it creates and signs an accusation $\mathcal{A}[m']_m^i$, where $\mathcal{N}_{m'}^i$ is the most recent note for m' known to m, and subsequently gossips this to the other members.
—**Member m receives a note $\mathcal{N}_{m'}^i$ for member m'.** Member m first checks that the received note is correctly signed according to Rule 1 and has a correct mask according to Rule 6. If not, the note is discarded. If m does not already have a note for m', then m adds $\mathcal{N}_{m'}^i$ to its notes set and considers m' a new live member. If m has a note $\mathcal{N}_{m'}^j$ where $j > i$, then the received note is discarded since it is obsolete according to Rule 2. If $j = i$, then m already knows about this note and needs to do nothing. If $j < i$, then the received note is more recent. Member m then updates its state with the new note $\mathcal{N}_{m'}^i$ and discards the old note. Any previous accusations for m' will then become invalid, due to Rule 3, and any previous timeouts $\mathcal{T}[m']_m^j$ become invalid due to Rule 4. If m' was previously considered crashed, then m may have accusations

for other members that now are invalid according to Rule 8. These accusations are removed as well.

- **Member m receives an accusation $\mathcal{A}[m']^i_{m''}$ for m'.** Member m first checks that the received accusation is correctly signed according to Rule 1 and has a valid and recent note according to Rule 3. If not, the accusation is discarded. If $m' = m$, then m replaces its note with a new one to act as a rebuttal according to Rule 5, which is subsequently gossiped to the other members. If $m \neq m'$ and m already has an accusation for m' on the same rings as the new accusation, then m replaces its accusation only if the new one is from a higher ranked accuser. Otherwise, m accepts the accusation and sets the removal timer $\mathcal{T}[m']^i_m$ if not set earlier.
- **The removal timer for $\mathcal{T}[m']^i_m$ expires.** Member m considers m' to have crashed.

4. DATA DIFFUSION

Membership maintenance in Fireflies requires a broadcast primitive where all correct members can be reached within Δ time. Corrupt members might therefore attempt to attack the protocol by slowing down dissemination or neglect forwarding data altogether. To fight such attacks, Fireflies uses a gossip-based broadcast service. Gossip 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 [Kermarrec et al. 2003]. In our particular situation, we have to concern ourselves with corrupt members.

One key concern with gossip is that corrupt members could gang up on a small set of correct members, overwhelming them with gossip load [Badishi et al. 2006]. Fortunately, Kermarrec et al. [2003] have shown that it is possible to build effective gossip protocols if each member only has a small set of uniformly chosen members with which it gossips. We therefore restrict who can gossip with whom using the same technique used in Section 3.3 to assign monitors, except that we here use a possibly different number of rings. Gossip neighbors are thus chosen from a pseudorandom low-diameter mesh that connects all correct members. To maintain this mesh, each member connects to its first live successor in each ring, and allows a connection from its first live predecessor. More precisely, Fireflies dictates the following rules:

RULE 9. *Let $m.live \subseteq m.view$ be the set of all members m considers live. For each ring r, member m maintains a secure mutually authenticated gossip connection with*

$$m' = \arg\min_{x \in m.live} \mathrm{rank}_r(m, x).$$

Correct members will relentlessly reconnect gossip connections that terminate from errors or timeouts. Changes in the gossip mesh occur only as a consequence of the following events in membership:

- Member m receives a note $\mathcal{N}^i_{m''}$ for some member m'' where $\mathrm{rank}_r(m, m'') < \mathrm{rank}_r(m, m')$ and either $\mathcal{C}_{m''}$ was previously unknown to m or there exists a timeout $\mathcal{T}[m'']^j_m > 2\Delta$ with $i > j$. In this case Rule 4 dictates that $\mathcal{T}[m'']^j_m$ is invalid and m'' transitions from failed to live in $m.view$, and thus $m'' = \arg\min_{x \in m.live} \mathrm{rank}_r(m, x)$.
- A valid timeout $\mathcal{T}[m']_m$ becomes older than 2Δ. In this case, m' transitions from live to failed in $m.view$.

If at any point in time member m should determine a better gossip partner m'' for ring r according to Rule 9, m terminates the existing connection with m' and contacts m'' instead.

When receiving a request for gossip from some member m', member m will use its local view to check if m' should be allowed to connect using the following rule:

RULE 10. *For each ring r, member m accepts gossip only from m′ if*

$$m = \underset{x \in m.live}{\arg\min}\, \mathrm{rank}_r(m', x).$$

Note that Rule 5 dictates that a correct member m always consider itself live and so have $m \in m.live$. Also, note that Rule 10 does not require the connecting member m' to be in $m.live$, which enables recovering members to reintegrate themselves into the gossip mesh. Given that $m'' = \arg\min_{x \in m.live} \mathrm{rank}_r(m', x)$ for some member $m'' \neq m$, then any connection attempt from m' will be rejected by m. To help m' integrate itself into the gossip mesh in such cases, m will, as part of the gossip handshake protocol, redirect m' to m'' by transmitting the note $\mathcal{N}_{m''}$ before terminating the connection. In the case that m'' had indeed failed, but just had not timed out in $m.live$, then m' will wait for a period $>2\Delta$ before retrying connecting to m. Newly joining and recovering members should gossip with at least $t + 1$ different randomly chosen members before they can be reasonably certain that they are integrated into the true membership, as opposed to a fake membership created by corrupt members [Singh et al. 2004].

4.1. Ensuring Connectivity

Allowing members to gossip with only a limited number of other members enables Fireflies to reduce the opportunity for corrupt members to attack. For such a scheme to work, however, the number of gossip partners for each member must still be large enough to form a connected graph of correct nodes.

The classic result of Erdös and Rényi [1960] shows that in a random graph of N nodes, if the independent probability of each pair of nodes being connected is at least $p_N = (\log N + o(1))/N$, then the graph will almost surely be connected.

The number of correct members, n, is expected to be at least $(1 - p_{\text{corrupt}}) \times N$, where p_{corrupt} is the configured upper bound on the probability that a live member is corrupt, and N is the total of all members (correct, crashed, and corrupt). If each member has k neighbors, then the probability that one member is connected to another is $1 - (1 - 1/N)^k \approx 1 - \exp(-k/N) \approx k/N$. Thus, $p_n \approx 2k/N$. In order for the correct members to be connected with probability ε, we obtain

$$k \geq \frac{N}{2n} \times \left(\log \frac{-n}{\log \varepsilon} + o(1) \right). \qquad (10)$$

4.2. Timeout Value Δ

Next, we determine the resulting Δ: the time needed to disseminate a message in a random graph. To better preserve resources, instead of each updating all of its k outbound neighbors in every round, we instead select 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 every neighbor were updated in each round. If d_n is the diameter of the graph of correct members, then the expected length of time to disseminate an update reliably among the correct members is $\Delta = k \times d_n$.

An asymptotic value for the diameter of the resulting graph d_n can be determined. The result of Chung and Lu [2001] shows that if $np_n \to \infty$, which holds true in our case, then the expected diameter of our graph is given by

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

Unfortunately, this expression does not provide the constants needed to tune the mesh. In order to find suitable constants, we ran simulation experiments with N

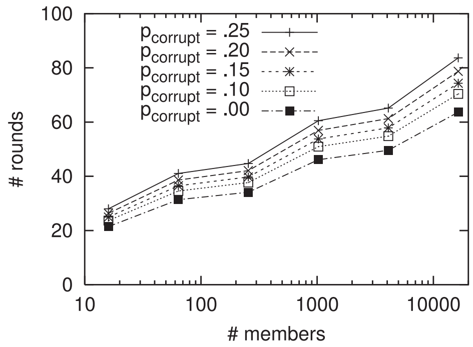


Fig. 5. Number of rounds required to disseminate an update.

