PeerReview: Practical Accountability for Distributed Systems

Andreas Haeberlen[†‡], Petr Kouznetsov[†], and Peter Druschel[†]
[†]Max Planck Institute for Software Systems, [‡]Rice University

Technical Report MPI-SWS-2007-003

ABSTRACT

We describe PeerReview, a system that provides accountability in distributed systems. PeerReview ensures that Byzantine faults whose effects are observed by a correct node are eventually detected and irrefutably linked to a faulty node. At the same time, PeerReview ensures that a correct node can always defend itself against false accusations. These guarantees are particularly important for systems that span multiple administrative domains, which may not trust each other.

PeerReview works by maintaining a secure record of the messages sent and received by each node. The record is used to automatically detect when a node's behavior deviates from that of a given reference implementation, thus exposing faulty nodes. PeerReview is widely applicable: it only requires that a correct node's actions are deterministic, that nodes can sign messages, and that each node is periodically checked by a correct node. We demonstrate that PeerReview is practical by applying it to three different types of distributed systems: a network filesystem, a peer-to-peer system, and an overlay multicast system.

1. INTRODUCTION

Nodes in distributed systems can fail for many reasons: a node can suffer a hardware or software failure; an attacker can compromise a node; or a node's operator can deliberately tamper with its software. Moreover, faulty nodes are not uncommon [46]. At large scale, it is increasingly likely that some nodes are accidentally misconfigured or have been compromised as a result of unpatched security vulnerabilities.

In systems that span multiple administrative domains, the lack of central administration tends to aggravate these problems. Moreover, multiple trust domains pose the additional threat of deliberate manipulation by node operators with different interests. Examples of systems with multiple administrative domains are network services such as DNS, NTP, NNTP and SMTP, federated information systems, computational Grids, Web services, peer-to-peer systems, and the Internet's inter-domain routing system.

We consider the use of *accountability* to detect and expose node faults in distributed systems [63]. For the purposes of this paper, an accountable system maintains a tamper-evident record that provides non-repudiable evidence of all nodes' actions. Based on this record, a faulty node whose observable behavior deviates from that of a correct node can be detected eventually. At the same time, a correct node can defend itself against any false accusations.

Clearly, accountability by itself is not sufficient for systems in which faults can have serious and irrecoverable effects, such as deleting all replicas of a file or dispensing cash from an automated teller machine. However, accountability offers several benefits, by itself and in combination with other techniques:

- **Deterring faults:** The mere presence of accountability can reduce the incidence of certain faults. For instance, the threat of exclusion may discourage freeloading, and the threat of public embarrassment may cause organizations to update and configure their software with increased diligence.
- **Detection in fault tolerant systems:** Accountability complements Byzantine fault tolerant techniques such as data or state machine replication [13]. Systems that use these techniques can tolerate a bounded number of faulty nodes, typically less than one-third of the system size. By enabling the timely recovery of faulty nodes, accountability can help these systems to maintain this bound, thus increasing their ability to withstand successive node faults.
- **Detection in best-effort systems:** Systems that provide best-effort services can naturally tolerate recoverable faults, because such faults "merely" degrade the system's service. However, too many unidentified faults can degrade the system to the point of becoming unusable. Accountability enables recovery, which helps to maintain the system's health.
- **Assigning blame:** In systems with multiple trust domains, accountability irrefutably pinpoints the party responsible for a problem, while allowing other principals to prove their innocence to customers, peering partners, and authorities.

This paper describes and evaluates PeerReview, a general and practical system that provides accountability for distributed systems. PeerReview creates a per-node secure log, which records the messages a node has sent and received, and the inputs and outputs of the application. Any node i can request the log of another node j and independently determine whether j has deviated from its expected behavior. To do this, i replays j's log using a reference implementation that defines j's expected behavior. By comparing the results of the replayed execution with those recorded in the log, PeerReview can detect Byzantine faults without requiring a formal specification of the system.

This is an expanded version of a paper that appeared at the 21st ACM Symposium on Operating Systems Principles (SOSP'07).

An accountable system requires strong node identities. Otherwise, an exposed faulty node could escape responsibility by assuming a different identity [21]. In this paper, we assume that each node is in possession of a cryptographic key pair that can be linked to a unique node identifier. We assume that correct nodes' actions are deterministic and that communication between correct nodes eventually succeeds. These assumptions are common in the literature and reasonable in practice [3, 13, 29, 60].

PeerReview was designed for deployment in the Internet and is subject to certain limitations imposed by that environment. First, PeerReview cannot expose a faulty node that ignores some messages but never sends a message that a correct node would not send. The reason is that packet loss, or network or processing delays could make a correct node temporarily seem as though it were ignoring a message. With PeerReview, a node that does not respond to a message is eventually suspected by every correct node. A suspected node can exonerate itself by acknowledging the message. By restricting correct nodes to communicate with only non-suspected nodes, we can isolate a suspected node until it complies.

Second, PeerReview detects a fault after a correct node is causally affected by the fault, but not before. Since PeerReview does not assume the availability of trusted probes at each node, it can reliably observe only messages sent and received by correct nodes. Thus, PeerReview can detect only faults that manifest themselves through these messages. Detecting faults before they impact correct nodes would require PeerReview to make much stronger assumptions. We think that PeerReview strikes a reasonable balance between detection power and wide applicability in real-world systems.

We have applied PeerReview to three example systems: a network filesystem, a peer-to-peer email system, and an overlay multicast system. An experimental evaluation shows that PeerReview is applicable to a range of systems and that its overhead is reasonable in practice.

Specifically, signing messages imposes a fixed processing delay; this overhead is noticeable only in a LAN and when messages are short. The processing and message overheads depend on the fault assumptions and the desired strength of the detection guarantee. The processing overhead grows linearly with the number of nodes that need to inspect a given node's actions to be sure at least one of the inspecting nodes or the inspected node is correct. If we insist that every instance of misbehavior is eventually detected, then PeerReview's message overhead grows with the square of the number of nodes in the system; experiments show that this limits PeerReview's scalability to moderately sized systems of hundreds of nodes. However, if we settle for a probabilistic detection guarantee, then the message overhead becomes logarithmic in the number of nodes, and PeerReview can scale to much larger systems.

The rest of this paper is structured as follows. We discuss related work in Section 2. We provide a precise definition of the type of accountability and fault detection provided by PeerReview in Section 3. We describe the PeerReview algorithm in Section 4; our implementation in Section 5; and our three applications in Section 6. Section 7 presents the results of our evaluation, and Section 8 concludes this paper.

2. RELATED WORK

Accountability in distributed systems has been suggested as a means to achieve practical security [34], to create an incentive for cooperative behavior [20], to foster innovation and competition in the Internet [4, 35], and even as a general design goal for dependable networked systems [62]. However, it has been an open question whether accountability can be implemented in a general and efficient manner [63].

PeerReview offers strong accountability for any distributed system that can be modeled as a collection of deterministic state machines. To our knowledge, no prior work has achieved this level of generality. The type of accountability used in PeerReview, and a precise definition of the types of faults it can detect, has appeared in a prior workshop position paper [24]. This paper contributes the design and implementation of a practical system with these properties, as well as experimentally evaluates it with three different applications.

CATS [64] implements a network storage service with strong accountability properties. Like PeerReview, it maintains secure logs that record the messages sent and received by each node. Unlike PeerReview, however, CATS depends on a trusted publishing medium that ensures the integrity of these logs. CATS detects faults by checking logs against a set of rules that describes the correct behavior of a specific system (a network storage service). PeerReview does not require a formal specification of correct behavior; it verifies the nodes' behaviors by replaying their logs using the system's existing reference implementation.

Repeat and Compare [39] uses accountability to ensure content integrity in a peer-to-peer CDN built on untrusted nodes. It detects faults by having a set of trusted verifier nodes locally reproduce a random sample of the generated content, and by comparing the results to the content returned by the untrusted nodes.

The study of the Byzantine failure model originate in two papers by Lamport, Pease, and Shostak [33, 47]. State machine replication [32, 54] is a classic technique for masking a limited number of such Byzantine faults. Byzantine fault tolerance (BFT) protocols [13,49,61] can mask faults as long as less than one-third of the nodes are faulty [9]. Aiyer et al. [1] introduced the BAR model, which can tolerate a limited number of Byzantine nodes plus an unlimited number of "rational" nodes. PeerReview complements the existing work on BFT by providing strong accountability.

Generic simulations suggest a more general manner of handling Byzantine faults. Bracha [8] has described a protocol that hides the malicious effects of Byzantine faults by simulating more benign "identical Byzantine" faults on top of them. Simulations of even more restrictive classes of omission and crash faults in the Byzantine failure model were proposed by Srikanth and Toueg [57], Neiger and Toueg [45], Coan [17], and Bazzi and Neiger [7]. Unlike PeerReview, these protocols are not intended to provide verifiable evidence of misbehavior. They are typically designed for broadcast-based algorithms and assume a synchronous system or a large fraction of correct nodes.

A trusted computing platform detects faults that involve modifications to a node's software [23]. This approach requires special hardware features, a trusted OS kernel, and a software and hardware certification infrastructure.

The concept of failure detectors, abstract oracles that produce information about faults, was introduced by Chandra

and Toueg [15] for the crash failure model. An extension of this work to a specific class of "muteness" failures was explored by Malkhi and Reiter [37] and Doudou et al. [22]. Kihlstrom et al. [30] proposed a consensus algorithm using a failure detector that produces suspicions of "detectable" misbehavior. Unlike the detector in PeerReview, [30] does not provide evidence of misbehavior and assumes algorithms that are based on reliable broadcast: every message is relayed to all nodes when received for the first time.

A technique for a statistical evaluation of the number of nodes with Byzantine faults was proposed by Alvisi et al. [2]. This technique is designed for specific replicated data services based on quorums and does not provide a means to identify which nodes are faulty.

A variety of techniques address specific types of misbehavior. These techniques include secure routing for structured overlays [11, 56], incentive mechanisms to prevent freeloading [18, 43], and content entanglement to prevent censorship [59]. These techniques can be more efficient than PeerReview, because they tend to be tailored to an application and a specific type of fault; however, they are difficult to reuse and offer no protection against unforeseen types of misbehavior. PeerReview offers a reusable, general detection mechanism for a large class of faults, including unanticipated faults.

Intrusion detection systems (IDS) can handle certain types of protocol violations [19, 26, 31]. However, they either are based on heuristics and require careful balancing between false positives and false negatives, or require a formal specification of the expected behavior, which can be difficult to write and maintain for a complex system. PeerReview avoids these problems by using a reference implementation of the system as an implicit specification.

Reputation systems such as EigenTrust [27] can be used to detect Byzantine faults, but they typically can detect only nodes that misbehave repeatedly. Also, a coalition of faulty nodes can denounce a correct node. PeerReview can detect even a single instance of detectable misbehavior, and it never exposes a correct node.

PeerReview's secure logging and auditing techniques were inspired by secure timelines [38], as well as by earlier work on tamper-evident logs [55]. Secure histories have been used for other purposes [16]. The fork consistency model introduced with SUNDR [36] is similar to PeerReview's model of observable behavior.

3. DETECTING FAULTY BEHAVIOR

Making a distributed system accountable involves two steps: the first creates a secure record of all nodes' actions, the second inspects the recorded information and detects faulty behavior. In this section, we consider the problem of fault detection and discuss some of its limitations.

3.1 The problem of detection

We consider a distributed system of *nodes* in which each node is supposed to follow a given *protocol*. If it does, then we call the node *correct*; otherwise, we call the node *faulty*.

To illustrate the approach, we consider a simple toy protocol (Figure 1a). In this protocol, each node is responsible for providing and allocating 10 units of a resource, e.g. storage space. At any time, a node (client) may request resources from another node (server) by sending a message `REQUEST_k`. If the server has sufficient resources, it must allocate k units of the resource for the client and return a message `GRANT_k`. When the client is done with the resource, it sends a `RELEASE_k` to the server so that the server can release the k units and grant other requests.

