Prime: Byzantine Replication Under Attack

Yair Amir, Brian Coan, Jonathan Kirsch, John Lane

Technical Report CNDS-2009-4
May 2009

Abstract—Existing Byzantine-resilient replication protocols satisfy two standard correctness criteria, safety and liveness, in the presence of Byzantine faults. In practice, however, faulty processors can, in some protocols, significantly degrade performance by causing the system to make progress at an extremely slow rate. While "correct" in the traditional sense, systems vulnerable to such performance degradation are of limited practical use in adversarial environments. This paper argues that techniques for mitigating such performance attacks are needed to bridge this "practicality gap" for intrusion-tolerant replication systems. We propose a new performance-oriented correctness criterion, and we show how failure to meet this criterion can lead to performance degradation. We present a new Byzantine replication protocol that achieves the criterion and evaluate its performance in fault-free configurations and when under attack.

1 INTRODUCTION

EXISTING Byzantine-resilient state machine replication (SMR) protocols satisfy two standard correctness criteria in the presence of Byzantine faults: safety and liveness. Safety means that two servers remain consistent replicas of one another, while liveness means that each update is executed eventually. Since no asynchronous Byzantine agreement protocol can always be both safe and live [1], systems requiring strong consistency semantics are usually designed to meet safety in all executions, while guaranteeing liveness only during periods of sufficient synchrony and connectivity [2] or in a probabilistic sense [3], [4].

Designers of practical Byzantine-resilient replication systems recognize that real systems are not completely asynchronous. Rather, these systems exhibit extended periods of stability (synchrony), possibly interspersed with periods of instability. Realistic Byzantine-resilient replication systems generally guarantee liveness in a sufficiently stable subset of the set of all asynchronous executions. In this paper we observe that during stable periods, the system can satisfy much stronger performance guarantees. Thus, when the network is stable, there is a potential gap in the type of performance that is promised by existing protocols (i.e., eventual execution of each update) and the type of performance that is attainable.

In Byzantine environments, faulty processors can exploit this gap to degrade system performance to a level far below what would be achievable with only correct processors. Specifically, a small number of faulty processors can cause the system to make progress at an extremely slow rate. While "correct" in the traditional sense (both safety and liveness are met), systems vulnerable to such performance degradation are of limited practical use in adversarial environments.

We experienced this problem first hand during a red-team experiment conducted on our Steward system [5]. Although the system survived all of the tests according to the metrics of safety and liveness, we observed that it was slowed down to twenty percent of its potential performance in one experiment. After analyzing the attack, we found that we could in fact slow the system down to roughly one percent of its potential performance. Thus, our provably correct system, which achieves high performance in fault-free configurations, could be made effectively unusable in practice under a relatively simple attack. This experience led us to conclude that liveness is a necessary but insufficient correctness criterion for achieving high performance Byzantine replication under attack. This paper argues that new *performance-oriented* criteria are needed.

Preventing the type of performance degradation experienced by Steward requires addressing what we call *Byzantine performance failures*. Previous work focused on Byzantine failures in the value domain (where faulty processors send incorrect or conflicting messages) and the time domain (where messages from faulty processors do not arrive within protocol timeouts, if at all). Processors exhibiting performance failures, however, send correct messages slowly but without triggering protocol timeouts; they are thus correct in both of the traditional domains, despite having the potential to significantly degrade performance. Performance failures have been considered in benign environments [6], [7]. To the best of our knowledge, we are the first to (1) propose a useful performance-oriented metric to evaluate Byzantine protocols and (2) present a SMR protocol that performs well according to this metric.