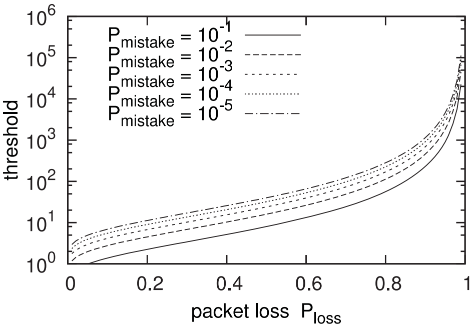


Fig. 6. Crash detection threshold τ as a function of packet loss rate.

ranging from 2^4 to 2^{14} for varying values of p_{corrupt} and with k chosen as earlier (ignoring the $O(1)$ constant 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 5 we report the maximum number of required gossip rounds that we observed for each N and p_{corrupt} with $\varphi = 0.99999$. Rings are added as the number of members increase. This boosts the connectivity of the mesh, which can be seen as intervals of constant slopes in the figure.

5. ADAPTIVE CRASH DETECTION

To detect crashes, members monitor one another by sending probe messages. Each probe involves a member m sending a ping message to its neighbor m' at regular intervals. If m' is correct, it returns a pong message.

A probe is only successful if both the ping and the pong messages are received. In the time period between the ping and the pong message, we say that a probe is *pending*. After some period of time, a pending probe will timeout and the member being probed will be considered to have crashed. Because the Internet implements best-effort protocols, messages can be lost. Therefore, if a probe fails, m should attempt to resend a ping probe. Member m concludes that m' has crashed after τ consecutive probes have failed.

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

5.1. Setting the Threshold τ

Bolot [1993] shows that the loss of probe packets is essentially random when the probe traffic consumes less than 10% of the available network bandwidth. Also, Barford and Sommers [2004] 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, S, reflected in the measured packet-loss rate. A successful probe requires that both the ping and the pong messages are delivered. Hence, the packet-loss probability rate λ, and the probability of a successful probe S are related by $S = (1 - \lambda)^2$.

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

$$P[X = x] = (1 - S)^{x-1}S, \quad x = 1, 2, \ldots .$$

If m repeats a probe τ times and m' is live, the probability that at least one probe succeeds is given by

$$P[X \leq \tau] = \sum_{x=1}^{\tau}(1 - S)^{x-1}S = 1 - (1 - S)^{\tau}.$$

Hence, if after τ failed probes m decides that m' has crashed, the probability that m is wrong because all probes failed due to packet loss is given by

$$P_{\text{mistake}} = 1 - P[X \leq \tau] = (1 - S)^{\tau} = (2\lambda - \lambda^2)^{\tau}. \tag{11}$$

Thus, if m wants to establish with certainty P_{mistake} that m' has crashed, then the number of consecutive probes it must submit is given by

$$\tau = \frac{\log(P_{\text{mistake}})}{\log(2\lambda - \lambda^2)}. \tag{12}$$

The threshold τ increases exponentially with λ. As such, we cannot effectively determine a crash with high accuracy when packet loss is high as illustrated in Figure 6.

5.2. Rounding Error

Equation (12) may produce fractional output values. For instance, if $P_{\text{mistake}} = 10^{-4}$ and $\lambda = 0.10$, then $\tau = 5.546$. Clearly, m cannot probe m' 5.546 times and must choose either 5 or 6. In either case, a rounding error is introduced. Because λ is determined by the packet-loss rate of the underlying network, it cannot absorb this error. Hence, the error must be absorbed by P_{mistake}. Say that m chooses τ to be 5. For this to occur, Equation (11) gives us $P_{\text{mistake}} = 2.47 \times 10^{-4}$. In other words, even though m 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 had chosen τ to be 6, the observed P_{mistake} would be 4.70×10^{-5}, which is 2.1 times lower.

5.3. Estimating Packet-Loss Rate

The previous calculations rely on us knowing the packet-loss rate. For this, we estimate S, the probability of a probe succeeding, by measuring the number of probes that m sends before it receives a response from m'. For negative binomial experiments that produce a geometric distribution, the average number of trials required before a success is given by $E[X] = \frac{1}{S}$. By substituting the expectation into Equation (12), we obtain

$$\tau = \frac{\log(P_{\text{mistake}})}{\log\left(1 - \frac{1}{E(X)}\right)}. \tag{13}$$

The value for $E[X]$ can be estimated by m by recording the difference between ping messages sent and pong replies received. For instance, if m sends six pings to m', but receives a pong reply from m' only for the last ping, then m concludes that 5/6 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 for round i, x is the number of pings sent before a pong is received and $0 \leq \alpha \leq 1$ is the smoothing factor, then

$$E_{i+1}[X] = \alpha E_i[X] + (1 - \alpha)x. \tag{14}$$

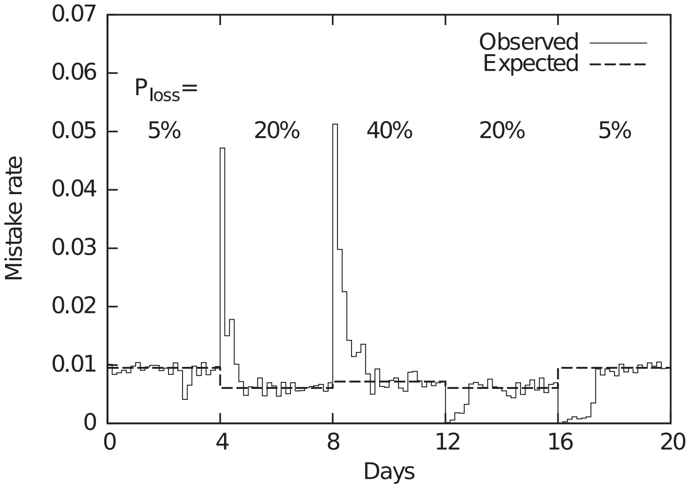


Fig. 7. Adapting the timeout threshold τ to packet-loss rate.

To measure the effectiveness of our adaptive pinging protocol, we constructed a simulation where a member m monitors some other member, m'. We currently do not have an automated mechanism determining a good value for the smoothing factor α. Instead, we found through trial and error that setting $\alpha = 0.99995$ gave us a good balance between smoothing and responsiveness. The pinging interval was set to 1s. Packets were lost at random and both members were correct during the course of the experiment. Figure 7 shows the observed rate at which m made estimation mistakes when the packet-loss rate, λ, 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.

The extreme values for the threshold need to be considered. If packet loss is very low, the τ threshold may be set unrealistically low. With no packet loss ($P_{\text{mistake}} = 0$), 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 also impose a maximum threshold τ_{max} on the timeout threshold. Algorithm 1 shows the complete probing protocol.

5.4. Pinging Attacks

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

Corrupt members can, however, generate a modest amount of overhead on the system by not responding to ping messages from correct members, and then rebutting the ensuing accusations. At low frequency, such nuisance attacks are indistinguishable from transient network outages and packet loss, which are handled by our pinging protocol and accusation-rebuttal scheme. With higher frequency, such behavior is easily identifiable since the high frequency of rebuttals will be visible to all members. Correct members are not expected to send more than t rebuttals in a short time span. The CA can remove such members simply by revoking their public key certificates.

ALGORITHM 1: Adaptive Pinging Protocol to assess packet-loss rate

```
on time to ping m in ring r
    // calculate threshold
    $\tau = \log(P_{\mathrm{mistake}}) / \log(1 - 1/m.avgLoss)$;
    if $m.nPing > \max(\tau, \tau_{min})$ then
        $m.accusations.add($ new Accusation$(m.note, r, self.id))$
    else
        send ($m$, new Ping($self.id$));
        $m.nPing$++
    end

on receive pong from m
    $m.avgLoss = \alpha \times m.avgLoss + (1 - \alpha) \times m.nPing$;
    $m.nPing = 0$
```

6. IMPLEMENTATION

Fireflies is implemented using a combination of Python and C++ extension modules on top of the Twisted event-based networking framework.[2] The latest version of Fireflies is publicly available on SourceForge.[3] We will describe several important issues specific to the current implementation of Fireflies, including several protocol optimizations.

6.1. Certificates and Bootstrapping

Fireflies uses the OpenSSL library and tools [Cox et al. 2011] for all cryptographic operations and can be configured to use all its key and hashing variants. By default, Fireflies uses the 224 bits NIST-recommended P-224 [Locke and Gallagher 2009] elliptic curves for signatures and authentication. The use of elliptic curve cryptography is beneficial to Fireflies due to its smaller signature length compared to RSA and DSA.