There are two ways in which a node can be faulty in this protocol: it can either send a message that a correct node would never have sent in a given state (Figures 1c and 1d) or ignore a message a correct node would have accepted (Figure 1e).

Ideally, if a node misbehaves in either of these ways, each correct node should detect the misbehavior (we say that the correct node *exposes* the faulty node). An *ideal* fault detector should thus guarantee

- **Ideal completeness:** Whenever a node becomes faulty, it should be exposed by all correct nodes.

However, malicious nodes may try to disrupt the system by tricking some correct nodes into exposing other correct nodes. To prevent this, an ideal detector should also guarantee

- **Ideal accuracy:** No correct node is ever exposed by a correct node.

Of course, these idealistic requirements still lack important details. For example, it is often difficult to decide whether a message is being ignored or just delayed. It may also be difficult to identify misbehavior that cannot be observed by any correct node. Before we refine our requirements (Section 3.5), we consider some practical limitations of distributed detection mechanisms.

3.2 What can (or cannot) be detected?

Naturally, the power of any detection system is limited by the data that are (or are not) available to it. In this paper, we assume that a node must reason about faults of other nodes based on the messages it receives. This results in the following fundamental limitations for a practical detector.

First, we can only detect faults that directly or indirectly affect a message. For example, we cannot detect that a node's CPU is overheating or that its display has failed. This would require a more powerful detector, e.g. one that has access to special hardware.

Second, it is difficult to verify whether a node correctly reports its external inputs. For example, a faulty weather station might report a light breeze as a heavy gale. Detecting faults of this type would require additional information, e.g. other weather stations in the same area or a satellite image.

Third, in an asynchronous system, it can be difficult to distinguish omission faults from messages with a particularly long delay. For example, in the absence of strong synchrony assumptions, if a node has been granted resources but does not release them for a long time, we can only *suspect* that it has become faulty, but there is no 'smoking gun' that would allow us to expose it. The concept of suspected nodes is well known from the literature on unreliable fail-stop failure detectors [15]. A suspicion may sometimes turn out to be groundless, e.g. if the `RELEASE` message eventually arrives.

Finally, we can only detect faults that are *observable by a correct node*. The reason is that we cannot expect the faulty nodes to share any information about the messages they have observed. For example, they could be trying to cover the traces of other faulty nodes with whom they are

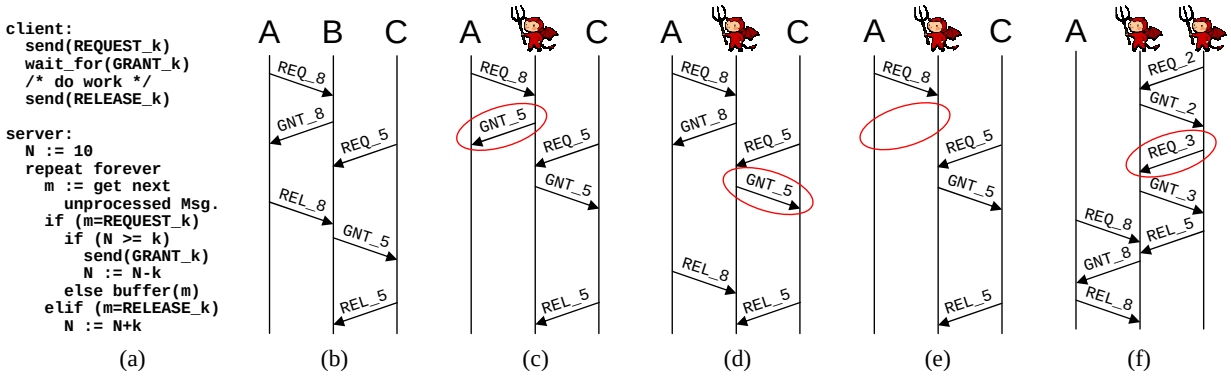

Figure 1: A simple example protocol (a) and five message exchanges: Correct behavior (b), incorrect response from node B that A can detect without C's help (c), incorrect response from node B that A and C can detect only if they cooperate (d), missing response from node B (e), and faulty behavior by node C with B as an accomplice (f). In case (f), because node A cannot observe C's behavior, it cannot expose the fault on C unless B cooperates.

colluding. Hence, if one faulty node sends a telltale message to another faulty node without changing its observable state, we cannot hope to expose the deviating node on the basis of this message only. Consider the example in Figure 1f. Here, node B can clearly observe that C is faulty, because it issues a second request without releasing its resources first. However, node A can only observe B's `GRANT_8` message, which is correct. If node B does not share its observations with A, it can prevent C from being exposed, which makes B a *faulty accomplice* of C.

However, if none of the correct nodes can observe a particular fault, it is of little practical relevance because, from the correct nodes' perspective, the faulty nodes are acting *as if they were correct*.

3.3 The letter versus the spirit of the law

It is tempting to think of fault detection as a panacea that works against all kinds of bad, or 'faulty', behavior. However, recall that our definition of 'faulty' depends on a specific protocol. Thus, an action may be against the *intent* behind the protocol (the spirit of the law), but may be perfectly legal with respect to the protocol's specification (the letter of the law).

For instance, our example protocol in Figure 1a is not deadlock-free. Therefore, adding fault detection will neither make it deadlock-free nor enable nodes to detect or resolve deadlocks. However, if we augment the protocol to be deadlock-free, then a faulty node can create a deadlock only by violating the protocol. Fault detection will then be able to identify the fault, thus enabling the correct nodes to break the deadlock.

A similar argument applies to external input that is not verifiable. Consider a cooperative storage system that allows users to declare freely the amount of storage they wish to contribute. If "zero" is a valid setting, fault detection cannot expose freeloaders who use this setting; their behavior is against the intent behind the protocol but does not violate its specification. However, if the protocol specifies a minimum storage capacity, fault detection can expose participants who refuse to store at least the minimal amount of data.

3.4 Definitions

Before we can formally specify the properties of our practical detector, we need to introduce some definitions.

We consider a set Π of *nodes*. Every node i is modeled as a state machine A_i and a detector module B_i (Figure 2). Informally, we say that a node i is correct if it respects the specifications of both A_i and B_i. Otherwise, the node is faulty.

Nodes communicate with each other through message passing. We assume that messages are uniquely identified. For a message m, let $sender(m)$ and $receiver(m)$ denote the sender and the receiver of m, respectively. For the moment, we do not put any restrictions on local processing time and communication delays. However, we assume that, after some number of retransmissions, a message sent from a correct node to a correct node is eventually received.

An *event* is either $send_i(m) \in O_i$, where $i = sender(m)$, or $receive_j(m) \in I_j$, where $j = receiver(m)$, or an application-specific input or output. We distinguish events associated with the state machine A_i and events associated with the detector module B_i. We say that sequence E_i of events associated with A_i (respectively, B_i) is *valid* if it *conforms* to A_i (respectively, B_i), i.e., given the sequence of incoming messages and application-specific inputs in E_i, A_i (respectively, B_i) could have produced the sequence of outgoing messages and application-specific outputs in E_i.

Let E be a sequence of events. $E|A_i$ denotes the subsequence of E that consists of all events associated with A_i in E, and $E|B_i$ denotes the subsequence of E that consists of all events associated with B_i in E. We say that a node i is *correct* in E if (1) both $E|A_i$ and $E|B_i$ are valid, and (2) if E is infinite, then both $E|A_i$ and $E|B_i$ are also infinite. Otherwise we say that i is *faulty* in E.

We call a sequence of events E an *execution* if in E, each m is sent and received at most once, each $receive_i(m)$ is preceded by the corresponding $send_j(m)$, and, in case E is infinite, every message sent from a correct (in E) node to a correct (in E) node is received.

A *history* of a node i is a sequence of events of A_i. A pair (h_1, h_2) of histories of i is *consistent* if h_1 is a prefix of h_2, or vice versa. If i is a correct node in an execution E, one

trivial example of a valid history is $E|A_i$; if E' is a prefix of E, then the pair of histories $(E|A_i, E'|A_i)$ is consistent.

Let $\mathcal{M}(E)$ denote the set of messages received by the nodes in an execution E. We assume that there exists a *history map* $\mathcal{H}$ that associates every message $m \in \mathcal{M}(E)$ with a history of *sender*(m). For a correct node, $\mathcal{H}(m)$ is the prefix of the local history $E|$*sender*(m) up to and including *send*(m). Thus, for any message m sent by a correct node, $\mathcal{H}(m)$ is valid, and for every pair of messages m and m' sent by a correct node, $\mathcal{H}(m)$ and $\mathcal{H}(m')$ are consistent.

We say that a message m is *observable in* E if there exists a correct node i and a sequence of messages $m_1, \ldots, m_k$ such that

(i) $m_1 = m$,
(ii) *receive*(m_k) belongs to $E|A_i$,
(iii) for all $j = 2, \ldots, k$: *receive*(m_{j-1}) belongs to $\mathcal{H}(m_j)$.

In other words, m is observable if it causally precedes at least one event on a correct node.

We say that a node i is *detectably faulty* with respect to a message m sent by i in an execution E if m is observable in E and satisfies one of the following properties:

(1) $\mathcal{H}(m)$ is not valid (for i)
(2) There exists a message m' that was also sent by i and is observable in E, such that $\mathcal{H}(m)$ and $\mathcal{H}(m')$ are inconsistent

If i is detectably faulty with respect to m, then the set of nodes causally affected by m and m' (if m' exists) are called *accomplices* of i with respect to m.

We say that a node i is *detectably ignorant* in E if i is not detectably faulty in E and there exists a message m sent to i by a correct node, such that, for all observable messages m' sent by i, *receive*$_i(m)$ does not appear in $\mathcal{H}(m')$.

3.5 Problem statement

Now we are ready to relax and refine our "idealistic" requirements from Section 3.1.

PeerReview produces two kinds of fault indications: *exposed* and *suspected*. Intuitively, we say that a node i is exposed by a node j if j has a proof of i's misbehavior, and i is suspected by j if, according to j, i has not acknowledged a certain message sent to it. Note that j withdraws the "suspected" indication when it learns that i accepted the message. To account for long message delays, we require only that faults are detected *eventually*, after some delay.

Now we define the following requirements:

- **Completeness:** (1) Eventually, every detectably ignorant node is suspected forever by every correct node, and (2) if a node i is detectably faulty with respect to a message m, then eventually, some faulty accomplice of i (with respect to m) is exposed or forever suspected by every correct node.

- **Accuracy:** (1) No correct node is forever suspected by a correct node, and (2) no correct node is ever exposed by a correct node.

Although a system with these properties is weaker than our idealized detector from Section 3.1, it is still very strong in practice: every instance of detectably faulty behavior is eventually detected, and there are no false positives. As we will see later, relaxing completeness in favor of a probabilistic detection guarantee permits a highly scalable implementation, while still detecting faults with high probability and avoiding false positives.

4. DESIGN OF PEERREVIEW

In the previous section, we discussed fault detection and defined its properties. Next, we describe the design of PeerReview, an accountability system that satisfies these properties. A formal proof of these properties can be found in Appendix A.

4.1 Overview

We first describe a simplified version of PeerReview, called FullReview. For FullReview, we make the (unrealistic) assumption that there is a trusted entity that can reliably and instantly communicate with all nodes in the system. The system's membership is static, and each node knows the specification of the entire system.

FullReview works as follows: All messages are sent through the trusted entity, which ensures that all correct nodes observe the same set of messages in the same order. Furthermore, each node i maintains a log λ_{ij} for each other node j, and in this log it records all messages that were sent either from or to j. Periodically, i checks each of its logs against the system specification. If a node i finds that a node j has not yet sent the message it should have sent in its last observed state, then i *suspects* j until that message is sent. If j has sent a message it should not have sent according to the specification, then i *exposes* j.