- *Yair Amir, Jonathan Kirsch, and John Lane are with the Department of Computer Science, Johns Hopkins University, 3400 N. Charles St., Baltimore, MD 21218. Email: {yairamir, jak, johnlane}@cs.jhu.edu.*
- *Brian Coan is with Telcordia Technologies, One Telcordia Drive, Piscataway, NJ 08854. Email: coan@research.telcordia.com.*
- *This publication was supported by Grants 0430271 and 0716620 from the National Science Foundation. Its contents are solely the responsibility of the authors and do not necessarily represent the official view of Johns Hopkins University or the National Science Foundation.*

Byzantine protocols whose progress is driven by messages from a large number of correct processors (e.g., [3], [8]) are less vulnerable to performance degradation due to performance failures. The voting in such protocols masks performance failures, in addition to value and timing failures, because no collection of faulty processors can prevent the correct processors from moving forward. For efficiency, however, other protocols rely on select processors to perform certain tasks correctly and in a timely manner, reducing the number of messages that must be sent in the common case. These protocols typically use cryptographic tools and timeouts to restrict the adversary in the value and time domains, respectively, but they do not address performance failures.

In this paper we focus on this latter class of Byzantine SMR protocols, which we refer to as *leader-based* protocols. These protocols (e.g., [5], [9], [10], [11], [12], [13], [14]) rely on a leader to coordinate the global ordering and are thus vulnerable to performance degradation caused by a slow leader. The problem is magnified in environments (such as wide-area networks) where it is difficult to predict the type of performance that should be expected of the leader. We demonstrate this vulnerability through analysis and experimental evaluation of BFT [9], the first leader-based Byzantine fault-tolerant SMR protocol to achieve practical performance in fault-free executions.

By applying the understanding gained from our experience with BFT, we developed a new Byzantine fault-tolerant SMR protocol, Prime (Performance-oriented Replication In Malicious Environments) [15], resilient to performance degradation under attack. Prime has two key properties: (1) The resources required by the leader for global ordering are bounded and independent of system throughput, enabling non-leader servers to aggressively monitor the leader's performance, and (2) Non-leader servers compute a threshold level of acceptable performance, which is a function of current network latencies, against which they judge the leader. Prime meets a new performance-oriented correctness criterion, BOUNDED-DELAY, which makes a stronger guarantee than traditional liveness criteria. We present experimental results showing that Prime performs competitively with BFT in fault-free configurations and performs an order of magnitude better when under attack. Our results show that the performance of Prime when under attack is within a reasonable factor of its fault-free performance.

The remainder of this paper is presented as follows. Section 2 presents our system model and describes the service properties provided by our system. Section 3 describes the vulnerabilities of existing leader-based protocols to performance degradation under attack, using BFT as a case study. We present the Prime protocol in Section 4 and the Prime view change protocol in Section 5. In Section 6, we sketch the proof that Prime meets BOUNDED-DELAY. Section 7 presents experimental results for our new system. Section 8 details related work, and Section 9 concludes the paper.

2 SYSTEM MODEL AND SERVICE PROPERTIES

We consider a system consisting of N servers, which communicate by passing messages. Each server is uniquely identified from the set $\mathcal{R} = \{1, 2, \ldots, N\}$. We assume a Byzantine fault model. Servers are either *correct* or *faulty*; correct servers follow the protocol specification, while faulty servers can deviate from the protocol specification arbitrarily. We employ digital signatures, and we make use of a cryptographic hash function to compute message digests. We denote a message m signed by server i as $\langle m \rangle_{\sigma_i}$, and we denote a digest of m as D(m). We assume that all adversaries, including faulty servers, are computationally bounded such that they cannot subvert these cryptographic mechanisms.

The consistency of our new protocol, Prime, is given in the following two properties:

DEFINITION 2.1: SAFETY: If two correct servers execute the i^{th} update, then these updates are identical.

DEFINITION 2.2: VALIDITY: Only an update that was proposed by a client may be executed.

Prime guarantees safety and validity in all executions, including those in which the network is asynchronous and may drop or duplicate messages. Like existing leader-based Byzantine replication protocols, Prime guarantees liveness only in executions in which the network eventually meets certain stability conditions, which we now state. In what follows, K_{Lat} is a known network-specific constant accounting for latency variability.