To initialize a new Fireflies group, the CA must first create an X.509 compliant [Housley et al. 2002] self-signed *group certificate*. We impose few restrictions on such certificates, and established best-practices for generating and managing them can be used. However, we do require that group certificates include the number of membership rings and the number of gossip rings that are to be used. Ideally, these values would be stored in an X.509v3 extension field, but due to a bug in a third-party library this was not possible and so we currently encode these values in the X.509 subject organization field.

The group certificate is self-signed with the private key of the CA and is made available so that all potential members may download it. It is the responsibility of each group member to check the validity of downloaded group certificates. We assume that the CA, correct members, and crashed members never reveal their private keys. Any member compromised by an attacker, either hacked or manipulated by a malicious insider, is considered corrupt.

To join a Fireflies group, a new member *m* first generates an X.509 certificate request containing its public key, its network address, and subject information. The request file is then sent to the CA for signing. Similar to group certificates, we impose few restrictions on the content of member certificates. Established best-practices for certificate generation, verification, and management should be followed. We do, however, impose that *m*'s network address and port number is stored in the X.509 subject

[2] http://twistedmatrix.com/.

[3] http://fireflies.sf.net/.

localityName field of $\mathcal{C}_m$. For each certificate, the CA also generates a random member identity string, which it embeds in the *subjectKeyIdentifier* extension field. This deviates somewhat from the default usage of the X.509 *subjectKeyIdentifier* field, which is typically set to the hash of the contained public key. However, the default scheme cannot be trusted since it gives members some freedom in choosing their identity and subsequently their position in the overlay network. The length of the identity is set to match the strength of the underlying hashing function used in the X.509 certificates. Currently, we use SHA-256 for hashing and so the generated identities are 32 bytes (B) long.

When member m has obtained a valid member certificate $\mathcal{C}_m$, it can start gossiping with other group members. The use of X.509 certificates and Secure Socket Layer (SSL) for gossip prevents man-in-the-middle attacks. It is therefore sufficient that m gossips with a single correct member to become integrated in the correct membership, rather than a false one controlled by corrupt members. To ensure this, m receives a list of *initial contacts* from the CA as part of acquiring $\mathcal{C}_m$. This list must either include the member certificate of a trusted boot node, or at least $t+1$ member certificates, randomly selected from the set of all live members. Only after gossiping with all initial contacts will m consider itself fully integrated in the membership.

6.2. Data Structure Optimization

Public keys and member certificates are large objects when compared to hashes and signatures. All certificates created by the CA contain a 32B member identity that uniquely identifies the certificate and its embedded public key. We have ignored this identity earlier in this article, but in practice we can improve communication efficiency by replacing the larger certificates in notes and accusations with this smaller member identity.

A further reduction on the size of accusations can also be accomplished by removing the notes from the accusations altogether. To identify the note of the member who is accused, it is sufficient for an accusation to contain only the *subjectKeyIdentifier* from the accused certificate and the epoch number of the accused's note. Using the default configuration of 224-bits ECC certificates, SHA-256 for hashing, and 32B member identifier, we obtain public-key certificates of about 364B, notes of 108B, and accusations of 136B.

One possible complication of this optimization is that accusations and notes are not self-contained. Consequently, the validity of an accusation cannot be established without having previously received the accused note or the accuser's certificate. Similarly, a member cannot ascertain the validity of a note without having previously received the corresponding member certificate. Buffering of unverifiable data structures until the needed data is received is problematic since an attacker can easily target such a mechanism by gossiping notes and accusations containing false identities to trigger overflows. Instead, correct members should simply discard any unverifiable or invalid data they receive. The underlying gossip scheme used in Fireflies will ensure that the required data to verify received notes and accusations is available, as described in the following.

6.3. Communication

Since ICMP is disabled in many networks, we currently use UDP for the pinging mechanism. Members gossip using SSL connections over TCP. Each member m connects to the first live successor m' in each gossip ring according to Rule 9. Using their X.509 certificates, members establish mutually authenticated connections using the SSL protocol. Although Fireflies does not require network-level encryption to ensure correct behavior, the SSL handshake includes a secure exchange of X.509 certificates between connecting end points. This enables members to identify the other member in their views and decide whether to accept or reject an inbound connection based on the restrictions imposed by the gossip rings as dictated by Rule 10. If the connection at some point terminates or times out, then m, after an exponential back-off delay, will try to reestablish that connection until m' is considered crashed by the Fireflies protocol.

One complication is that even when m and m' are both correct, they may have different views. In particular, m' may know a better gossip neighbor m'' for m that is not in m's current view. If such is the case, m' sends note $\mathcal{N}_{m''}$ to m. Should m have valid accusations for m'', then it returns those to m' and terminates the attempt to gossip. If no such accusation exists, then m was unaware of m''. In that case m adds m'' to its view and tries to gossip with m'' instead.

Having members m and m' communicate all certificates, notes, and accusations back and forth is inefficient as most of the membership data held by m and m' are likely to be the same. Fortunately, there exist protocols that reconcile sets of information by only exchanging volume of information that is on the order of the size of the difference between the sets. In particular, our current implementation of Fireflies uses the *partitioned set reconciliation* protocol suggested by Minsky and Trachtenberg [2002]. To reduce network and CPU overhead, we have modified the protocol so that it will reconcile the set of object hashes instead of the data itself. The data is transferred in a separate stage after the reconciliation is completed. Also, to ensure received data structures are verifiable, as described in the previous section, we reconcile certificates before notes, and notes before accusations.

6.4. Calculating the Required Number of Rings

Using the bounds on k from Equation (9), we can calculate the required number of membership rings given p_{corrupt}, N, and ε. Due to the use of the Chernoff bound, however, these formulas give significantly higher estimates for k than is strictly necessary in practice. Although an overestimate improves resilience to corrupt members, it also increases network overhead.

Instead, we wish to compute the exact probability ε that no members will be unfortunate. Recall that we then need to find the minimal k such that

$$P[Z = 0] > \varepsilon. \tag{15}$$

Although (15) is difficult to solve symbolically for k, it can easily be computed, as shown in Algorithm 2, using the binomial cumulative distribution $\text{cdf}(x; n, p) = \sum_{i=0}^{x} \binom{n}{i} p^i (1-p)^{n-i}$ of any statistical software package. This follows from

$$\varepsilon < P[Z = 0] = (1 - P[X_m \geq t + 1])^N = P[X_m \leq t]^N = \text{cdf}(t; 2t + 1, p_{\text{corrupt}})^N.$$

Figure 8 shows the value of k for various N and p_{corrupt} using $\varepsilon = 0.99$, computed using Algorithm 2. It is evident that k grows logarithmically with N, as predicted by Theorem 3.2.

7. EVALUATION

To evaluate Fireflies, we ran experiments in both simulated environments and on PlanetLab. We refer to each participant in the protocol as a Fireflies *agent*. To emulate corrupt behavior in our system, we implemented two types of 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 crashed at any opportunity. The attacker will only create accusations that are valid in accordance to its view of the membership, since invalid accusations are simply ignored and might be forwarded to the CA as proof

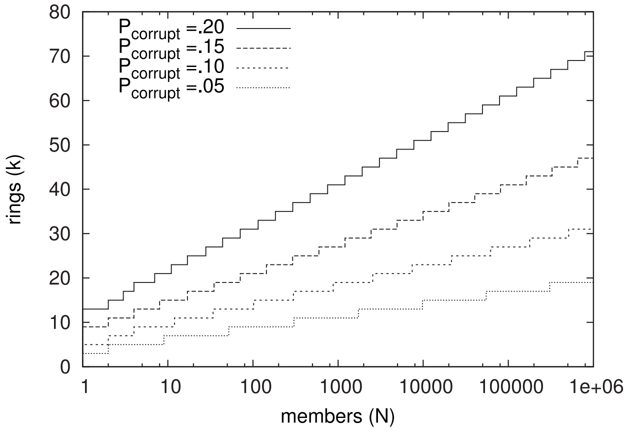


Fig. 8. Required number of rings for different parameter values.