It is easy to see that FullReview is both complete and accurate. On the one hand, when a node i sends an "incorrect" message, the trusted entity forwards the message to all nodes, and i is exposed by every correct node. Also i is suspected by every correct node as long as a message from i is missing. On the other hand, no correct node can be exposed or indefinitely suspected by any correct node.

However, FullReview is based on strong assumptions: a trusted, reliable communication medium and a formal system specification. Moreover, FullReview's complexity is at least quadratic in the number of nodes, for messages, storage, and computation. In PeerReview, we refine this simple design to arrive at a practical system:

- Each node only keeps a full copy of its own log; it retrieves other logs when necessary. Nodes exchange just enough information to convince another node that a fault is, or is not, present.

- Tamper-evident logs and a commitment protocol ensure that each node keeps its log consistent with the set of messages it has exchanged with all correct nodes or else risk exposure.

- Each node is associated with a small set of other nodes, who act as its *witnesses*. The witnesses collect evidence about the node, check its correctness, and make the results available to the rest of the system.

- PeerReview uses a reference implementation of the node software to check logs for faulty behavior. Thus, it does not require a formal system specification, which is difficult to obtain and maintain in practice.

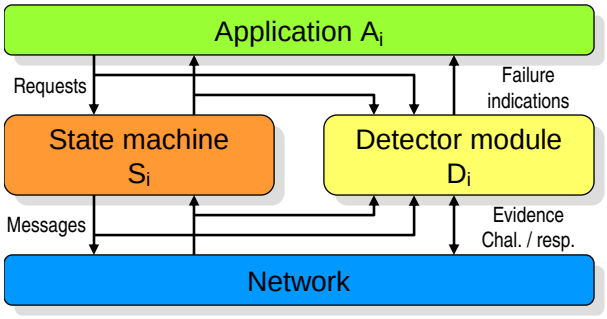

Figure 2: Information flow between application, state machine, and detector module on node i.

- PeerReview uses a challenge/response protocol to deal with nodes that do not respond to some messages. This allows PeerReview to operate on an unreliable network that satisfies only weak synchrony assumptions.

We will describe each of these refinements in the following subsections.

4.2 System model

Each node i is modeled as a state machine S_i, a detector module D_i, and an application A_i (Figure 2). The state machine represents all the functionality that should be checked by PeerReview, whereas the application represents other functions that need not be checked, e.g. a GUI. The detector module D_i implements PeerReview; it can observe all inputs and outputs of S_i, and it can communicate with the detector modules on other nodes. We assume that a correct node implements S_i and D_i as specified, whereas a faulty node may behave arbitrarily.

The detector module issues *failure indications* about other nodes to its local application. Informally, *exposed(j)* is raised when i has obtained proof of j's misbehavior; *suspected(j)* says that i suspects that j does not send a message that it is supposed to send; *trusted(j)* is issued otherwise.

4.3 Assumptions

The design of PeerReview is based on the following assumptions:

1. The state machines S_i are deterministic.
2. A message sent from one correct node to another is eventually received, if retransmitted sufficiently often.
3. The nodes use a hash function $H(\cdot)$ that is pre-image resistant, second pre-image resistant, and collision resistant.

Assumptions 1–3 are common for techniques based on state machine replication [54], including BFT [13].

4. Each node has a public/private keypair bound to a unique node identifier. Nodes can sign messages, and faulty nodes cannot forge the signature of a correct node.
5. Each node has access to a reference implementation of all S_j. The implementation can create a snapshot of its state, and its state can be initialized according to a given snapshot.

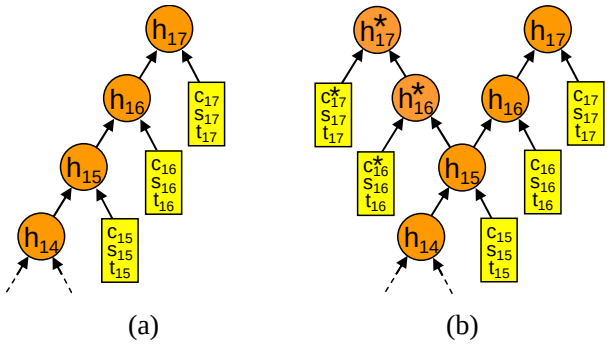

Figure 3: (a) A linear log and its hash chain, which is recursively defined on the log entries, and (b) a forked log with two branches.

6. There is a function w that maps each node to its set of witnesses. It is assumed that for each node i, the set $\{i\} \cup w(i)$ contains at least one correct node; otherwise, PeerReview might lose completeness with respect to node i.

Assumption 4 can be met, for instance, by installing each node with a certificate that binds the node's public key to its unique identifier. However, any type of name binding that avoids Sybil attacks [21] will work. In symmetric systems where all nodes run the same protocols, a node can simply use its own implementation as the reference implementation (Assumption 5). Otherwise, nodes can obtain a reference implementation for another node from a trusted source. The appropriate definition of w (Assumption 6) depends on the system configuration, which will be discussed in Section 5.3.

4.4 Tamper-evident logs

To enforce accountability, PeerReview must keep a secure record of the inputs and outputs of each node, and it must be able to detect if that record has been tampered with. PeerReview implements such a record using a technique inspired by secure histories [38].

A *log* is an append-only list that contains all the inputs and outputs of a particular node's state machine in chronological order. The log also contains periodic state snapshots and some annotations from the detector module. Each log entry $e_k = (s_k, t_k, c_k)$ has a sequence number s_k, a type t_k, and some type-specific content c_k. The sequence numbers must be strictly increasing but may be non-contiguous; for example, a timestamp could be used. Additionally, each record includes a recursively defined hash value $h_k = H(h_{k-1}||s_k||t_k||H(c_k))$ (Figure 3a); $||$ stands for concatenation. The base hash h_{-1} is a well-known value.

The resultant hash chain, along with a set of *authenticators*, makes the log tamper-evident. An authenticator $\alpha_k^j = \sigma_j(s_k, h_k)$ is a signed statement by node j that its log entry e_k has hash value h_k; $\sigma_j(\cdot)$ means that the argument is signed with j's private key.

By sending α_k^j to node i, a node j *commits* to having logged entry e_k and to the contents of its log before e_k. If j subsequently cannot produce a prefix of its log that matches the hash value in α_k^j, then i has verifiable evidence that j has tampered with its log and is therefore faulty.

Moreover, i can use α_k^j as verifiable evidence to convince other nodes that an entry e_k exists in j's log. Any node

can also use α_k^j to inspect e_k and the entries preceding it in j's log. To inspect x entries, i challenges j to return $e_{k-(x-1)}, \ldots, e_k$ and h_{k-x}. If j responds, i calculates the hash value h_k from the response and compares it with the value in the authenticator. If j has not returned the correct log entries in the correct order, the hash values will differ. In this case, i has evidence that j is faulty. We discuss the case in which j does not respond to the challenge in Section 4.8.

Summary: *Logs and authenticators form a tamper-evident, append-only record of a node's inputs and outputs.*

4.5 Commitment protocol

We must ensure that a node cannot add an entry to its log for a message it has never received. Also, we have to ensure that a node's log is complete, i.e. that it contains an entry for each message sent or received by the node to or from a correct node.

When node i sends a message m to node j, i must commit to having sent m, and j must commit to having received m. They obtain an authenticator from the other node included in the message and its acknowledgment, respectively. This authenticator covers the corresponding log entry. A log entry for a received message must include a matching authenticator; therefore, a node cannot invent log entries for messages it never received.

When i is about to send m to j, it creates a log entry $(s_k, \texttt{SEND}, \{j, m\})$, attaches h_{k-1}, s_k and $\sigma_i(s_k||h_k)$ to m, and sends the result to j. Thus, recipient j has enough information to calculate h_k and to extract α_k^i. If the signature in α_k^i is not valid, j discards m. Otherwise, j creates its own log entry $(s_l, \texttt{RECV}, \{i, s_k, m\})$ and returns an acknowledgment with h_{l-1}, s_l, and $\sigma_j(s_l||h_l)$ to i. This allows i to extract and verify α_l^j. If i does not receive a valid acknowledgment, i sends a challenge to j's witnesses; we will explain the details in Section 4.8.

Summary: *The commitment protocol ensures that the sender (respectively the receiver) of each message* m *obtains verifiable evidence that the receiver (respectively the sender) of* m *has logged the transmission.*

4.6 Consistency protocol

A faulty node can attempt to escape detection by keeping more than one log or a log with multiple branches (Figure 3b). For example, this could be a promising strategy for node B in Figure 1d, who might keep one log for messages from A and another for messages from C. Both logs would show correct behavior, even though node B is clearly faulty. To avoid this attack, we exploit the fact that a node can produce a connecting log segment for each pair of authenticators it has ever signed if, and only if, it maintains a single, linear log.

If a node i receives authenticators from another node j, it must eventually forward these authenticators to the witness set $w(j)$. Thus, the witnesses obtain verifiable evidence of all the messages j has sent or received. Periodically, each witness $\omega \in w(j)$ picks the authenticators with the lowest and the highest sequence number and challenges j to return all log entries in this range. If j is correct, these log entries form a linear hash chain that contains the hash values in all the other authenticators. If they do not, ω has obtained verifiable evidence that j is faulty. We discuss the case in which j does not respond to the challenge in Section 4.8.

Finally, each $\omega \in w(j)$ uses the log entries to extract all the authenticators that j has received from other nodes and sends them to the corresponding witness sets. This is necessary because j could be acting as a faulty accomplice of some node k; it could forward k's messages without sending the authenticators to $w(k)$. We note that this step has $O(|w(j)| \cdot |w(k)|)$ message complexity.

Summary: *The consistency protocol ensures that each node either maintains a single, linear log that is consistent with all the authenticators the node has issued, or it is exposed by at least one correct witness.*

4.7 Audit protocol

In the next step, we use a node's log to check whether the node's behavior conforms to that of its reference implementation. Each witness ω of a node i periodically looks up its most recent authenticator from i (say, α_k^i) and then challenges i to return[1] all log entries since its last audit, up to and including e_k. Then ω appends the new entries to its local copy $\lambda_{\omega i}$ of i's log.

Next, ω locally creates an instance of i's reference implementation and initializes it with a recent snapshot from $\lambda_{\omega i}$. Then, it replays all the inputs starting from that snapshot and compares S_i's output with the output in the log. Since we require that S_i be deterministic, any discrepancy indicates that i is faulty. In this case, ω can use α_k^i and a suffix of $\lambda_{\omega i}$ as verifiable evidence against i, and this evidence can be checked by any correct node.

Summary: *The audit protocol ensures that, for each node* i*, either* i*'s actions are consistent with the reference implementation of* i*'s state machine, or* i *is exposed by at least one correct witness.*

4.8 Challenge/response protocol

The protocols described so far can expose faulty nodes if they respond to challenges. But what if a node does not respond to a challenge, or it fails to acknowledge a message that was sent to it? Unless we make stronger assumptions about synchrony, we cannot distinguish an uncooperative faulty node from a correct node that is slow or suffering from network problems.

When a node j concludes that another node i is refusing to cooperate, it indicates the suspected state for i and creates a *challenge* for i. The challenge must contain enough evidence to convince another correct node that if i were correct, it would be able to answer. The node j then sends the challenge to i's witnesses, who forward it to i. If i does not send a response, the witnesses indicate that i is suspected. There are two types of challenges.

An *audit challenge* consists of two authenticators: a_k^i and a_l^i with $k < l$. After checking the signatures on a_k^i and a_l^i, any correct node has enough evidence to convince itself that either i is faulty or the entries e_k and e_l must exist in i's log. If i is correct, it can answer the challenge with the corresponding log segment, whose hash values will form a hash chain connecting a_k^i to a_l^i.