DEFINITION 2.3: PRIME-STABILITY: There is a time after which the following condition holds for a set of at least $2f + 1$ correct servers (the stable servers):

- For each pair of stable servers r and s, there exists a value Min_Lat(r, s), unknown to the servers, such that if r sends a message to s, it will arrive with delay $\Delta_{r,s}$, where Min_Lat(r, s) $\leq \Delta_{r,s} \leq$ Min_Lat(r, s) $* K_{Lat}$.

In those executions in which PRIME-STABILITY is met, Prime guarantees the following liveness property:

DEFINITION 2.4: PRIME-LIVENESS: If a stable server initiates an update, all stable servers will eventually execute the update.

PRIME-LIVENESS is similar to the liveness guarantees provided by existing leader-based protocols (except that PRIME-LIVENESS contains a stronger degree of stability). While it is critical to guarantee that in those executions that are sufficiently stable each update is eventually executed, such liveness properties do not guarantee how quickly the updates are executed when the network is

stable. Systems that solely meet liveness thus provide a very weak performance-related guarantee.

For this reason, in those executions in which PRIME-STABILITY is met, Prime also provides a stronger performance guarantee, which we call BOUNDED-DELAY:

DEFINITION 2.5: BOUNDED-DELAY: There exists a time after which the update latency for any update initiated by a stable server is upper-bounded.

Prime achieves BOUNDED-DELAY in those executions in which PRIME-STABILITY is met, assuming the system is not overloaded (i.e., given load beyond its maximum throughput) and when correct servers have sufficient bandwidth with which to communicate. Indeed, no system (even in benign environments) can provide latency guarantees when these conditions are not met due to necessary queuing delays. Our current protocol requires knowledge of this minimal level of bandwidth to ensure that these assumptions are met. We believe that adaptively setting the bandwidth consumed by correct servers is an important open problem for Byzantine-resilient systems. Section 6 provides an analysis of the bound provided by Prime.

We remark that resource exhaustion denial of service attacks may cause PRIME-STABILITY to be violated for the duration of the attack. However, such attacks fundamentally differ from the attacks that are the focus of this paper, where malicious leaders can slow down the system without triggering defense mechanisms (see Section 3). Handling resource exhaustion attacks is a difficult problem that is orthogonal and complementary to the solution strategies considered in this paper.

3 CASE STUDY: BFT UNDER ATTACK

In this section we present a theoretical analysis of BFT [9], a leader-based Byzantine SMR protocol, when under attack. We chose BFT because (1) it is the standard protocol to which other Byzantine protocols are often compared, (2) many of the attacks that can be applied to BFT (and the corresponding lessons learned) also apply to other leader-based protocols, and (3) its implementation was publicly available. BFT achieves high throughputs in fault-free configurations or when servers exhibit only benign faults. We first provide background on BFT and then describe two attacks that can be used to significantly degrade its performance when under attack. We present experimental results validating the analysis in Section 7.

BFT assigns a total order to client updates. The protocol requires $3f + 1$ servers, where f is the maximum number of servers that may be Byzantine. An elected leader coordinates the protocol by assigning sequence numbers to updates. If a server suspects that the leader has failed, it votes to replace it. When $2f+1$ servers vote to replace the leader, a view change occurs, in which a new leader is elected and servers collect information regarding pending updates so that progress can safely resume in a new view.

A client sends its updates directly to the leader. The leader assigns a sequence number to the update and proposes the assignment to the rest of the servers. It sends a PRE-PREPARE message, which contains the view number, the assigned sequence number, and the update itself. Upon receiving the PRE-PREPARE, a non-leader server accepts the proposed assignment by broadcasting a PREPARE message. The PREPARE message contains the view number, the assigned sequence number, and a digest of the update. When a server collects the PRE-PREPARE and $2f$ corresponding PREPARE messages, it broadcasts a COMMIT message. A server globally orders the update when it collects $2f + 1$ COMMIT messages. Each server executes globally ordered updates according to sequence number. A server sends a reply to the client after executing the update.