ALGORITHM 2: Computing the required number of rings based on ε and p_{corrupt}

```
$t = 1$;
while $\varepsilon > \text{cdf}\ (t; 2t+1, p_{corrupt})^N$ do
    $t = t + 1$ ;
end
return $k = 2t + 1$;
```

that the attacker is not following the protocol. The attacker will also refrain from forwarding notes in an attempt to prevent correct members from rebutting false accusations made by either the attacker or by other members.

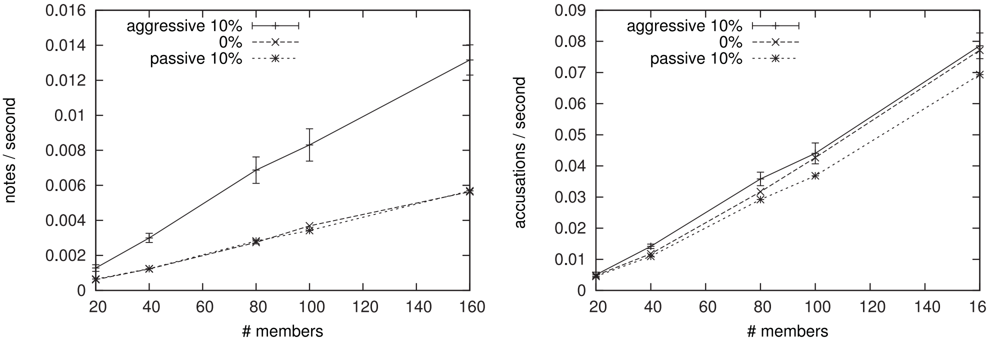
—A *passive attack,* where the goal of the attacker is to keep crashed members in the views of correct members. For this, the attacker never accuses members, and does not forward accusations of crashed members.

7.1. Overhead of Membership Maintenance

To produce repeatable experiments in a controlled environment, we constructed a simulated network environment in which Fireflies agents could run. The simulated Fireflies agents share most of the code base with their networked variants, but bypasses the Transmission Control Protocol (TCP) stack, the User Datagram Protocol (UDP) stack, the marshalling routines, and cryptographic operations for efficiency reasons. We also avoid copying accusation, note, and certificate structures by passing memory location pointers.

By using essentially the same code as in the networked variant of Fireflies, we get a detailed picture of the protocol's behavior. Using similar versions for simulation and experimental evaluation also simplifies debugging as errors can be reproduced in a controlled simulated environment. Unfortunately, the level of detail that we simulate limits our ability to scale up the experiments. For instance, on a single core of a 2.6GHz Intel E5-2670 processor, a simulation of 200 members runs close to real time. The scalability limits of our simulator do *not* indicate scalability limits of the Fireflies protocol since we expect the protocol to be network bound in practice.

In all experiments presented in this section, we configured Fireflies to tolerate up to $p_{\text{corrupt}} = 0.20$ corrupt members with a probability $\varepsilon = 0.99$. The ping and gossiping



(a) Note rate

(b) Accusation rate

Fig. 9. Simulated network overhead when under attack.

intervals were set to 30s and the probabilistic upper bound on the time for gossip to spread, Δ, was set to 2.5min. We set the probability of making a mistaken crash detection in the pinging protocol, P_{mistake}, to 0.01 in order to trigger frequent accidental false accusations by correct members. The total number of members, N, ranged from 20 to 160. Each experiment was run for six simulation hours. In addition, there was a one-hour warm-up period before measurements started to remove bias and effects from the extra load incurred while bootstrapping the system.

Initially, all members are live. After the warm-up period, we simulate churn by periodically crashing and restarting members. Studies of operational P2P systems indicate that churn characteristics vary between individual networks and applications [Stutzbach and Rejaie 2006; Steiner et al. 2009]. Application-specific observations, like those made by Steiner et al. [2009] indicating that the KAD DHT has a Mean Time To Failure (MTTF) of 155min, might not apply to systems built using Fireflies. Since our simulation experiments are primarily used to show linear scalability with membership size rather than specific overhead, we opted for a simple churn model and set both MTTF and Mean Time To Recovery (MTTR) to 6h, exponentially distributed. Each member is then expected to recover or crash at least once during a simulation run. Each experiment was repeated six to eight times with different initial random seeds. For each set of experiments, we calculated 95% confidence intervals. For clarity, the graphs presented in this section contain confidence intervals only where significant.

We varied the fraction of attackers from 0% to 10% with both aggressive and passive attacks, chosen randomly from the set of all members. At the end of each experiment, we ran the simulator for one additional hour with no churn, then checked the views of all correct members. Note that sufficient redundancy must be encoded in p_{corrupt} to accommodate failures of correct members in addition to the explicit attacks. With the simulated configuration and churn, increasing the rate of attacks to 20%, we observed that some simulation runs produced members with divergent views, indicating, as expected, that attackers were at least temporarily successful. We observed no such inconsistencies in the experiments with 0% or 10% fraction of attackers.

Figure 9(a) shows the resulting average rate of notes created for various p_{corrupt} and styles of attack, and Figure 9(b) shows the corresponding rate of created accusations. Aggressive attacks gave a noticeable increase in the rate of notes. This is expected because Rule 6 dictates exactly how many monitors each member can disable. With churn and pinging mistakes, correct members might not be able to permanently disable all their corrupt monitors. For instance, upon receiving a false accusation $\mathcal{A}[m]^{i}_{m'}$ from member m', the correct member m will disable m' in $\mathcal{N}^{i+1}_{m}$, enabling monitoring by some other predecessor m''. This gives m'' an opportunity to execute an aggressive attack on m, even if it was previously disabled for executing such an attack. Similarly, when some member m is correctly detected as crashed, this also gives any corrupt predecessors of m new opportunities for accusing the successors of m. The value for P_{mistake}, MTTF, and MTTR were intentionally chosen to frequently trigger such opportunities in the simulations.

As can be seen in Figure 9(b), the effect of the aggressive attacks is less noticeable in the rate of accusations than in the rate of notes. This is expected as each failed member can correctly be accused by up to $2t$ of its monitors, while an aggressive attack will only produce a single accusation. With continuous churn, correct accusations for failed members are expected to outnumber the ones from the attackers.

The passive attacks decreased the rate of both notes and accusations. This observation is also explained by the simulated churn and pinging mistakes. Because passive attackers refrain from making *any* accusations, they will not make pinging mistakes and will not accuse failed members, subsequently reducing the aggregate rate of both accusations and notes.

7.2. PlanetLab

Our simulations indicate that the network load induced by Fireflies increases linearly with the number of members. To gain a clearer understanding on how Fireflies behaves when running in the wide-area Internet, we deployed our code on PlanetLab [Peterson and Roscoe 2006]. We first ran Fireflies on PlanetLab in early February 2005, and found the experience useful to find pragmatic problems and test our solutions. However, the overheads we measured, some of which are presented in the following, are specific to PlanetLab only.

Each Fireflies agent is instrumented to write a checkpoint to a log on local disk every 10s, containing the current time and approximately 100B of measurement data. The local clocks on PlanetLab machines are synchronized using the Network Time Protocol (NTP). As we are measuring trends over time periods of minutes, the millisecond precision provided by NTP is sufficient for our purpose. We periodically checked all clocks for drift using a local reference clock. Occasionally we observed machines with clock drift and incorrectly configured time zones, which was compensated for during postprocessing of the logs.

7.2.1. Deployment Description. We will now describe the results of one of our PlanetLab experiments. The experiment started on March 12, 2012, and ended March 14, 2012.[4] The purpose of this experiment was to measure the behavior of Fireflies under both low and high churn load and when, at the same time, under attack by corrupt members. Configuration options were set the same as in the previous simulations, except that P_{mistake} was set to a more sensible 10^{-5}.

Our experiment started with Fireflies agents running on 376 PlanetLab machines. At approximately 06:39 on March 13, we terminated 25% of the agents, chosen randomly. At about 14:30 on March 13, these agents were restarted. If an agent has not written a checkpoint to its log during a 1-min period, it is considered crashed in that period. The number of live agents per time period is shown in Figure 10. 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 agents.

Agents were terminated by a script that would log into each individual PlanetLab machine and issue a UNIX `kill` signal. Hence, agents would crash abruptly and without

[4] All dates are in GMT using 24-hour clock notation.