A *send challenge* consists simply of a message m and the extra information appended by the consistency protocol (Section 4.6). After extracting and checking the authen-

[1]Because the audit protocol and the consistency protocol inspect the same log entries, our implementation retrieves them only once. Here, we separate the two protocols for clarity of presentation.

ticator from m, any correct node is convinced that i must acknowledge m. If i is correct and has not yet received m, i can accept m now and return an acknowledgment. If i has already received m, it can simply re-send the earlier acknowledgment.

Summary: *The challenge/response protocol ensures that, if a node* i *fails to respond to a challenge or does not acknowledge a message, it is eventually suspected by at least one correct witness. The suspicion persists until the node answers the challenges or acknowledges the message.*

4.9 Evidence transfer protocol

The mechanisms described so far ensure that at least one correct node obtains verifiable evidence of each fault or a challenge for each suspected node. We also need to make sure that all correct nodes eventually collect the same evidence (the same set of audit and send challenges) against faulty nodes. We achieve this by allowing every node i to periodically fetch the challenges collected by the witnesses of every other node j. Note that, in most practical settings, i may be interested only in accusations against the nodes that (directly or indirectly) communicate with it. In this case, i needs to contact only the witnesses of these nodes.

If a correct node i obtains a challenge for another node j, its detector indicates *suspected(j)*. When i receives a message from j in this state, it challenges j. Once i has received valid answers to all pending challenges, its detector indicates *trusted(j)* again. If i obtains a proof of j's misbehavior, its detector outputs *exposed(j)*.

Summary: *The evidence transfer protocol ensures that all correct nodes eventually output a failure indication for each faulty node.*

4.10 Summary

The use of logs and authenticators helps to maintain a shared tamper-evident, append-only record of every node's activity. By periodically checking and replaying the logs, and comparing the results with the provided authenticators, the consistency, commitment and audit protocols ensure that a detectably faulty node is exposed or permanently suspected by at least one of its correct witnesses. The challenge/response protocol ensures that a node is suspected if it is unresponsive but allows a correct node to exonerate itself by responding. Finally, the evidence transfer protocol allows the correct nodes to share their evidence and eventually to come to the same conclusion about each detectably faulty node. To summarize, PeerReview has the completeness and accuracy properties described in Section 3.5. A proof of these properties can be found in Appendix A.

4.11 Extension: Probabilistic guarantees

PeerReview provides strong guarantees under very conservative failure assumptions. This limits its scalability to moderately large systems. If we assume N nodes that send M messages per unit time, as well as an upper bound φ on the fraction of faulty nodes, then we can ensure strong completeness only if we assign $\psi = \lceil \varphi N \rceil$ witnesses to each node. In this case, the per-node message complexity is dominated by the consistency protocol's $O(M\psi^2)$ complexity and hence grows with $O(MN^2)$.

However, there are two extensions that can considerably improve PeerReview's scalability, at the expense of a slightly relaxed completeness guarantee. Rather than asking for strong completeness, we can require only that

1. the probability that the system contains a node with an all-faulty witness set is at most P_f (for example, $P_f = 10^{-5}$); and
2. the probability that a single instance of misbehavior is missed is at most P_m (for example, $P_m = 0.01$).

4.11.1 Smaller witness sets

If we accept a small probability $P_f > 0$ that an all-faulty witness set exists, we can choose the witness set size ψ as the smallest size that meets this constraint. If the system has N nodes of which a fraction φ is faulty, then we arrive at

$$1 - (1 - \varphi^{\psi})^N \leq P_f$$

If we want P_f to be independent of N, we need to set the witness set size ψ as follows:

$$\psi = \left\lceil \frac{\ln\left(1 - (1 - P_f)^{\frac{1}{N}}\right)}{\ln \varphi} \right\rceil$$

How does ψ grow with the system size? For $N \gg 1$, we get

$$\begin{aligned} \psi &\approx \frac{\ln\left(1 - (1 - P_f)^{\frac{1}{N}}\right)}{\ln \varphi} \\ &\approx \frac{\ln\left(1 - (1 + \frac{\ln(1-P_f)}{N})\right)}{\ln \varphi} \\ &\approx \frac{\ln N - \ln \ln \frac{1}{1-P_f}}{\ln \frac{1}{\varphi}} \end{aligned}$$

Thus, ψ now grows with $O(\log N)$ instead of $O(N)$. Since ψ is also the number of logs each witness must audit, the per-node auditing overhead is reduced from $O(MN)$ to $O(M \log N)$.

4.11.2 Probabilistic consistency checking

For consistency checking, the 'worst-case scenario' is a faulty node i that forks its log and records just a single message in the second branch, which is sent to an accomplice j. To detect this, *some* correct witness in $w(j)$ must send the authenticator to *some* correct witness in $w(i)$. It is sufficient that one such transmission takes place; however, if there are multiple correct nodes in $w(i)$ and/or $w(j)$, PeerReview will perform many redundant transmissions.

If we accept a small probability $P_m > 0$ that a given instance of misbehavior is missed, we can reduce the redundancy by using *probabilistic* consistency checking. In this approach, when a witness in $w(j)$ has an authenticator to send to a witness in $w(i)$, it transmits it only with probability ξ and discards it otherwise. At first glance, this reduces the message complexity only by a constant factor. However, the number of redundant transmissions grows with the size of the witness set, so we can compensate by reducing ξ as a function of N.

How quickly can ξ be reduced as the system grows? In order to satisfy the second condition, we need to make sure that the probability of at least one successful transmission (from a correct witness to a correct witness) is at least $1 - P_m$. Hence, we get

$$\left[1 - \xi\,(1 - \varphi)^2\right]^{\psi^2} \leq P_m$$

which leads us to

$$\xi \geq \frac{1 - P_m{}^{\frac{1}{\psi^2}}}{(1-\varphi)^2}$$

If we choose the smallest permissible value ξ_{min}, the message complexity of one consistency checking step is $\xi_{min}\ \psi^2$ or, in the limit,

$$\begin{aligned} & \lim_{\psi\to\infty} \frac{1 - P_m{}^{\frac{1}{\psi^2}}}{(1-\varphi)^2} \cdot \psi^2 \\ = & \lim_{\psi\to\infty} \frac{1 - (1 + \frac{\ln P_m}{\psi^2})}{(1-\varphi)^2} \cdot \psi^2 \\ = & \frac{\ln \frac{1}{P_m}}{(1-\varphi)^2} \end{aligned}$$

which no longer increases with ψ. Thus, the number of authenticators that each node needs to transmit is reduced from $O(M\psi^2)$ to $O(M)$.

Why does $P_m > 0$ lead to such a marked improvement? The reason is that for $P_m = 0$, the consistency protocol must guarantee completeness even when both the sender's and the receiver's witness set contain just a single correct node. As the size of the witness set grows, this extreme case becomes more and more unlikely.

4.11.3 Summary: Probabilistic guarantees

If we apply both extensions, we can reduce the per-node overhead for auditing to $O(M \log N)$ and the per-node overhead for consistency checking to $O(M)$. Thus, the total overhead of PeerReview with probabilistic guarantees grows with $O(MN \log N)$. Our results in Section 7.7 show that this is sufficient to scale a PeerReview-enabled system to more than $10,000$ nodes.

5. IMPLEMENTATION

In this section, we describe our implementation of PeerReview. We discuss optimizations and practical engineering challenges, such as log truncation. Then, we discuss how the implementation can be adapted to specific environments through an appropriate choice of witnesses and configuration parameters.

5.1 The PeerReview library

Our PeerReview codebase is written as a C++ library, which includes all application-independent parts of the system. The library is used by all three example applications described in Section 6 and can be reused for other systems. It implements the tamper-evident logs and PeerReview's five protocols. External libraries provide the cryptographic primitives and a transport layer for sending and receiving messages. The application system is expected to provide various callbacks, including one that instantiates a reference implementation of another node, which is used when PeerReview needs to check that node's log.

Conceptually, the library is interposed between the transport layer and the application. All incoming and outgoing messages go through the library. The library may append headers, such as an authenticator for the consistency protocol, or send messages of its own, e.g. a challenge. Our PeerReview library contains $5,961$ lines of code, counted by the number of newlines. It is available for download from the project homepage [48].

The library makes use of two simple optimizations to save bandwidth in the auditing protocol. First, when transferring log segments to a witness, it replaces all the outputs and state snapshots with hash values. The witness does not lose information that way, because it obtains the outputs during replay. It can compare the hash of these outputs with the hash values in the log. Second, if part of a message is never examined by the state machine, the library hashes that part also, because it does not affect the replay. For example, a message handled by a routing protocol can be reduced to its routing header and a hash of its body. The library provides the full set of snapshots and outputs upon request, e.g. when a new witness needs to be initialized.

Another optimization applies to the consistency protocol. Rather than forwarding authenticators to the witnesses immediately, the library buffers them locally for some time T_{buf} and then sends them in batches. This increases the time to detection by up to T_{buf} but increases message efficiency because authenticators are very small. An authenticator with a SHA-1 hash and a 1024-bit RSA signature is just 156 bytes long.

The library also supports authenticated dictionaries [44], which applications can use to commit efficiently to large data structures, such as an entire disk image. This is useful, e.g. when a node generates evidence. The node can include only those parts of the data structure that the recipient needs to check the evidence, e.g. the disk blocks that have actually been accessed.

5.2 Log truncation

PeerReview's append-only log must eventually be truncated. A simple solution is to allow each node to discard all log entries older than some time T_{trunc}. This approach requires very loosely synchronized clocks. Whether a node's clock is sufficiently accurate can be checked by PeerReview.

Despite log truncation, exposed faulty nodes remain exposed forever, because the incriminating evidence can be verified without access to the corresponding log entries. A fault could remain undetected if legitimate evidence of the fault were to surface only after the log was truncated. This case can be avoided by keeping logs much longer than the expected duration of node and network outages. In practice, log entries can and should be kept at least on the order of months.

As a result of log truncation, a suspected node that was never exposed may become trusted again after T_{trunc} has elapsed. However, an attacker cannot gain much leeway from this exoneration, because a faulty node must remain silent for at least $T_{trunc} - T_{audit} \approx T_{trunc}$ (e.g. several months) after each misbehavior to avoid exposure. Moreover, a human operator could infrequently check for nodes that have been suspected for a long time and permanently revoke or refuse to renew such nodes' certificates.

5.3 Configuring witnesses

Witness configuration depends on the type of system and the nature of the deployment in which PeerReview would be used. In a system that is overseen by a single organization (e.g. Planetlab), one can configure a dedicated set of machines as witnesses for all nodes. In a federated system (e.g. Internet inter-domain routing), each organization may wish to act as a witness for the organization's peering partners. In a client-server system (e.g. a Web service), clients

may act as witnesses for the servers they depend on. Alternatively, replicated servers can act as mutual witnesses. Lastly, in a peer-to-peer system (e.g. Skype), each node can be witnessed by a random set of other participating nodes.

Depending on the choice of witness configuration, an appropriate function w, which maps each node to its witness set, is defined. In the cases in which the system membership is relatively static, we can simply specify w in a configuration file, signed by the appropriate authority and distributed throughout the system. If membership is dynamic, w must be dynamic as well; new witnesses can initialize their state by first obtaining some recent authenticators from the old witnesses and then performing an audit to obtain the latest checkpoint.

In peer-to-peer systems, we use consistent hashing [28] to map witnesses to nodes; in this case, each node acts as a witness for the k nodes whose node identifiers are closest to its own. This approach spreads the auditing overhead evenly across the nodes, and it minimizes the number of witness sets that are affected by a random fault. Nodes must not be allowed to choose their own identifiers, and they must be able to securely evaluate w at runtime, even in the presence of faulty nodes. Secure routing [11] ensures this.

In systems with dynamic witness sets, additional bandwidth is required to initialize new witnesses, which may limit PeerReview's tolerance of high levels of churn. Note that *some* rate of churn among witnesses is actually desirable in configurations with small witness sets (Section 4.11): in this case, it ensures that the (unlikely) state in which some faulty node's witnesses are all faulty does not persist indefinitely.