3.1 Attack 1: Pre-Prepare Delay

A malicious leader can introduce latency into the global ordering path simply by waiting some amount of time after receiving an update before sending it in a PRE-PREPARE message.The amount of delay a leader can add without being detected as faulty is dependent on (1) the way in which non-leaders place timeouts on updates they have not yet executed and (2) the duration of these timeouts.

A malicious leader can ignore updates sent directly by clients. If a client's timeout expires before receiving a reply to its update, it broadcasts the update to all servers, which forward the update to the leader. Each non-leader server maintains a FIFO queue of pending updates (i.e., those updates it has forwarded to the leader but not yet executed). A server places a timeout on the execution of the first update in its queue; that is, it expects to execute the update within the timeout period. If the timeout expires, the server suspects the leader is faulty and votes to remove it from power. When a server executes the first update in the queue, it restarts the timer if the queue is not empty. Note that a server does not stop the timer if it executes a pending update that is not the first in the queue. The duration of the timeout is dependent on its initial value (which is implementation and configuration dependent) and the history of past view changes. Servers double the value of their timeout each time a view change occurs. The specification of BFT does not provide a mechanism for reducing timeout values.

BFT's queueing mechanism ensures fairness by guaranteeing that each update is eventually ordered. However, it also allows the leader to significantly delay the ordering of an update without being replaced. To stay in power, the leader must prevent $f + 1$ correct servers from voting to replace it. Thus, assuming a timeout value of TO, a malicious leader can use the following attack: (1) Choose a set S of $f + 1$ correct servers, (2) For each server $r \in S$, maintain a FIFO queue of the updates

forwarded by r, and (3) For each such queue, send a PRE-PREPARE containing the first update on the queue every $TO - \epsilon$ time units. This guarantees that the $f+1$ correct servers in S execute the first update on their queue each timeout period. If these updates are all different, the fastest the leader would need to introduce updates is at a rate of $f+1$ per timeout period. In the worst case, the $f+1$ servers would have identical queues, and the leader could introduce one update per timeout.

This attack exploits the fact that non-leader servers place timeouts only on the first update in their queues. To understand the ramifications of placing a timeout on *all* pending updates, consider the following scenario: Non-leader server s simultaneously initiates n updates. If server s sets a timeout on all n updates, then s will suspect the leader if the system fails to execute n updates per timeout period. Since the system has a maximal throughput, if n is sufficiently large, s will suspect a correct leader. The fundamental problem is that correct servers have no way to assess the rate at which a correct leader can coordinate global ordering.

3.2 Attack 2: Timeout Manipulation

One of the main benefits of BFT is that it ensures safety regardless of synchrony assumptions. The authors justify the need for this property by noting that denial of service attacks can be used by a malicious adversary to violate timing assumptions. While a DoS attack cannot impact safety, it can be used to increase the timeout value used to detect a faulty leader. During the attack, the timeout doubles with each view change. If the adversary stops the attack when a malicious leader is in power, then that leader will be able to slow the system down to a throughput of roughly $f+1$ updates per TO, where TO is potentially very large, using the attack described in the previous section. This vulnerability stems from the inability of BFT to reduce the timeout and adapt to the network conditions after the system stabilizes.

4 THE PRIME PROTOCOL

In this section we present Prime, a new Byzantine fault-tolerant state machine replication protocol designed to mitigate the types of attacks described in Section 3. Prime requires $3f+1$ servers to tolerate f Byzantine faults.

4.1 Prime Ordering Protocol

Prime uses a rotating coordinator protocol to assign a total order to client updates. The servers execute the updates according to this total order, and they thus remain replicas of one another. Prime establishes the total order in two phases. In the first phase, each server disseminates its updates to the other servers and coordinates an agreement protocol, which *preorders* those updates that it originated. Each preordering agreement protocol coordinated by a different server operates independently and in parallel. A preordered update, u, is bound to a *preorder identifier*, (o, i), where u is the ith update preordered by server o. Thus, the preordering phase enables correct servers to consistently refer to updates using their preorder identifiers. In the second phase, an elected leader coordinates a global ordering protocol, which establishes a total order on batches of preordered updates. The final total order on updates is achieved by deterministically assigning an order to the updates in each batch based on their preorder identifiers.