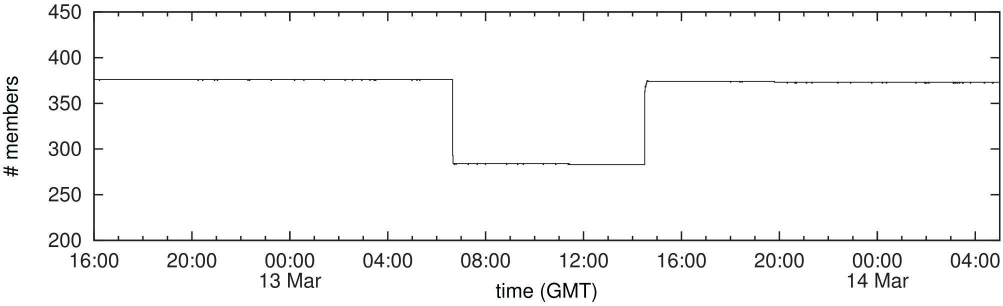


Fig. 10. Number of live members over time during experiment on PlanetLab.

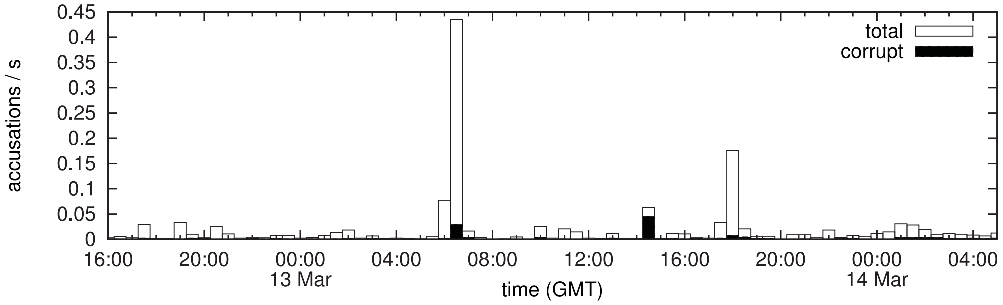


Fig. 11. Rate of accusations on PlanetLab.

warning. Starting agents involved a similar script. The scripts ran from our machine located at the Arctic University of Norway, each taking several minutes to complete.

To measure the impact of corrupt members, 10% of the Fireflies agents were configured to mount aggressive attacks, creating accusations at any opportunity. Another 10% were configured to mount a passive attack, neither accusing nor forwarding accusations to crashed members. Corrupt members were chosen randomly from the set of all members, except seven that were used as trusted boot nodes.

Figure 11 plots the observed aggregate rate of accusations created per second, divided into total accusations and accusations from corrupt members. One peak can be clearly distinguished: when the agents are terminated. This is as expected as the crashed agents are correctly accused by the remaining correct ones. Changes in the membership also gives corrupt members new opportunities to execute an aggressive attack as any new neighbors might not have disabled their new corrupt monitors yet. We see that as an increase in accusations from corrupt members in Figure 11 both when the members are killed and when they are restarted. Otherwise, corrupt members make few accusations, indicating that Fireflies can efficiently thwart such attacks. The peak in accusations at around 18:00 on March 13 was due to a transient outage of several PlanetLab machines.

7.2.2. Membership Churn. All our measurements indicate that there is a fair bit of membership churn not under our control. In Figure 11, we observe this as a relatively regular rate of new accusations being created. Because the liveness of a member in these experiments is determined by its ability to write a checkpoint to file, the churn, seen as a wiggle in Figure 10, is not necessarily due to outages or delays in the network. In practice, we have observed that the majority of PlanetLab nodes tend to be fairly well connected. However, some of the nodes are heavily loaded, to the point of making them

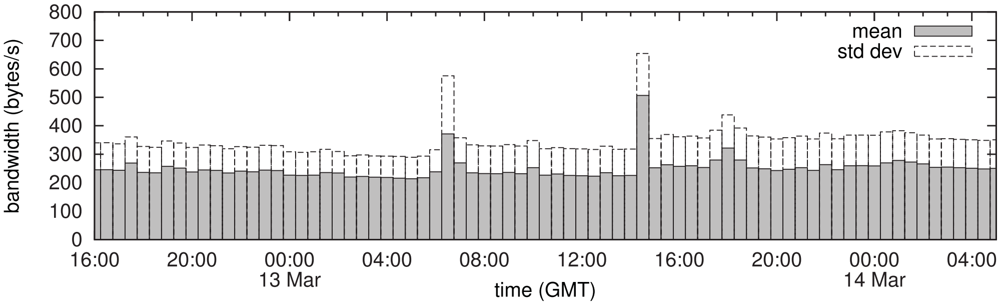


Fig. 12. Outbound TCP usage on PlanetLab.

effectively unreachable. We have also 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, while they could communicate through TCP. This has two consequences for Fireflies. First, a node that cannot receive UDP packets will accuse its successors, even if they are correct. This is not a problem, as these successors will use their *mask* 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 cannot disable all rings, which would have its own problems, leading to the observed continuous background gossip of accusations and notes. We also observed nodes that had problems communicating through TCP.

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 deployments 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 cannot reach until the partition is resolved. Occasionally, however, there are clean partitions. For example, in one run we observed two members in China forming a partitioned Fireflies structure.

7.2.3. Network Overhead. Next, we examine the load that our system incurs on the network. We instrumented our code to log the number of bytes sent and received through TCP. The instrumentation was done as far down in the protocol stack as possible so that all signaling and protocol overhead were captured, but we do not include overhead from TCP and Internet Protocol (IP) headers. Figure 12 shows the mean outbound network consumption per member due to gossiping of certificates, notes, and accusations. The bandwidth follows the rate of accusations, but mostly remains below 250 bytes per second (Bps) per correct member. The various peaks are caused by the previously described issues. The largest observed peaks are when the agents are terminated and restarted, in which case members consumed around 500Bps.

8. DISCUSSION

8.1. Full versus Partial Membership

An alternative to maintaining full membership views at each member is for a membership protocol to provide views containing only 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.

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 in the number of members. Given the availability of cheap memory, this is not necessarily a problem. For instance, in its default configuration with 364B certificates and 108B notes, Fireflies will be able to fit approximately $10{,}000$ members within 4.5 Megabyte (MB) of memory.

Full membership protocols also require that every group member 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 in the size of the membership.

Despite its scalability advantages in the number of member processes, providing only partial membership views has several drawbacks compared to providing full membership information.

(1) Services built on top of a full membership protocol can be made 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 [Pietzuch et al. 2005].
(2) Maintaining overlay structures, like DHTs, requires complex and expensive coordination when having only partial membership information [Stoica et al. 2003]. Complexity can be a real barrier that keeps a protocol from being used in practice [Kreitz and Niemelä 2010].
(3) Partial membership requires messages to be routed through the overlay structure, which make them more likely to get lost along the way and will result in higher end-to-end latency. For instance, DeCandia et al. [2007] argue that avoiding multihop DHT routing was necessary to keep latencies sufficiently small in Amazon's Dynamo key-value store. A similar argument was also used by Kreitz and Niemelä [2010] for the Spotify music application.

In our case, we are also concerned with intrusions and Byzantine faults. Each additional routing hop needed to deliver a message *m* to its destination increases the probability that *m* will be routed through a corrupt member who cannot be trusted to forward or process messages correctly. An additional complication in many DHTs is that lookups tend to converge to the same routing path. Several methods have been proposed to counter this by establishing diverse and distinct routing paths [Urdaneta et al. 2011]. These include probabilistic routing schemes for existing vulnerable overlay structures [Kapadia and Triandopoulos 2008], signing routing tables using a trusted online component [McLachlan et al. 2009], or attesting lookup paths using verifiably assigned shadow members [Mittal and Borisov 2009]. Some DHTs have also been structured specifically with secure routing in mind [Nambiar and Wright 2006].