5.4 Choosing parameters

PeerReview's most important parameter is the size of a node's witness set. Consider, for instance, the case when witness sets of all nodes have the same size ψ. A higher ψ increases overhead. PeerReview's storage requirement and CPU overhead grow with $O(\psi)$, and its message complexity with $O(\psi^2)$. To maintain PeerReview's completeness guarantee, ψ must be chosen such that for each node i, the set $\{i\} \cup w(i)$ contains at least one correct node at any given time. In other words, either i or one of its ψ witnesses must be correct. In practice, ψ should be chosen as the minimal value that satisfies this criterion.

If a lower value is chosen for ψ, PeerReview might lose its strong completeness guarantee, i.e., it can miss some faults. However, it still does not suspect or expose any correct nodes. For more details, see Section 4.11.

The audit interval T_{audit} and the time limit T_{buf} for buffering authenticators determine the maximum time to detection. Shorter intervals result in quicker detection and thus reduce the damage a faulty node can do before it is evicted; however, short intervals also increase message overhead because logs are transferred in small pieces rather than in large segments.

It is also possible to perform audits on demand, rather than periodically. For instance, in a system providing best-effort service, witnesses may refrain from auditing until they observe a loss of service quality that indicates the presence of faults. With this policy, detection is not guaranteed during periods when the witnesses do not audit. During periods when the witnesses audit, PeerReview provides the usual guarantees.

The key length k of the nodes' keys must be high enough to prevent an attacker from forging signatures; however, higher values also mean more CPU load for signing and verifying authenticators. For our experiments, we used RSA with the recommended key length of 1024 bits [5].

5.5 Alternatives to fault detection

PeerReview has two main components. One component provides a tamper-evident record of all observable actions; the other component detects faults by regularly checking that record against a reference implementation. For some applications, it may be useful to replace the second component with another fault detection mechanism. For example, systems like BGP [50] have formal specifications of correct or incorrect behavior; in these cases, the auditors could directly check the record against the specification, e.g. using [6]. Another example is an electronic voting system, in which the secure record could be inspected in court in case an election is contested.

6. APPLICATIONS

In this section, we first discuss the general steps for applying PeerReview to an application, and then we describe our three example applications.

6.1 How to apply PeerReview

Applying PeerReview to a new system generally involves two steps. The first step is to decide which parts of the system should be included in the state machine S_i for each node i and thus checked by PeerReview. Including more subsystems potentially enables PeerReview to detect more types of faults. However, because PeerReview allows any node to audit past and current states of S_i, any subsystems that deal with sensitive information, e.g. with cryptographic keys or clear-text private information, should not be included.

The second step is to make sure S_i is deterministic. Given some initial state and a sequence of inputs, S_i must always produce the same outputs and finish in the same state. In practice, many systems have some sources of non-determinism, but usually these can be identified and dealt with. For example, if a system uses a pseudo-random number generator, we include its seed value in the state snapshot, so the same random sequence is generated during replay. If a protocol relies on real time, e.g. for timestamps or timeouts, we record the time of each event in the log; we use this information to update a virtual clock used during replay and to trigger timeouts.

If a system is concurrent and generates events in a non-deterministic order, we can use synchronization to make the order deterministic. For instance, we can add synchronization to ensure that jobs finish in the order in which they were started; alternatively, we can log the sequence in which the jobs actually finished and enforce this sequence during replay.

In general, adding PeerReview to a system presents the same challenges as adding state-machine replication (e.g. BFT), and the same solutions apply (see e.g. [10]). In addition to eliminating sources of non-determinism, PeerReview requires state snapshots to enable replay by witnesses, whereas BFT requires snapshots to initialize new replicas. To transfer snapshots between different implementations, BFT can use state abstraction [14]; this technique can be applied to PeerReview as well.

6.2 Application #1: Overlay multicast

Our first application models an overlay multicast system like SplitStream [12], which delivers streaming content, such as audio or video, from a single source to a potentially large set of client nodes. By applying PeerReview to this system, we gain the ability to detect and isolate misbehaving nodes, e.g. nodes that tamper with the content, or nodes who refuse to contribute resources to the system.

6.2.1 System overview

In overlay multicast systems, most of the required bandwidth is provided by the clients themselves. For example, the client nodes can form a tree topology; the source sends the content to the root of the tree, and each tree node forwards it to its direct children. Thus, the content eventually reaches all the nodes, even though each node – including the source – contributes only a constant amount of bandwidth to the system.

For our experiments, we used a *multi-tree* system, in which the content is striped across k different trees. Thus, each node is required to forward roughly as much as it receives.

6.2.2 Technical details

For simplicity, we assume that system membership is static, and that each client knows the identities of all the other clients. However, this is not a necessary assumption; the tree could be constructed dynamically, as in SplitStream, and we could use a structured overlay to map nodes to witnesses, as shown in our third application.

Each client uses its membership list to construct $k = 10$ trees with outdegree 2 and thus learns its parent and its children in each tree. The source then sends content at a rate of 300 kbps in 18.7 kB chunks, which are distributed round-robin across the root nodes of the trees. Since clients usually have a different depth in each tree, they buffer incoming blocks for some time, so they can deliver them to the application in the right order.

For each chunk, a client c can determine the expected arrival time τ as well as the parent p in the corresponding tree. If the chunk has not arrived by time $\tau + \delta$, c audits a segment of p's log around τ. In the audit challenge, c explicitly specifies that messages sent from p to c should *not* be hashed. If the chunk was lost in transit from p to c, then c can obtain it from the response; if p does not respond, c forwards the audit challenge to p's witnesses.

The state machine includes just the content distribution algorithm. It exports a `send` method, which the source uses to publish new chunks, and it invokes a `receive` upcall whenever the local node has received a new chunk. The mapping from nodes to witnesses is static; we use consistent hashing on the node identifiers to assign $\psi = 2$ witnesses to each node.

6.3 Application #2: Network filesystem

Our second application models a group of NFS file servers that exports a set of volumes to its clients. PeerReview allows these servers to check each other. Because NFS servers do not tolerate Byzantine faults, a faulty server may deliver incorrect data to a client. However, PeerReview allows us to detect quickly if an attacker has tampered with one of the file servers, e.g. by removing or corrupting data.

6.3.1 System overview

In our experiment, each file server i exports a volume V_i via the NFSv2 [58] protocol. In this protocol, servers and clients communicate using remote procedure calls (RPCs). For example, if a client wants to read data from a file on volume V_i, it sends a `READ` RPC to server i, which checks the client's authorization and, if successful, returns a message with the requested data.

NFS is a client/server system, and it uses two different state machines. The state machine implemented by the servers is complex, and there are many examples of incorrect state transitions that PeerReview can expose, e.g. when a read request does not return the data that was previously written. The state machine implemented by the clients, however, is trivial: it simply accepts requests from the kernel and converts them into RPCs, which it then sends to the server. Thus, clients cannot break the protocol in a nontrivial way. Therefore, we decided to restrict auditing to the servers. Clients do not keep a log, and they do not have any witnesses. However, they do sign and acknowledge messages, they forward authenticators to the servers' witness sets, and they will challenge a server if it does not acknowledge one of their messages.

6.3.2 Technical details

We use the Linux 2.6.15 kernel on both the servers and the clients. On the servers, the state machine consists of the kernel-level NFS server and the `ext2` file system; on the client nodes, the state machine consists of the kernel-level NFS client.

Since we wanted to minimize changes to the kernel, we implemented the PeerReview logic in user-level wrapper processes. When the client sends an RPC to a server, the message is intercepted by the client-side wrapper, which encapsulates it in a PeerReview message and sends it to the server-side wapper. The server-side wrapper logs the message, extracts the RPC and forwards it to the NFS server in the kernel. The response is processed in a similar way, passing through the server-side and client-side wrappers before it arrives at the NFS client.

Since the `ext2` file system is not completely deterministic, we had to apply a small 467-line kernel patch to remove sources of nondeterminism. This patch implements the following two changes:

- It allows the user-level wrapper process to control the timestamps used by the file system, e.g. when it updates the access time field of an inode. Witnesses use this function when they replay log entries that were recorded earlier.
- It removes sources of randomness. For example, the original block allocator sometimes uses the ID of the current process as a random number. Instead, we use a function of the current timestamp.[2]

Nondeterminism can also be caused by concurrent RPCs that access the same file, since these do not necessarily complete in the same order in which they were issued. For example, a `READ` request might complete before or after a conflicting `WRITE` request, with different results. To avoid this, the

[2]According to a comment in `ext2_find_near`, the original code colors the starting block of an allocation with the PID modulo 16 to 'prevent it from clashing with concurrent allocations for a different inode in the same block group'. Our transformation should preserve this property.

server-side wrapper must delay RPCs that could potentially conflict, and issue them serially.

Since we expect system membership to change rarely, we specify the mapping from nodes to witnesses in a configuration file. We obtain the mapping by using consistent hashing on the servers' identifiers.

To save bandwidth, each witness maintains a full replica of the volumes it audits, rather than repeatedly transferring checkpoints. This state of this replica corresponds to the state of the original volume after the last replayed log entry. When the witness obtains a new log segment during an audit, it applies the operations in the log to its local replica and thus brings it up to date.

Since a witness cannot easily create a second instance of the kernel-level NFS server, it exports the local replicas as additional volumes, but blocks any RPCs from clients that are trying to access them. When the witness needs to replay an RPC from a log entry, it first locates any file handles in the message and modifies them such that they point to the local replica (by changing the device number in the handle). Next, it changes the filesystem time of the replica volume to the timestamp in the log entry. Finally, it simply fowards the RPC to its local NFS server. Thus, when the NFS server on the witness processes the RPC, it performs the same operations on the replica volume that the NFS server on the server has performed on the original volume.

To support checkpoints efficiently, we developed another kernel extension that monitors accesses to each volume's disk image and keeps a list of all blocks that have been modified since the last checkpoint. The server-side wrapper can use this information to update an authenticated dictionary that covers all the blocks in the volume; the root of this dictionary is included in each checkpoint entry. This is also useful to reduce the size of evidence: rather than including a complete snapshot of the file system with each proof of misbehavior, we can include just the blocks that are accessed when the corresponding log segment is replayed.

6.4 Application #3: Peer-to-peer email

Our third application is ePOST [41], a peer-to-peer email service.

6.4.1 Overview

In ePOST, email and email folders are stored in a distributed hashtable (DHT), which is cooperatively implemented by all participating nodes. The DHT is replicated for availability and durability [25], and it can adapt to node failures and churn. To ensure confidentiality, all content is encrypted before it is added to the DHT. Metadata is kept in per-user logs, and all mutable state is written by a single writer (the user's node) which can sign the state, as in Ivy [42]. The key of an immutable object in the DHT is the hash of its content, and the hash of a mutable object is the hash of the public key of the corresponding writer. This makes an object's integrity easy to verify.

Given the existing security and fault tolerance mechanisms in ePOST, the remaining threat is denial of service. A faulty node can misroute or drop messages and thus prevent other nodes from retrieving content, or it can manipulate the topology of the peer-to-peer system to hide the presence of other nodes. We use PeerReview to identify misbehaving nodes and to eject them from the system. Thus, faulty nodes cannot degrade the (best-effort) service indefinitely.

6.4.2 Technical details

ePOST contains several subsystems: it is built on the POST [40] platform, which in turn relies on PAST [52] for storage, Scribe [53] for event notification, Glacier [25] for archival storage, and Pastry [51] for key-based routing. To address the threats mentioned earlier, it is sufficient to include PAST and Pastry in the state machine. Glacier already includes mechanisms for handling Byzantine faults, POST only works with a node's local data structures, and Scribe is used merely as an optimization. ePOST's mechanisms for IMAP handling and encryption need not be checked by PeerReview, which has the additional advantage of keeping key material and email cleartext out of the log, and thus preserves confidentiality.