Preordering Phase: When originating server o receives update u from one of its clients, it sends a $\langle \text{PO-REQUEST}, seq, u, o \rangle_{\sigma_o}$ message, req, to all servers, where seq is a local sequence number that o increments each time it sends a new PO-REQUEST. We refer to this local sequence number as a *preorder sequence number*. Upon receiving req, each correct server, i, sends a $\langle \text{PO-ACK}, seq, D(u), o, i \rangle_{\sigma_i}$ message to all other servers if i has not previously received a PO-REQUEST from o with sequence number seq. A set consisting of req and $2f$ matching PO-ACK messages constitutes a *preorder-certificate*, which is proof that the correct servers agree that preorder identifier (o,seq) is uniquely bound to u.

Each server, i, maintains a vector, PO_Aru[], where PO_Aru[o] contains the maximum sequence number, n, such that i has preorder-certificates for all preordered updates with identifiers (o,j), with $j \leq n$. Each server, i, periodically broadcasts a $\langle \text{PO-ARU}, vec, i \rangle_{\sigma_i}$ message, where vec is its local PO_Aru vector. The PO-ARU message serves as a cumulative acknowledgement for preordered updates. Given two PO-ARU messages, m_1 and m_2, Figure 1 defines what it means for m_1 to be *at least as up-to-date as* m_2, *more up-to-date* than m_2, and *consistent* with m_2. Each server stores the most up-to-date, consistent PO-ARU message received from each other server in a vector, Last_PO_Aru[], indexed by server identifier. We describe how we blacklist faulty servers that send PO-ARU messages that are not consistent when we present the SUSPECT-LEADER protocol, below.

Global Ordering Phase: Prime's global ordering phase is similar to BFT and uses three message rounds (see Section 3). While BFT establishes a total order on PRE-PREPARE messages containing updates, Prime's global ordering phase establishes a total order on PRE-PREPARE messages containing *proof matrices*. Each proof matrix is a vector of PO-ARU messages. A correct leader, l, periodically sends a $\langle \text{PRE-PREPARE}, v, seq, pm, l \rangle_{\sigma_l}$ message, where v is the current view number, seq is a global sequence number, and pm is the leader's Last_PO_Aru vector (which is a proof matrix). $pm[o]$ is either a PO-ARU message signed by server o or a null vector of length $|\mathcal{R}|$, indicating that o has not yet cumulatively acknowledged any preorder-certificates.

We now explain how a server obtains a total order on updates from the totally ordered stream of PRE-PREPARE messages. Call this stream of PRE-PREPARE messages $T = \langle T_1, T_2, \ldots \rangle$. Intuitively, globally ordering a PRE-PREPARE message expands the set of preordered updates

1. For $m_1 = \langle \text{PO-ARU}, vec_1, i \rangle_{\sigma_i}$ and $m_2 = \langle \text{PO-ARU}, vec_2, i \rangle_{\sigma_i}$, we say that:
   - m_1 is *at least as up-to-date as* m_2 when $(\forall j \in \mathcal{R})[vec_1[j] \geq vec_2[j]]$.
   - m_1 is *more up-to-date than* m_2 when m_1 is at least as up to date as $m_2 \ \wedge \ (\exists j \in \mathcal{R})[vec_1[j] > vec_2[j]]$.
   - m_1 and m_2 are *consistent* when m_1 is at least as up to date as m_2, or m_2 is at least as up to date as m_1.

2. For originating server o and preorder sequence number *po_seq*, *Preorder_Proof_Exists*(*o*, *po_seq*, $\langle$PRE-PREPARE,*,*,pm,l$\rangle_{\sigma_l}$) is true iff:
   - $|\{i : i \in \mathcal{R} \wedge pm[i][o] \geq po\_seq\}| \geq 2f + 1$

3. M($pp = \langle \text{PRE-PREPARE}, *, seq, *, * \rangle_{\sigma_*}$) = $\{(o, s) : o \in \mathcal{R} \ \wedge \ s \in \mathbb{N} \wedge$ *Preorder_Proof_Exists*(*o*,*s*,*pp*)$\}$

4. $\mathcal{B}$ is a set of blacklisted servers.