Structural defenses against intrusions prevent members from using latency and bandwidth optimizing techniques like exploiting network proximity [Gummadi et al. 2003]. Embedding in each message a route that has previously shown itself capable of delivering messages correctly can mitigate some of the overhead incurred by secure DHT routing [Obelheiro and da Silva Fraga 2006], though establishing such routes still incurs significant network overhead in a dynamic environment.

Having a full view of the membership, applications can send messages directly to their destination. This reduces network latency, bandwidth overhead, and avoids having to route messages through potentially corrupt third parties. Whether or not the cost and benefits of maintaining full membership outweighs the costs and complexities associated with secure DHT routing, depends on the application.

8.2. Building Intrusion-Tolerant Services with Fireflies

Although Fireflies maintains its overlay structure in an intrusion-tolerant manner, services built on top of it do not automatically inherit this property. For instance, to ensure safe storage of files, a service built on top of Fireflies must still ensure that each file is replicated to a sufficiently large number of members. In this section, we will discuss some specific ways that Fireflies can be used to construct higher-level services and applications.

8.2.1. Software Dissemination. Disseminating software updates quickly and reliably is important, particularly when they fix security holes. By intruding into the software distribution mechanism, an attacker may try to delay or prevent computers from receiving critical security updates, allowing the attacker more time to construct and deploy malware that targets the vulnerable code [Flake 2004; Brumley et al. 2008]. Open-source communities, like the Linux Kernel Archive and the Ubuntu Linux project, are particularly vulnerable to such attacks because they depend on donated third-party servers, or mirrors, to distribute their software.

FirePatch [Johansen et al. 2007] is a software distribution network that makes use of Fireflies to fight attacks from hostile mirrors that have intruded into the system. By layering a push-pull data dissemination scheme [Pai et al. 2005] on top of the Fireflies gossip mesh, FirePatch will have sufficient link redundancy and diversity so that the set of correct software mirrors will form a connected mesh. Consequently, there exists, with high probability, at least one path of only correct mirrors from the software vendor to each correct mirror and to each correct client, which ensures correct and timely delivery of all data.

8.2.2. Multimedia Streaming. SecureStream [Haridasan and van Renesse 2006] is an intrusion-tolerant multimedia diffusion protocol that layers a push-pull messaging scheme on top of Fireflies in a similar manner as FirePatch. Like security patch distribution, multimedia dissemination is 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 overrequesting, 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 significantly higher ratio of packets within acceptable time than SplitStream [Castro et al. 2003] when under attack.

8.2.3. Distributed Hash Table (DHT). Intrusion-tolerant DHT functionality can be trivially implemented on Fireflies. Assuming object and member identifiers are chosen from the same identifier space, a member can simply consult its view to find the member whose identifier is closest to the object identifier. That member is then the destination, and messages can be sent directly to it. Such an implementation is called a One-Hop

Distributed Hash Table (OHDHT)[5] [Gupta et al. 2003], as messages are not routed through intermediate members. If replication is required, for instance to securely store a file, one or more Fireflies membership rings can be used to assign to each message multiple destinations.

9. RELATED WORK

Membership protocols that provide agreement on membership views in benign environments have been extensively researched within the context of multicast-oriented Group Communication Systems (GCSs). Variants of such systems that tolerate Byzantine failures, such as SecureRing [Kihlstrom et al. 2001] and Rampart [Reiter 1994], have been constructed. However, the overhead of Byzantine consensus makes these protocols unscalable in practice [Gupta et al. 2002].

Byzantine fault tolerance has also been extensively researched in the context of state machine replication. Recent work has come a long way in improving throughput and latency when replicas are correct and the system is stable [Kotla et al. 2007; Guerraoui et al. 2010; Kapitza et al. 2012]. Still, such systems are intended for a small and fixed set of participants connected by high-speed networks.

The Scamp [Ganesh et al. 2003] peer-sampling service uses an epidemic-style membership protocol that, like Fireflies, uses a small number of gossip partners in order to obtain good scalability. Unlike Fireflies, Scamp members maintain only partial membership views and the protocols have no mechanism to verify or enforce that gossip partner selection is random. The Scamp gossip mesh can converge to a nonrandom structure that is not guaranteed to connect all correct members, in particular if Byzantine failures cannot be prevented.

Brahms [Bortnikov et al. 2008] is a peer-sampling protocol that tolerates Byzantine faults by combining push and pull style gossiping with careful orchestration of which members identities to exchange and keep in local views. The Brahms view update protocol is organized in synchronized rounds that require gossip to complete with several partners before the local view can be updated. This might lead to prohibitive high latency and slow convergence of membership views and update propagation. To the best of our knowledge, Brahms has not been implemented.

Tarzan [Freedman and Morris 2002] provides anonymous messaging using a P2P overlay. Participating nodes select relay partners verifiable at random using a ring structure on the full membership, similarly to Fireflies. Unlike Fireflies, Tarzan's gossip-based membership discovery protocol does not impose restrictions on partner selection, which makes it susceptible to DoS attacks. Also, Tarzan does not have a mechanism for collaborative fault detection. Replacing Tarzan's membership protocol with Fireflies would increase its resilience to intrusions and provide lower latency when establishing routes.

The SWIM membership protocol [Das et al. 2002] is similar to Fireflies in that it combines an accusation-rebuttal scheme with a pinging protocol and epidemic dissemination. SWIM is not designed to tolerate Byzantine failures. Although the SWIM pinging scheme will prevent a corrupt member from keeping crashed members within the views of correct members, SWIM cannot 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 crash notification messages. Upon m receiving a crash notification message for m', m will immediately remove m' from its view. There are no restrictions on who can generate crash notification messages for whom.

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

An attacker can therefore use crash notification messages to falsely claim that any correct member has crashed. Unlike Fireflies, SWIM piggybacks membership events on ping messages, which prevents SWIM from taking advantage of set-reconciliation mechanisms to reduce the number of 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.

Secure gossip has been rigorously studied [Malkhi et al. 1999, 2001; Minsky and Schneider 2003; Burmester et al. 2007]. Unlike in the BAR gossip protocol [Li et al. 2006], Fireflies does not distinguish between rational and Byzantine behavior. Pace [2011] argues that rational members in Fireflies might omit forwarding notes and accusations to reduce their bandwidth consumption. In the case that most members are not altruistic, such rational behavior might compromise the system. Pace therefore proposes extending Fireflies with a local blacklist similar to the BAR-B mechanism [Aiyer et al. 2005].

In the S-Fireflies system, Dolev et al. [2007] propose several modifications to the initial Fireflies protocol [Johansen et al. 2006] for increased stability under high churn. Most importantly, S-Fireflies uses a different method for assigning monitors where each member can only accuse its immediate successor in each ring. To accommodate for crashed members, the number of rings is increased so that each member will have the required k unique successors. Although these improvements eliminate the need for skipping crashed members, as in Fireflies, S-Fireflies might potentially require a large number of ring structures to be computed and stored in memory. To see this, let N be the number of members in a S-Fireflies group whereof m' are live (ignoring the members' ability to disable misbehaving monitors). Then, to find k unique predecessors, the total expected number of rings needed is given by $\sum_{i=0}^{k-1} \frac{N}{n-i}$. For groups with mostly live members, only a few extra rings are needed. For instance, in a group of 1,000 members where 10% have crashed and using $k = 21$, then 24 rings would be required on average. However, if 90% of those members have crashed, then 234 rings are needed on average. When all but 21 members have crashed, S-Fireflies would on average require 3,645 rings with a variance of 1.59×10^6 rings. This structure would require 1.59×10^9 hashes to be computed and approximately 95 Gigabyte (GB) of memory.

Another modification proposed by S-Fireflies is to include a list of banned member identifiers in the notes instead of a mask bitmap. This seemed a good idea since it can prevent false accusations when a member has the same misbehaving monitor on multiple rings, and we adapted it into our implementation of Fireflies. However, a list of t member identities is a large data structure compared to a bitmap of $k = 2t + 1$ rings, resulting in a significant increase in bandwidth consumption. Without also adapting the S-Fireflies ring scheme, the benefits of the banned lists did not outweigh the increased bandwidth requirements. We therefore reverted back to our original scheme of disabling rings using bitmaps.