Unlike our other applications, ePOST has dynamic membership, so we cannot statically assign a witness set to each node. Instead, we dynamically choose the witness set of a node i to be the ψ live nodes whose identifiers are closest to i's identifier. When another node j needs to contact i's witnesses at runtime, j first uses use secure routing [11] to locate the corresponding nodes; to save bandwidth, j caches the results locally for some time.

Each node also needs to know the ψ other nodes for which it must act as a witness, i.e. the ψ live nodes whose identifiers are closest to its own. This set overlaps with Pastry's leaf set, so the information is already available. However, this set may change over time (due to joins and failures), so a node i sometimes finds that it has become a witness for a new node j. In this case, i immediately asks each of j's other witnesses for their most recent authenticator α_k^j; then it begins to audit j, starting with the most recent checkpoint before log entry e_k. Note that completeness is guaranteed only if j's witness set contains at least one correct witness *at all times*; if the last correct witness fails before a new one can take over, j has an opportunity to fork its history without getting exposed.

We could not experiment with a live deployment because ePOST no longer has a significant user base; instead, our experiments are based on message traces from a past deployment provided by the ePOST authors.

6.5 Summary

Our three example applications are not the only, and may not be the most natural, applications for PeerReview. We chose them for two reasons. First, they represent different types of systems that PeerReview can be applied to. The filesystem is a small-scale client-server system for a LAN environment, whereas overlay multicast and ePOST are decentralized, cooperative systems for wide-area deployment. Second, the applications place stress on different aspects of PeerReview and explore its limitations. For instance, the fileserver has a large amount of state and a high rate of latency-sensitive requests. Overlay multicast has a high message rate and transmits a large volume of data. ePOST is a complex peer-to-peer system that includes distributed storage and a DHT.

7. EVALUATION

In this section, we present experimental results from our three example applications. We cover the systems' behavior under faults, the overheads of running PeerReview and its impact on application performance.

7.1 Methodology

Each of our applications places stress on a different aspect of PeerReview. For example, the network filesystem is sensitive to latency increases and is limited by throughput, whereas ePOST tolerates wide-area latencies and is typically lightly loaded. ePOST runs on nodes in residential or wireless networks and thus has limited bandwidth, whereas the network filesystem usually runs on a cluster with high-speed links. For this reason, we chose to evaluate each performance metric in the context of the application for which that metric is most critical.

We used Sun V20Z rack servers in a local cluster, consisting of dual 2.5 GHz Opteron CPUs connected with 1 Gbps switched Ethernet, running Linux 2.6.15. For the filesystem experiments, the workload was generated by a host in the same subnet. We compare the performance of the kernel-level NFS server with and without our user-level implementation of PeerReview.

For the multicast experiments, we generated a CBR stream of 300 Kbps, which is a common rate for streaming video to clients with broadband connections. We used a delay buffer of 10 seconds and counted blocks as lost if they were not received by this deadline. To obtain controlled conditions, we used a simple network emulator that forwards packets after a configurable delay. However, our implementation is complete and can also be run on a network testbed like PlanetLab.

Our ePOST experiment was driven by a one-day trace from a real ePOST deployment with 25 nodes/users, which was kindly provided by the authors of ePOST. The trace contains all messages sent on December 22, 2005, as well as all DHT operations (get/put) and all churn events (online/offline). We could not obtain a list of the exact keys and sizes of the objects in the DHT, but we were able to estimate its overall size. The experiment used our network emulator, although our code also runs on real networks.

In all experiments, we applied consistent hashing to choose a set of ψ witnesses per node, and we configured a high audit frequency of $T_{audit} = 10$ secs. The buffer delay T_{buf} was 100 ms for overlay multicast and 5 secs otherwise.

7.2 Fault injection experiments

In our first experiment, we inject a few faults into our systems. In the first scenario, three[3] faulty ePOST nodes F_1–F_3 try to censor an object O. First, a node A joins and inserts O into the DHT. To make it easy for the three nodes, we choose O's key such that F_1–F_3 store the replicas. Then A leaves the system and, 10 seconds later, another node B attempts to look up the object in the DHT. The request is routed to the faulty nodes, who reply that O does not exist.

Without PeerReview in place, lookups of O take a long time, because ePOST's archival store has to retrieve the missing object each time. Moreover, this state can persist indefinitely, because the system cannot identify and remove the faulty nodes[4]. With PeerReview, however, the faulty nodes are exposed after their first incorrect response. The effect is the same as if F_i had left the system: ePOST repairs its routing tables, new replicas of O are restored from ePOST's archival background store and subsequent lookups of object O succeed immediately. We note that this is possible even though A and B never communicate directly; they are never even online at the same time.

Our second experiment tests overlay multicast's response to freeloaders. We set up three multicasts, each with one source, 100 clients, 10 trees, and $\psi = 2$ witnesses per node. One of the clients either behaved as expected ("Cooperative") or freeloaded with a "Reluctant" strategy, forwarding content only when challenged by its witnesses, or freeloaded with a "Silent" strategy, refusing to forward any content. Without PeerReview, both strategies would enable the freeloader to receive all content without forwarding any content.

With PeerReview in place (Figure 4), the "Reluctant" strategy still enables the freeloader to receive most of the content in time but it must send more traffic than with the cooperative strategy because it cannot avoid forwarding each message eventually, and it must constantly answer challenges sent to it via its witnesses. The "Silent" strategy, by contrast, causes the freeloader to be exposed, so the other nodes stop sending content to it. Thus, both freeloading strategies are unattractive for the freeloader.

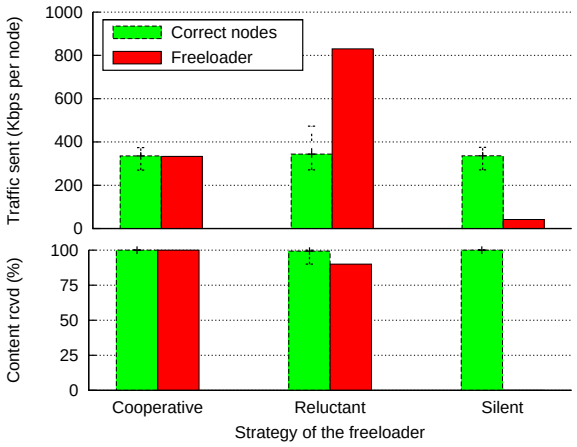

Figure 4: Multicast with a single freeloader: Impact on an average correct node (left bar) and on the freeloader (right bar). The error bars show minimum and maximum values. The freeloader uses different strategies, which are described in the text.

Run	Min	Avg	Max
nfs	66 μs	78 μs	99 μs
nfs+wrap	91 μs	113 μs	136 μs
peerreview-nosig	140 μs	144 μs	241 μs
peerreview-1024	1,591 μs	1,624 μs	1,938 μs

Table 1: Round-trip time for a NULL RPC.

7.3 Message latency

PeerReview increases the latency of message transfers because the sender must sign each message before transmitting it, and the receiver must check the signature before accepting the message. Also, both sides must append an entry to their log.

[3]ePOST stores three full replicas of each object in its DHT plus a number of erasure-coded fragments in its archival store. The archival store is accessed only if the object is not found in the DHT.

[4]Note that secure routing [11] would not solve this problem. The lookups are delivered to the right nodes, but these nodes do not respond correctly.

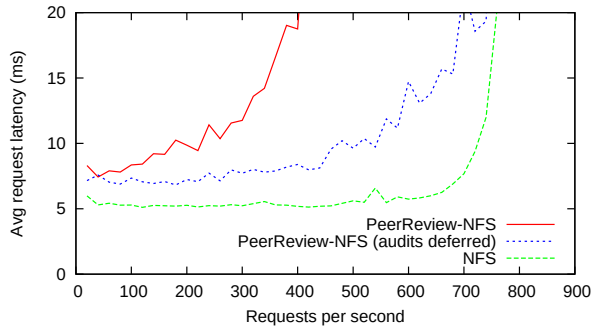

Figure 5: Throughput results for random 1 kB NFS read accesses over 10 GB of data.

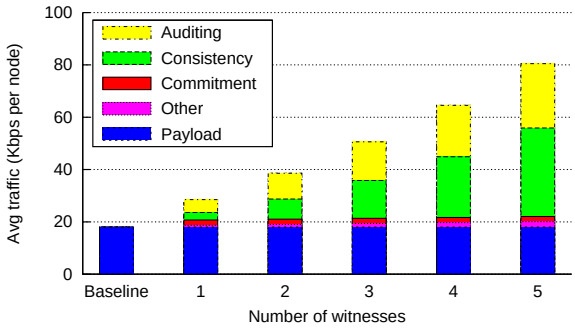

Figure 6: Average network traffic in ePOST.

To quantify the impact, we set up an experiment in which we measured the latency of an NFS NULL request (essentially a no-op). This is a worst-case scenario in terms of the impact on message latency: both the propagation delay and the processing time are very small, so the additional latency has a high impact. Table 1 shows our results. 'Nfs' is the bare NFS server; 'nfs+wrap' adds the cost of invoking the user-level wrapper for each RPC request and each response; and 'peerreview-nosig' adds the PeerReview wrapper with dummy signatures. The final result is for the full PeerReview protocol with 1024-bit RSA keys.

Our results show that the overhead is dominated by the time needed to generate and verify cryptographic signatures. Each RPC requires two messages, and for each message, a signature must be generated by the sender and verified by the receiver. This adds a constant delay that depends on the key length and the cryptographic algorithm used. For RSA-1024, the delay is about 1.5 ms per RPC, which matters for local-area applications with many small requests but is insignificant for wide-area applications. Moreover, if we use the more efficient ESIGN algorithm with 2048-bit keys, two signatures can be generated and verified in less than 250 μs on the same hardware. As an additional optimization, small messages could be signed in batches.

7.4 Throughput

In configurations where nodes are witnesses for each other, a node must replay and check requests directed to ψ other nodes, as part of its responsibility as a witness. Hence, we expect the average throughput of each node to drop to $\frac{1}{\psi+1}$ of its capacity.

To verify this, we deployed our network filesystem with four servers, using bare NFS in one experiment and adding PeerReview with $\psi = 2$ witnesses per server in another. Each server exported 10 GB worth of data. Then we sent random 1 kB read requests to each server at a constant rate, and we measured the average request latency. When the request rate reaches the server's capacity, we expect the latency to increase sharply. Figure 5 shows our results for a single server. As expected, throughput drops to approximately one-third with PeerReview enabled.

However, if the workload is bursty, PeerReview can temporarily increase its throughput by deferring audits to periods of low load, at the expense of a slightly higher time to detection. We repeated the previous experiment with short request bursts (see "audits deferred" in Figure 5) and found that under these conditions, the PeerReview-enabled servers could almost reach the full throughput of the unmodified NFS server. This is important because, in practice, most systems experience bursty and/or diurnal workloads. With a sufficiently high T_{audit}, these systems can profit from accountability without having to pay for it with a lower throughput.

7.5 CPU overhead

If a PeerReview-enabled node handles many very short requests, the cryptographic operations required for each message can form a bottleneck. Suppose verifying one signature and creating another takes c seconds. Then a node cannot respond to more than $\frac{1}{c}$ incoming messages per second, even if the actual processing takes no time at all. However, concurrent signature generations are trivially parallelizable, so the problem can be overcome by using more CPUs or a CPU with multiple cores.

To quantify this, we saturated a PeerReview-enabled NFS server with NULL RPCs and found that it could handle 3,900 RPCs per second with one of the two CPUs disabled. When we enabled the second CPU, the throughput rose to 7,500 RPCs per second, a 92% increase. This shows that PeerReview can easily take advantage of additional CPUs. Because multi-core CPUs will become the standard, we expect that most applications will not be limited by the cost of asymmetric cryptography.