5. For $pp = \langle \text{PRE-PREPARE}, *, *, p_{pp}, * \rangle_{\sigma_*}$ and $pm = \langle \text{PROOF-MATRIX}, p_{pm}, * \rangle_{\sigma_*}$, where p_{pp} and p_{pm} denote proof matrices, we say that:
   - *pp covers pm* if $\forall i \in \mathcal{R} - \mathcal{B}$, $p_{pp}[i]$ is at least as up-to-date as $p_{pm}[i]$.

6. Preordered update (*o*,*s*) is *eligible* (for execution) iff $\exists$ a globally ordered PRE-PREPARE, *pp*, such that (*o*,*s*) $\in$ M(*pp*)

Fig. 1: Definitions and terminology used by the Prime ordering protocol.

that are eligible for execution. Let M map a globally ordered PRE-PREPARE, *pp*, to a set of preordered updates, P, where P contains those preordered updates, (o, s), for which *Preorder_Proof_Exists*(o, s, *pp*) is true (see Figure 1). Let L be a function that lexicographically orders the elements of P by their preorder identifiers. Then the final total order, U, on updates is obtained by $U = L(M(T_1)) \ || \ L(M(T_2) - M(T_1)) \ || \ L(M(T_3) - M(T_2)) \ldots$, where $||$ denotes concatenation and $-$ denotes set difference.

Prime guarantees that for all pairs of globally ordered PRE-PREPARE messages, $\langle \text{PRE-PREPARE}, *, seq, pm, * \rangle_{\sigma_*}$ and $\langle \text{PRE-PREPARE}, *, seq', pm', * \rangle_{\sigma_*}$, where $seq > seq'$, $(\forall i \in \mathcal{R})[pm[i]$ is at least as up-to-date as $pm'[i]] \wedge (pm \neq pm')$. This constraint ensures that Prime's global ordering phase correctly establishes a global order on preordered updates. The correct servers enforce this guarantee by performing a validity check on each PRE-PREPARE before sending a corresponding PREPARE.

Part A of Figure 2 summarizes the path of an update, u, through the system in the fault-free case. The update is preordered in two rounds, after which its preordering is cumulatively acknowledged in PO-ARU messages. When the network is stable, faulty servers cannot delay the preordering of u because correct servers need only wait for PO-ACK messages from each other to collect a preorder-certificate for u. In turn, the faulty servers cannot delay how quickly the preordering of u is cumulatively acknowledged in the PO-ARU messages of correct servers. A correct leader sends a PRE-PREPARE, *pp*, whose proof matrix includes these PO-ARU messages. u will be executed when *pp* is globally ordered.

Reconciliation: In Prime, a server sends PREPARE and COMMIT messages for a PRE-PREPARE message, *pp*, even

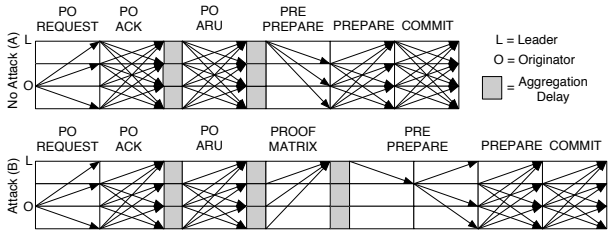

Fig. 2: Common case operation of Prime ($f = 1$). Part A shows the messages and protocol rounds when the leader is correct. Part B shows the delay added by a malicious leader that performs well enough to stay in power. The malicious leader ignores PO-ARU messages and sends its PRE-PREPARE to only one correct server.

if it has not received those updates that will become eligible for execution when *pp* is globally ordered. Consequently, although Prime guarantees that at least $f + 1$ correct servers receive each eligible update, it makes no guarantees regarding *which* correct servers have received a particular eligible update. Malicious servers can attempt to exploit this behavior to block execution.