10. CONCLUSION

This article describes and evaluates Fireflies, an overlay network structure that tolerates faults introduced intentionally by an intruder. The key idea behind Fireflies is to organize members in a pseudorandom mesh structure that prevents hostile members from modifying the overlay link topology to their advantage. By providing each member with a full view of all participating members, instead of only a partial view, data-intensive and latency-sensitive services built with Fireflies can avoid costly multihop message routing. To support efficient gossip-based broadcast-style dissemination between members, each member is provided with a small set of neighbors.

Fireflies guarantees that the neighbor graph of correct members is connected with high probability.

There are limitations to what Fireflies can offer. For example, as with any Byzantine fault-tolerant system, Fireflies cannot determine which members are corrupt 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. Given constant churn, members might never reach agreement on the state of the membership. Instead, Fireflies provides eventual and probabilistic consistency among the views of the correct members. With high probability, correct members will agree on the set of long-time correct members and on the set of long-time crashed members. This novel trade-off between scalability, consistency, and security, enables construction of resilient overlay network services for a wide range of real-world data and latency-sensitive Internet applications.

REFERENCES

Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. 2005. BAR fault tolerance for cooperative services. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 45–58. DOI:http://dx.doi.org/10.1145/1095810.1095816

Gal Badishi, Idit Keidar, and Amir Sasson. 2006. Exposing and eliminating vulnerabilities to denial of service attacks in secure gossip-based multicast. *IEEE Transactions on Dependable and Secure Computing* 3, 1 (March 2006), 45–61. DOI:http://dx.doi.org/10.1109/TDSC.2006.12

Paul Barford and Joel Sommers. 2004. Comparing probe- and router-based packet-loss measurement. *IEEE Internet Computing* 8, 5 (Oct. 2004), 50–56. DOI:http://dx.doi.org/10.1109/MIC.2004.34

Rida A. Bazzi and Goran Konjevod. 2005. On the establishment of distinct identities in overlay networks. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*. ACM, New York, NY, 312–320. DOI:http://dx.doi.org/10.1145/1073814.1073873

Jean-Chrysostome Bolot. 1993. Characterizing end-to-end packet delay and loss in the Internet. *Journal of High Speed Networks* 2, 3 (Dec. 1993), 305–323. DOI:http://dx.doi.org/10.3233/JHS-1993-2307

Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. 2008. Brahms: Byzantine resilient random membership sampling. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing (PODC'08)*. ACM, New York, NY, 145–154. DOI:http://dx.doi.org/10.1145/1400751.1400772

David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, 143–157. DOI:http://dx.doi.org/10.1109/SP.2008.17

Mike Burmester, Tri van Le, and Alec Yasinsac. 2007. Adaptive gossip protocols: Managing security and redundancy in dense ad hoc networks. *Ad Hoc Networks* 5, 3 (April 2007), 313–323. DOI:http://dx.doi.org/10.1016/j.adhoc.2005.11.007

Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. 2003. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of the 19th Symposium on Operating Systems Principles (SOSP'03)*. ACM, New York, NY, 298–313. DOI:http://dx.doi.org/10.1145/945445.945474

Andrew Chasin. 2001. *The Gnutella protocol specification*. Specification Version 0.41. Clip2 Distributed Search Solutions. Document revision 1.2.

Fan Chung and Linyuan Lu. 2001. The diameter of random sparse graphs. *Advances in Applied Math* 26, 4 (May 2001), 257–279. DOI:http://dx.doi.org/10.1006/aama.2001.0720

Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. 2011. *The OpenSSL cryptography and SSL/TLS toolkit*. Software Version 0.9.8r. The OpenSSL Software Foundation, http://www.openssl.org.

Abhinandan Das, Indranil Gupta, and Ashish Motivala. 2002. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN'02)*. IEEE, Los Alamitos, CA, 303–312. DOI:http://dx.doi.org/10.1109/DSN.2002.1028914

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*, Vol. 41. ACM, New York, NY, 205–220. DOI:http://dx.doi.org/10.1145/1323293.1294281

Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium (SSYM'04)*. USENIX Association, Berkley, CA, 21–21. http://dl.acm.org/citation.cfm?id=1251375.1251396

Danny Dolev, Ezra N. Hoch, and Robbert Van Renesse. 2007. Self-stabilizing and Byzantine-tolerant overlay network. In *Principles of Distributed Systems*, Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal (Eds.). Lecture Notes on Computer Science, Vol. 4878. Springer, Berlin, Germany, 343–357. DOI:http://dx.doi.org/10.1007/978-3-540-77096-1_25

John R. Douceur. 2002. The Sybil attack. In *Peer-to-Peer Systems*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Lecture Notes on Computer Science, Vol. 2429. Springer, Berlin 251–260. DOI:http://dx.doi.org/10.1007/3-540-45748-8_24

Peter Druschel and Antony Rowstron. 2001. PAST: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*. IEEE, Los Alamitos, CA, 75–80. DOI:http://dx.doi.org/10.1109/HOTOS.2001.990064

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

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

Michael J. Freedman and Robert Morris. 2002. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS'02)*. ACM, New York, NY, 193–206. DOI:http://dx.doi.org/10.1145/586110.586137

Ayalvadi J. Ganesh, Anne-Marie Kermarrec, and Laurent Massoulié. 2003. Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Comput.* 52, 2 (Feb. 2003), 139–149. DOI:http://dx.doi.org/10.1109/TC.2003.1176982

Rachid Guerraoui, Kévin Huguenin, Anne-Marie Kermarrec, Maxime Monod, and Ýmir Vigfússon. 2012. Decentralized polling with respectable participants. *J. Parallel and Distrib. Comput.* 72, 1 (Jan. 2012), 13–26. DOI:http://dx.doi.org/10.1016/j.jpdc.2011.09.003

Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys'10)*. ACM, New York, NY, 363–376. DOI:http://dx.doi.org/10.1145/1755913.1755950

Krishna P. Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. 2003. 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*. ACM, New York, NY, 381–394. DOI:http://dx.doi.org/10.1145/863955.863998

Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. 2003. One hop lookups for peer-to-peer overlays. In *Proceedings of the of the 9th Conference on Hot Topics in Operating Systems (HOTOS'03)*. USENIX Association, Berkley, CA, 7–12.

Indranil Gupta, Kenneth P. Birman, and Robbert van Renesse. 2002. Fighting fire with fire: Using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International* 18, 3 (June 2002), 165–184. DOI:http://dx.doi.org/10.1002/qre.473

Frank Harary. 1962. The maximum connectivity of a graph. *Proceedings of the National Academy of Sciences of the United States of America* 48, 7 (July 1962), 1142–1146. http://www.pnas.org/content/48/7/1142.short.

Maya Haridasan and Robbert van Renesse. 2006. Defense against intrusion in a live streaming multicast system. In *Proceedings of the 6th International Conference on Peer-to-Peer Computing*. IEEE, Los Alamitos, CA, 185–192. DOI:http://dx.doi.org/10.1109/P2P.2006.15

Russell Housley, Warwick Ford, Tim Polk, and David Solo. 2002. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. Request for Comments 3280. The Internet Society.

Håvard Johansen, André Allavena, and Robbert van Renesse. 2006. Fireflies: Scalable support for intrusion-tolerant network overlays. In *Proceedings of the 1st ACM European Conference on Computer Systems (Eurosys'06)*. ACM, New York, NY, 3–13. DOI:http://dx.doi.org/10.1145/1217935.1217937

Håvard Johansen, Dag Johansen, and Robbert van Renesse. 2007. FirePatch: Secure and time-critical dissemination of software patches. In *New Approaches for Security, Privacy and Trust in Complex Environments*, Hein Venter, Mariki Eloff, Les Labuschagne, Jan Eloff, and Rossouw von Solms (Eds.). IFIP AICT, Vol. 232. Springer, New York, NY, 373–384. DOI:http://dx.doi.org/10.1007/978-0-387-72367-9_32

Håvard D. Johansen. 2007. *Intrusion-tolerant membership management for peer-to-peer overlay networks*. PhD dissertation. University of Tromsø.

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

Apu Kapadia and Nikos Triandopoulos. 2008. Halo: High-assurance locate for distributed hash tables. In *Proceedings of the 16th Annual Network & Distributed System Security Symposium*. Internet Society, Reston, VA, Article 4, 19 pages. http://www.internetsociety.org/events/ndss-symposium-2008.

Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 295–308. DOI:http://dx.doi.org/10.1145/2168836.2168866

Anne-Marie Kermarrec, Laurent Massoulié, and Ayalvadi J. Ganesh. 2003. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems* 14, 3 (March 2003), 248–258. DOI:http://dx.doi.org/10.1109/TPDS.2003.1189583

Kim Potter Kihlstrom, Louise E. Moser, and Peter M. Melliar-Smith. 2001. The SecureRing group communication system. *ACM Transactions on Information and System Security* 4, 4 (Nov. 2001), 371–406. DOI:http://dx.doi.org/10.1145/503339.503341

Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine fault tolerance. In *Proceedings of the 21st Symposium on Operating Systems Principles (SOSP'07)*. ACM, New York, NY, 45–58. DOI:http://dx.doi.org/10.1145/1294261.1294267

Gunnar Kreitz and Fredrik Niemelä. 2010. Spotify-Large scale, low latency, P2P music-on-demand streaming. In *Proceedings of the 10th IEEE International Conference on Peer-to-Peer Computing (P2P'10)*. IEEE, Los Alamitos, CA, 1–10. DOI:http://dx.doi.org/10.1109/P2P.2010.5569963

John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*. ACM, New York, NY, 190–201. DOI:http://dx.doi.org/10.1145/378993.379239

Avinash Lakshman and Prashant Malik. 2010. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2 (April 2010), 35–40. DOI:http://dx.doi.org/10.1145/1773912.1773922

Harry C. Li, Allen Clement, Edmund L. Wong, Jeff Napper, Indrajit Roy, Lorenzo Alvisi, and Michael Dahlin. 2006. BAR gossip. In *Proceedings of the 7th Symposium on Operating System Design and Implementation (OSDI'06)*. USENIX Association, Berkley, CA, 191–204.

Gary Locke and Patrick Gallagher. 2009. *Digital Signature Standard (DSS)*. FIPS PUB 186-3. National Institute of Standards and Technology.

Dahlia Malkhi, Yishay Mansour, and Michael K. Reiter. 1999. On diffusing updates in a Byzantine environment. In *Proceedings of the 18th Symposium on Reliable Distributed Systems*. IEEE, Los Alamitos, CA, 134–143. DOI:http://dx.doi.org/10.1109/RELDIS.1999.805090

Dahlia Malkhi, Michael K. Reiter, Ohad Rodeh, and Yaron Sella. 2001. Efficient update diffusion in Byzantine environments. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*. IEEE, Los Alamitos, CA, 90–98. DOI:http://dx.doi.org/10.1109/RELDIS.2001.969758

Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. 2009. Scalable onion routing with Torsk. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, New York, NY, 590–599. DOI:http://dx.doi.org/10.1145/1653662.1653733

Yaron Minsky and Fred B. Schneider. 2003. Tolerating malicious gossip. *Distributed Computing* 16, 1 (Feb. 2003), 49–68. DOI:http://dx.doi.org/10.1007/s00446-002-0082-4

Yaron Minsky and Ari Trachtenberg. 2002. *Practical Set Reconciliation*. Technical Report 2002-01. Boston University.

Prateek Mittal and Nikita Borisov. 2009. ShadowWalker: Peer-to-peer anonymous communication using redundant structured topologies. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*. ACM, New York, NY, 161–172. DOI:http://dx.doi.org/10.1145/1653662.1653683

Arjun Nambiar and Matthew Wright. 2006. Salsa: A structured approach to large-scale anonymity. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. ACM, New York, NY, 17–26. DOI:http://dx.doi.org/10.1145/1180405.1180409

Rafael R. Obelheiro and Joni da Silva Fraga. 2006. A lightweight intrusion-tolerant overlay network. In *Proceedings of the 9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. IEEE, Los Alamitos, CA, 496–503. DOI:http://dx.doi.org/10.1109/ISORC.2006.7

Alessio Pace. 2011. *Gossiping in the wild-Tackling practical problems faced by gossip protocols when deployed in the Internet*. Ph.D. dissertation. University of Grenoble.

Vinay S. Pai, Kapil Kumar, Karthik Tamilmani, Vinay Sambamurthy, and Alexander E. Mohr. 2005. Chainsaw: Eliminating trees from overlay multicast. In *Peer-to-Peer Systems IV*, Miguel Castro and Robbert van Renesse (Eds.). Lecture Notes on Computer Science, Vol. 3640. Springer, Berlin, 127–140. DOI:http://dx.doi.org/10.1007/11558989_12

Larry Peterson and Timothy Roscoe. 2006. The design principles of PlanetLab. *ACM SIGOPS Operating Systems Review* 40, 1 (Jan. 2006), 11–16. DOI:http://dx.doi.org/10.1145/1113361.1113367

Peter Pietzuch, Jeffrey Shneidman, Jonathan Ledlie, Matt Welsh, Margo Seltzer, and Mema Roussopoulos. 2005. Evaluating DHT-based service placement for stream-based overlays. In *Peer-to-Peer Systems IV*, Miguel Castro and Robbert van Renesse (Eds.). Lecture Notes on Computer Science, Vol. 3640. Springer, Berlin, 275–286. DOI:http://dx.doi.org/10.1007/11558989_25

Michael K. Reiter. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd Conference on Computer and Communications Security (CCS'94)*. ACM, New York, NY, 68–80. DOI:http://dx.doi.org/10.1145/191177.191194

Rodrigo Rodrigues and Charles Blake. 2004. When multi-hop peer-to-peer lookup matters. In *Peer-to-Peer Systems III*, Geoffrey M. Voelker and Scott Shenker (Eds.). Lecture Notes on Computer Science, Vol. 3279. Springer, Berlin, 112–122. DOI:http://dx.doi.org/10.1007/978-3-540-30183-7_11

Atul Singh, Miguel Castro, Peter Druschel, and Antony Rowstron. 2004. Defending against Eclipse attacks on overlay networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*. ACM, New York, NY, Article 21, 6 pages. DOI:http://dx.doi.org/10.1145/1133572.1133613

Emil Sit and Robert Morris. 2002. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*, Peter Druschel, Frans Kaashoek, and Antony Rowstron (Eds.). Lecture Notes on Computer Science, Vol. 2429. Springer, Berlin, 261–269. DOI:http://dx.doi.org/10.1007/3-540-45748-8_25

Mudhakar Srivatsa and Ling Liu. 2004. Vulnerabilities and security threats in structured overlay networks: A quantitative analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*. IEEE, Los Alamitos, CA, 252–261. DOI:http://dx.doi.org/10.1109/CSAC.2004.50

Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. 2009. Long term study of peer behavior in the KAD DHT. *IEEE/ACM Transactions on Networking* 17, 5 (Oct. 2009), 1371–1384. DOI:http://dx.doi.org/10.1109/TNET.2008.2009053

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. 2003. Chord: A scalable peer-to-peer lookup protocol for Internet applications. *IEEE/ACM Transactions on Networking* 11, 1 (Feb. 2003), 17–32. DOI:http://dx.doi.org/10.1109/TNET.2002.808407

Daniel Stutzbach and Reza Rejaie. 2006. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement (IMC'06)*. ACM, New York, NY, 189–202. DOI:http://dx.doi.org/10.1145/1177080.1177105

Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. 2011. A survey of DHT security techniques. *Comput. Surveys* 43, 2, Article 8 (Feb. 2011), 49 pages. DOI:http://dx.doi.org/10.1145/1883612.1883615

Bimal Viswanath, Mainack Mondal, Krishna P. Gummadi, Alan Mislove, and Ansley Post. 2012. Canal: Scaling social network-based Sybil tolerance schemes. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*. ACM, New York, NY, 309–322. DOI:http://dx.doi.org/10.1145/2168836.2168867

Scott Wolchok and J. Alex Halderman. 2010. Crawling BitTorrent DHTs for fun and profit. In *Proceedings of the 4th USENIX Conference on Offensive Technologies (WOOT'10)*. USENIX Association, Berkeley, CA, 1–8.

Received June 2013; revised November 2014; accepted December 2014