7.6 Network traffic and disk space

PeerReview requires network bandwidth for auditing and consistency checking, and disk space to store logs and evidence. As discussed in Section 4, the overhead depends mainly on the number ψ of witnesses per node. To quantify this, we used our ePOST application and the trace from a deployment with real users. We replayed the trace several times, varying the parameter ψ, and we measured the average number of bytes sent per node per hour. We also measured the average log size.

Figure 6 shows our results. With $\psi = 5$ witnesses, the average network traffic of an ePOST node increased from 18.0 Kbps to 80.5 Kbps. The overhead is initially dominated by the audits, but their share grows only with $O(\psi)$, whereas the consistency protocol's share grows with $O(\psi^2)$. We also note that our optimizations from Section 5.1 are effective; without them, the audit traffic would be at least ψ times the payload.

PeerReview adds a considerable amount of network traffic. However, the bulk of the additional network traffic is caused by auditing and consistency. With the typical bursty or

diurnal workloads that many systems experience, both of these tasks can be deferred to periods of low load. Thus, they do not necessarily reduce the peak throughput of the system, nor do they increase the peak network traffic.

On average, each node's log grew at a rate of 14.4 MB per hour, independent of the parameter ψ. This means that a year's log can be kept in ePOST in less than 125 GB of disk storage per node. In the multicast experiment, the log grew at a rate of 147 MB per hour. However, this includes 135 MB of streamed content, which could be safely discarded after a few minutes.

7.7 Scalability

In a system with a large number of nodes, it can be difficult to ensure an absolute bound on the number of faulty nodes. Instead, a bound on the fraction φ of faulty nodes is often assumed. To maintain PeerReview's completeness guarantees, we must choose $\psi = \lceil \varphi N \rceil$ in this case; otherwise, all witnesses of a faulty node could also be faulty. Because the consistency protocol creates a per-node overhead that grows with $O(\psi^2)$, we arrive at an overhead of $O(N^2)$. Extrapolating from the results in Figure 6, and assuming that each node has a broadband connection with an upstream bandwidth of 512 Kbps, ePOST should be able to scale to a configuration with $\psi = 13$ witnesses. Assuming a bound of $\varphi = 10\%$ faulty nodes, the system would scale to about 130 nodes with broadband connections.

However, scalability can be improved considerably by relaxing PeerReview's guarantees as described in Section 4.11. We implemented smaller witness sets and randomized consistency checking for both ePOST and overlay multicast. Figure 7 shows how the per-node traffic grows with the system size N if we assume a bound of $\varphi = 10\%$ faulty nodes and $P_f = P_m = 10^{-6}$. When both modifications are combined, ePOST scales to over $10,000$ nodes and overlay multicast to at least $1,000$ nodes with broadband connections.

7.8 Additional benefits

While experimenting with PeerReview, we discovered some unexpected benefits. PeerReview turned out to be a useful tool for tracking down race conditions and other 'heisenbugs'. These bugs are exposed by PeerReview because they typically occur either during execution or during replay, but not both. In this case, PeerReview conveniently produces evidence that can be used to reproduce the bug. We found several of these bugs in the original ePOST codebase, and these have been confirmed by the ePOST developers.

Another unexpected benefit was the ability to check for protocol conformance in a system that contains more than one implementation of the same protocol. Because each node uses its own implementation as a reference when witnessing another node, deviations from the protocol (e.g. in corner cases) can be detected. In this case, PeerReview provides evidence that can be used to reproduce the different behavior.

8. CONCLUSION

In this paper, we have described PeerReview, a general and practical system that provides accountability and fault detection for distributed systems. PeerReview guarantees the eventual detection of all Byzantine faults whose effects are observed by a correct node. Verifiable evidence of a fault is irrefutably linked to a faulty node, while correct nodes can defend themselves against false accusations. We applied PeerReview to three sample applications and experimentally evaluated the resultant systems. Our experiments indicate that PeerReview is practical and useful for a range of distributed applications. Despite its strong guarantees, PeerReview scales to moderately large systems; however, by relaxing completeness in favor of a probabilistic detection guarantee, PeerReview can scale to very large systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mike Schroeder, who provided helpful feedback on the SOSP version of this paper. Thanks to Alan Mislove and James Stewart for their help with ePOST. This research was supported in part by US National Science Foundation grants ANI-0225660 and CNS-0509297.

9. REFERENCES

[1] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. SOSP'05*, Oct 2005.

[2] L. Alvisi, D. Malkhi, E. T. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Trans. Parallel Distrib. Syst.*, 12(9):996–1007, 2001.

[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4033: DNS security introduction and requirements. http://www.ietf.org/rfc/rfc4033.txt, Mar 2005.

[4] K. Argyraki, P. Maniatis, O. Irzak, and S. Shenker. An accountability interface for the Internet. In *Proc. 14th ICNP*, Oct 2007.

[5] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid. Recommendation for key management. NIST Special Publication 800-57, revised Mar 2007.

[6] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. VMCAI'04*, Jan 2004.

[7] R. A. Bazzi and G. Neiger. Simplifying fault-tolerance: Providing the abstraction of crash failures. *J. ACM*, 48(3):499–554, 2001.

[8] G. Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, Nov 1987.

[9] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.

[10] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT LCS, Jan 2001.

[11] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proceedings of OSDI'02*, Boston, MA, Dec 2002.

[12] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in cooperative environments. In *Proceedings of SOSP'03*, Oct 2003.

[13] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[14] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Trans. Comput. Syst.*, 21(3):236–269, 2003.

[15] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar 1996.

[16] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. SOSP*, Oct 2007.

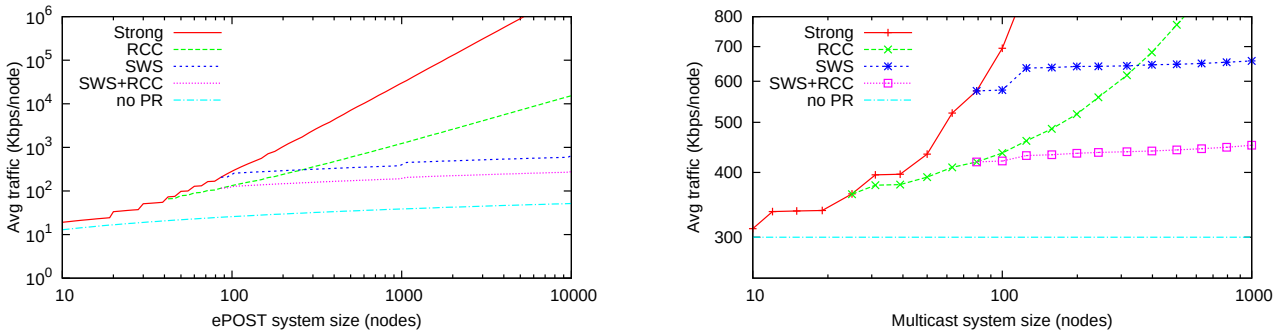

Figure 7: Scalability for ePOST (left) and overlay multicast (right) with strong guarantees and with probabilistic guarantees, using smaller witness sets (SWS) and/or randomized consistency checking (RCC). The ePOST numbers are extrapolated from Figure 6, while the other results are from the multicast experiment.

[17] B. A. Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Trans. Computers*, 37(12):1541–1553, 1988.

[18] L. P. Cox and B. D. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proceedings of SOSP*, Oct 2003.

[19] D. E. Denning. An intrusion-detection model. *IEEE Trans. on Software Engineering*, 13(2):222–232, 1987.

[20] R. Dingledine, M. J. Freedman, and D. Molnar. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, chapter Accountability. O'Reilly and Associates, 2001.

[21] J. R. Douceur. The Sybil attack. In *Proceedings of IPTPS*, Mar 2002.

[22] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC*, pages 71–87, 1999.

[23] T. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS support and applications for trusted computing. In *Proc. HotOS'03*, May 2003.

[24] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proceedings of HotDep*, Nov 2006.

[25] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. NSDI*, May 2005.

[26] M. Jelasity, A. Montresor, and O. Babaoglu. Detection and removal of malicious peers in gossip-based protocols. In *Proceedings of FuDiCo*, Jun 2004.

[27] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The EigenTrust algorithm for reputation management in p2p networks. In *Proc. 12th International WWW Conference*, May 2003.

[28] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *STOC*, 1997.

[29] S. T. Kent, C. Lynn, J. Mikkelson, and K. Seo. Secure border gateway protocol (S-BGP) - real world performance and deployment issues. In *NDSS*, 2000.

[30] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[31] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs using execution monitoring. In *Proceedings of the 10th Computer Security Application Conference*, Dec 1994.

[32] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Prog. Lang. Syst.*, 6(2):254–280, 1984.

[33] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[34] B. W. Lampson. Computer security in the real world. In *Proc. Annual Computer Security Applications Conference*, Dec 2000.

[35] P. Laskowski and J. Chuang. Network monitors and contracting systems: competition and innovation. In *Proc. SIGCOMM'06*, pages 183–194, Sep 2006.

[36] J. Li, M. Krohn, D. Mazières, and D. Sasha. Secure untrusted data repository (SUNDR). In *Proceedings of OSDI 2004*, Dec 2004.

[37] D. Malkhi and M. K. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW*, pages 116–125, 1997.

[38] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proc. of the 11th USENIX Security Symposium*, Jan 2002.

[39] N. Michalakis, R. Soulé, and R. Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proc. NSDI'07*, Apr 2007.

[40] A. Mislove. POST: A decentralized platform for reliable collaborative applications. Master's thesis, Rice Unversity, Houston, TX, Dec 2004.

[41] A. Mislove, A. Post, A. Haeberlen, and P. Druschel. Experiences in building and operating ePOST, a reliable peer-to-peer application. In *Proceedings of EuroSys 2006*, Apr 2006.

[42] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. OSDI'02*, Dec 2002.

[43] A. Nandi, T.-W. Ngan, A. Singh, P. Druschel, and D. Wallach. Scrivener: Providing incentives in cooperative content distribution systems. In *Proc. Middleware Conference*, Nov 2005.

[44] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proc. USENIX Security Symposium*, pages 217–228, 1998.

[45] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed systems. In *PODC*, pages 248–262, 1988.

[46] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proc. USITS'03*, Mar 2003.

[47] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[48] PeerReview project homepage. `http://peerreview.mpi-sws.mpg.de/`.

[49] H. V. Ramasamy, A. Agbaria, and W. H. Sanders. A parsimonious approach for obtaining resource-efficient

and trustworthy execution. *IEEE Trans. on Dependable and Secure Comp.*, 4(1):1–17, Jan 2007.
[50] P. Reynolds, O. Kennedy, E. G. Sirer, and F. B. Schneider. Securing BGP using external security monitors. Technical Report 2006-2065, Cornell University, Computing and Inform. Science, Dec 2006.
[51] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of Middleware'01*, Nov 2001.
[52] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proceedings of SOSP'01*, Lake Louise, Alberta, Canada, Oct 2001.
[53] A. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC 2001*, UCL, London, Nov 2001.
[54] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.
[55] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *Proc. 7th USENIX Security Symposium*, pages 53–62, Jan 1998.
[56] A. Singh, T.-W. Ngan, P. Druschel, and D. Wallach. Eclipse attacks on overlay networks: Threats and defenses. In *Proceedings of INFOCOM*, Apr 2006.
[57] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.
[58] Sun Microsystems Inc. RFC 1094: Network file system protocol specification. http://www.ietf.org/rfc/rfc1094.txt, Mar 1989.
[59] M. Waldman and D. Mazières. Tangler: a censorship-resistant publishing system based on document entanglements. In *Proc. 8th ACM conf. on Comp. and Comm. Security*, pages 126–135, 2001.
[60] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *Proc. NSDI*, May 2006.
[61] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP*, pages 253–267, Oct 2003.
[62] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for internet services. In *ACM SIGOPS European Workshop*, 2004.
[63] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of HotDep*, Jun 2005.
[64] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *FAST*, 2007.