To understand how this is possible, note that a correct server can only execute the gap-free prefix of the totally ordered eligible updates that it possesses. Each time a malicious server originates and preorders update u, it can intentionally fail to send u to f correct servers. If u becomes eligible, these servers will block until they recover u. Note that, without a reconciliation mechanism, each malicious server can block execution at f correct servers. Therefore, when $f \geq 3$, all correct servers can be blocked, because the number of servers that can be blocked (f^2) exceeds the number of correct servers ($2f+1$). In order to prevent these kinds of attacks, Prime incorporates a bandwidth-efficient and timely update reconciliation mechanism. Together, Prime's preordering phase and its reconciliation procedure provide a reliable broadcast service; if update u becomes eligible for execution, reconciliation guarantees that all correct servers will receive u. Pseudocode for Prime's reconciliation procedure is contained in Figure 3.

Conceptually, the reconciliation procedure operates on the ordered sequence of updates defined by the total order $U = U_1 \ || \ U_2 \ || \ldots$. Recall that each U_i is a sequence of preordered updates that became eligible for execution with the global ordering of pp_i, the PRE-PREPARE globally ordered with sequence number i. From the way U_i is created, for each preordered update (o, s) in U_i, there exists a set, $R_{o,s}$, of at least $2f+1$ servers whose PO-ARU messages cumulatively acknowledged (o, s) in pp_i. Prime's reconciliation procedure operates by having $2f+1$ servers in $R_{o,s}$ send erasure-coded parts of the PO-REQUEST containing (o, s) to those servers that have not cumulatively acknowledged preordering it. Note that if $|R_{o,s}| > 2f + 1$, the set of $2f + 1$ senders is chosen deterministically. Since f of the senders may be faulty, Prime uses an MDS($2f + 1$,$f + 1$) maximum distance separable erasure encoding [16], such that a server needs to receive $f + 1$ out of $2f + 1$ reconciliation messages to

```
/* Reconciliation Procedure run at server i */
Reconcile( seq )
A1. pp ← ⟨PRE-PREPARE,*,seq,pm,l⟩σl
A2. pp' ← ⟨PRE-PREPARE,*,seq-1,*,l'⟩σl'
A3. For each PO identifier (o,s) in L(M(pp) - M(pp'))
A4.   c ← 0
A5.   For j = 1 to N
A6.      if pm[j][o] ≥ s
A7.         c ← c + 1
A8.         if ( j = i and c ≤ 2f + 1 )
A9.            req = ⟨PO-REQUEST,s,*,o⟩σo
A10.           part ← Erasure_Encoded_Part( req, c )
A11.           For r = 1 to N
A12.              if Last_PO_Aru[r][o] < s
A13.                 SEND to server r:
A14.                 ⟨RECONCILIATION,o,s,c,part,i⟩σi
```

Fig. 3: Reconciliation Procedure, used to send erasure-coded reconciliation messages. $2f+1$ servers (at least $f+1$ of which are correct) send erasure-coded parts for each preordered update (o, s). M (line A3) is defined in Figure 1. L (line A3) is a function that lexicographically orders a set of preordered updates.

decode the associated PO-REQUEST. This guarantees that a correct server will receive enough parts to be able to decode the PO-REQUEST.

To improve efficiency, each server runs the reconciliation procedure speculatively; instead of waiting for a PRE-PREPARE message, pp, to be globally ordered, each server runs Reconcile upon first receiving pp. This proactive approach allows updates to be recovered in parallel with the remainder of the global ordering protocol.

Since a correct server will not send a reconciliation message unless at least $2f+1$ servers have cumulatively acknowledged the corresponding PO-REQUEST message, reconciliation messages for a given update are sent to a maximum of f servers. Assuming an update size of s_u, the $2f+1$ erasure-coded parts have a total size of $(2f+1)s_u/(f+1)$. Since these parts are sent to at most f servers, the amount of reconciliation data sent per update across all links is at most $f(2f+1)s_u/(f+1) < (2f+1)s_u$. During preordering, an update is sent to between $2f$ and $3f$ servers, which requires at least $2fs_u$. Therefore, reconciliation uses approximately the same amount of aggregate bandwidth as update dissemination. Note that a single server needs to send at most one reconciliation part per update, which guarantees that at least $f+1$ correct servers share the cost of reconciliation.