APPENDIX

A. PROOF OF CORRECTNESS

Our goal is to prove that PeerReview has the properties defined in Section 3.5 of this paper, namely:

- **Eventual strong completeness:** (1) Eventually, every detectably ignorant node is suspected forever by every correct node, and (2) if a node i is detectably faulty with respect to a message m, then eventually, some faulty accomplice of i (with respect to m) is exposed or forever suspected by every correct node.
- **Eventual strong accuracy:** (1) No correct node is forever suspected by a correct node, and (2) no correct node is ever exposed by a correct node.

We begin by giving some additional details of the PeerReview algorithm in Section A.1. Then we prove each of the above four claims in Sections A.2 and A.3.

A.1 Validating evidence

In the context of PeerReview, *evidence* is a piece of data that supports a statement about the behavior of a particular node. There are three different kinds of evidence for a node i. A *challenge* `chal(x)` states that i has refused to respond to request `x` and therefore may be detectably faulty or detectably ignorant. A *response* `resp(x)` states that i has in fact responded to request `x` and thus refutes the corresponding challenge. Finally, a *proof* `proof(x)` states that i is detectably faulty. A proof also includes some information about the nature of the fault.

A faulty node could try to incriminate a correct node by forging evidence. Therefore, every node must establish that the evidence is *valid* before using it. A list of all types of evidence about a node i, including the conditions under which they are valid, is given below:

- `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` is evidence that node i is refusing to return the log segment $\{e_x, \ldots, e_y\}$. The challenge is valid iff both $a_{i,x}$ and $a_{i,y}$ are authenticators signed by i, and $y > x$.
- `resp(audit,`$a_{i,x}$`,`$a_{i,y}$`,`L`)` shows that i has properly responded to the challenge above. The response is valid iff `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` is valid, $L = \{e_x, \ldots, e_y\}$ is a well-formed log segment signed by i, and the hashes in $a_{i,x}$ and $a_{i,y}$ match those of the corresponding entries in L.
- `chal(send,`m`)` is evidence that i is refusing to accept a message m. The challenge is valid iff $receiver(m) = i$.
- `resp(send,`m`,`$a_{i,x-1}$`,`$a_{i,x}$`)` shows that i has in fact accepted the message m. The response is valid iff `chal(send,`m`)` is valid, both $a_{i,x-1}$ and $a_{i,x}$ are signed by i, and $a_{i,x}$ validates[5] a log entry $send_i(ack(m))$.
- `proof(inconsistent,`$a_{i,x}$`,`L`)` shows that i is maintaining several inconsistent histories. The proof is valid iff L is signed by i and contains an entry e_x that has the same sequence number as $a_{i,x}$, but a different hash.
- `proof(invalid,`c`,`L`)` shows that i's history does not conform to its state machine A_i. The proof is valid iff L is signed by i, c matches the first checkpoint in L, and L fails the conformance check for A_i.

A piece of evidence is *invalid* if it is not valid according to these rules. Invalid evidence is not considered further by PeerReview.

Each node j maintains an *evidence set* ε_{ij} for every other node i. For simplicity, we will assume that if j and k are correct nodes, then ε_{ij} and ε_{ik} are eventually consistent, so we can treat them as a single set ε_i. In practice, this can be achieved e.g. by allowing correct nodes to gossip about evidence.

To save space, nodes immediately discard invalid evidence. Also, if ε_i contains both a valid challenge c and a matching, valid response $r(c)$, the nodes may eventually discard both.

A node j generates failure indications for another node i as follows: If ε_{ij} contains a valid challenge but no matching, valid response, then j outputs $suspected_i$. If ε_{ij} contains a valid proof, then j outputs $exposed_i$. In all other cases, j outputs $trusted_i$.

[5]To check this, the previous hash h_{x-1} is required, which can be taken from $a_{i,x-1}$.

A.2 Eventual strong completeness

Theorem 1 *Eventually, every detectably ignorant node is suspected forever by every correct node.*

PROOF. Assume the opposite, i.e. there is a detectably ignorant node i and a correct node k such that, for every time t, there is another time $t' > t$ at which k does not suspect i. Since i is detectably ignorant, we know that it is not detectably faulty, and that there exists some message m that a correct node j has sent to it, but i has never sent another message m' such that $receive_i(m)$ appears in $\mathcal{H}(m')$.

Since acknowledgments are mandatory, j must have resent m several times, then it must eventually have given up and added a challenge `chal(send,`m`)` to ε_i, so eventually all correct nodes must have started suspecting i. Let t_1 be the time k is first notified of the challenge. By our assumption, we know that there is a time $t_2 > t_1$ at which k does not suspect i. But this is only possible if the challenge has been refuted because `resp(send,`m`,`$a_{i,x-1}$`,`$a_{i,x}$`)` has been added to ε_i, and because $a_{i,x}$ validates a log entry $send_i(ack(m))$. By definition, i is not detectably faulty, so $receive_i(m)$ must have preceded $send_i(ack(m))$ in $\mathcal{H}(ack(m))$, which means that i cannot be detectably ignorant. This is a contradiction. □

Recall that nodes commit to the contents of their logs by publishing authenticators, or signed hashes of their logs. We say that a node j is *notified* of a history $\mathcal{H}(m)$ of another node i if it receives an authenticator $a_{i,x}$ for a log that corresponds to $\mathcal{H}(m)$. The authenticator allows j to obtain $\mathcal{H}(m)$ from i, which j can then validate against the hash value in the authenticator. If i does not comply, j can use $a_{i,x}$ in an `audit` challenge and thus cause all correct nodes to suspect i.

As described in Section 4.7, the notification does not have to be performed by i itself. Since an authenticator is included with each log entry $receive_i(m)$, other nodes can extract the authenticators during audits and then perform the notification on i's behalf.

Lemma 1 *If a message m is observable by a correct node c via a chain of messages $m_1, \ldots, m_k$, then either each correct node is notified of $\mathcal{H}(m_x)$ for all $1 \leq x \leq k$, or some $sender(m_x)$ is exposed or forever suspected by all correct nodes.*

PROOF. Note that, by the definition of observability, $m_1 = m$, $receiver(m_k) = c$, and for all $2 \leq j \leq k$, $receive(m_{j-1})$ belongs to $\mathcal{H}(m_j)$.

We begin by observing that, since c is correct, it will certainly notify all other nodes of $\mathcal{H}(m_k)$. Thus, it is sufficient if we can show that, if all nodes are notified of $\mathcal{H}(m_x)$ (for some $x > 1$), then all nodes are also notified of $\mathcal{H}(m_{x-1})$, or $sender(m_x)$ is either exposed or forever suspected.

If $sender(m_x)$ is exposed or forever suspected, the claim follows immediately. Otherwise, it must have fully cooperated with all correct nodes. If it had refused to answer an audit, an `audit` challenge would eventually have been added to its evidence set, and it would have been suspected forever by all correct nodes. Moreover, we know that each correct node must have seen a *valid* log; otherwise that node would have added an `invalid` proof of misbehavior the evidence set, and $sender(m_x)$ would have been exposed.

Let h_x be the history of $sender(m_x)$ as observed by some correct node c_x. We know that $\mathcal{H}(m_x)$ ends with an entry $send_{receiver(m_x)}(m_x)$ (otherwise $receiver(m_x)$ would never have accepted m_x). Therefore, we know that h_x must also contain $send_{receiver(m_x)}(m_x)$, because otherwise c_x would have added an `inconsistent` proof of misbehavior to its evidence set, and $sender(m_x)$ would have been exposed. But if h_x contains $send_{receiver(m_x)}(m_x)$, it must also contain an entry $receive(m_{x-1})$ (if this was not the case, m_k would not be causally connected to m_{k-1}). When auditing this entry, c_x extracts $\mathcal{H}(m_{x-1})$ and notifies all the other nodes.

The claim follows by reverse induction over x. □

Theorem 2 *If a node i is detectably faulty with respect to some message m, then eventually, either i itself or some faulty accomplice of i with respect to m is exposed or forever suspected by every correct node.*

PROOF. Assume the contrary, i.e. that there is a node i that is detectably faulty with respect to some message m, but neither i nor any of its faulty accomplices with respect to m is exposed or forever suspected by some correct node j. Since correct nodes are not exposed or forever suspected under any circumstances (see Section A.3), it follows that *no* node along the path of causality is exposed or forever suspected.

By Lemma 1, we know that under these circumstances, all nodes must eventually be notified of $\mathcal{H}(m)$. Let c be some correct node. We know that i has cooperated with c and responded to all of its audits, since c does not forever suspect i. We also know that i's history h, as seen by c, is valid, since c does not expose i. For the same reason, we know that $\mathcal{H}(m)$ is consistent with h, so this cannot be the reason i is detectably faulty.

The only other potential reason i could be detectably faulty is that it has sent some other message m' that is observable by some correct node c', such that $\mathcal{H}(m)$ is inconsistent with $\mathcal{H}(m')$. But, by Lemma 1, we know that either some node along the path of m' is exposed or forever suspected, or all correct nodes are eventually notified of $\mathcal{H}(m')$ as well. In the first case, i is faulty with respect to m', and a faulty accomplice is exposed or forever suspected, so the theorem follows. In the second case, we know that c has not exposed i, so $\mathcal{H}(m')$ must be consistent with h and therefore also with $\mathcal{H}(m)$. This is a contradiction. □

A.3 Eventual strong accuracy

Theorem 3 *No correct node is forever suspected by a correct node.*

PROOF. Assume the opposite, i.e. there is a correct node i that is forever suspected by another correct node k after some time t_1. Let $t_2 > t_1$ be the first time k checks ε_i. Since k still suspects i after t_2, ε_i must have contained some valid, unrefuted challenge c. Moreover, we know that k must have challenged i with c, but i did not provide a valid response.

The challenge c can either be `chal(audit,`$a_{i,x}$`,`$a_{i,y}$`)` or `chal(send,`m`)`. If it is an `audit` challenge, it can only be valid if both $a_{i,x}$ and $a_{i,y}$ are authenticators signed by i, and $y > x$ (recall that signatures cannot be forged). But since i is correct, its log is well-formed, so the authenticators $a_{i,x}$ and $a_{i,y}$ must correspond to existing log entries e_x and e_y with the corresponding hash values. Thus, i can extract the

log segment $L = e_x, \ldots, e_y$ and use it to construct a valid response.

Now assume c is a `send` challenge. There are two cases: Either i has previously received m, or it has not. If the former holds, i, being correct, must have an entry $send_i(ack(m))$ in its log. Using the authenticator $a_{i,x}$ covering this entry, it can construct a valid response. If i has not yet received m, it can accept it now, which produces the required log entries and again enables i to construct a valid response.

But if i can construct a valid response, it would have done so and replied to k. Eventually, this reply would have been received by k, so it cannot have suspected i forever. This is a contradiction. □

Theorem 4 *No correct node is ever exposed by a correct node.*

PROOF. Assume the opposite, i.e. there is a correct node i that is exposed by another correct node j. This means that ε_i contains a proof of misbehavior p, which is valid (because j, being correct, would have checked this before exposing i).

The proof p can either be `proof(inconsistent,`$a_{i,x}$`,`L`)`, or `proof(invalid,`c`,`L`)`. In the first case, $a_{i,x}$ and L must be signed by i, and L must contain an entry e_x that has the same sequence number as $a_{i,x}$, but a different hash. But i is correct, so it never uses the same sequence number twice. Thus, the second case must apply. Since p is valid, L must be signed by i, c must match the first checkpoint in L, and L must fail the conformance check for A_i. But i is correct, so it must have faithfully recorded its inputs and outputs in the log, and since A_i is deterministic by assumption, it must have produced the same outputs as the ones in L, so L cannot fail the conformance check. This is a contradiction. □