4.2 Detecting Malicious Leaders

A malicious leader can mount two types of performance attacks against Prime. First, it can propose a global ordering on preordered updates slowly by sending PRE-PREPARE messages at a slow rate. Some strategies for the leader to slow down the sending of its PRE-PREPAREs are illustrated in Part B of Figure 2. Prime uses the SUSPECT-LEADER protocol, described below, to detect slow leaders. Second, even if it sends timely PRE-PREPARE messages, a malicious leader can intentionally send a PRE-PREPARE, pp, whose proof matrix does not contain the most up-to-date PO-ARU messages that it has received. This can prevent preordered updates that would have become eligible for execution when pp is globally ordered from becoming eligible. Defending against these two performance attacks allows Prime to meet BOUNDED-DELAY (see Definition 2.5).

Enforcing up-to-date Pre-Prepare messages: To simplify this section, we first assume that all PO-ARU messages from the same server are consistent. The section on blacklisting (below) describes subtle issues regarding PO-ARU messages that are not consistent. Each non-leader server, i, periodically sends a $\langle \text{PROOF-MATRIX}, pm, i \rangle_{\sigma_i}$ message to the leader, where pm is i's Last_PO_Aru[]. Server i expects the leader to include PO-ARU messages that are at least as up-to-date as those in pm in its next PRE-PREPARE. To understand why a non-leader server is justified in this expectation, note that the leader can simply adopt any of the PO-ARU messages in pm that are more up-to-date than what it currently has in its Last_PO_Aru[]. Thus, a correct leader will send, in its next PRE-PREPARE, a proof matrix with PO-ARU messages that are at least as up-to-date as those in pm. We say that such a PRE-PREPARE *covers* pm (see Figure 1). A critical property of Prime, which differs from existing leader-based solutions, is that the leader requires a bounded amount of bandwidth and computational resources, independent of system throughput, to perform its role as leader; the size of a PRE-PREPARE is dependent only on the number of servers, and a single PRE-PREPARE can propose a global ordering on an arbitrary number of preordered updates.

Blacklisting Servers: A correct server always sends consistent PO-ARU messages. Therefore, a pair of inconsistent PO-ARU messages (i.e., two messages that are not consistent) from server r constitutes proof that r is malicious. A correct server that collects this proof adds r to a set of blacklisted servers, $\mathcal{B}$, and broadcasts the proof, causing all correct servers to blacklist r. As shown in Figure 1, when we test if a PRE-PREPARE message covers a PROOF-MATRIX message, we do not compare PO-ARU messages from blacklisted servers. This is important because, in order to stay in power, a correct leader may need to send a PRE-PREPARE message that covers all PROOF-MATRIX messages that it has received. If the leader receives PROOF-MATRIX messages that contain inconsistent PO-ARU messages from server r, then it may need to include one of these in its PRE-PREPARE. By definition, neither inconsistent PO-ARU is at least as up-to-date as the other, and therefore, the leader may fail to include the most up-to-date PO-ARU message from r in its PRE-PREPARE.

Without a blacklisting mechanism, this can cause a correct server, c, to suspect a correct leader, l, as follows: Let m_1 and m_2 be two inconsistent PO-ARU messages from the same malicious server. Suppose that: (1) l receives m_1 in a PROOF-MATRIX message, pm, from c, (2) l receives m_2 in a PROOF-MATRIX from a server other than c, and (3) then, l includes m_2 (instead of m_1) in PRE-PREPARE pp. When server c receives pp, it assesses whether the leader has performed as expected by checking if pp covers pm. Since pp does not cover pm, c will view the leader as performing worse than it

should. From the way that the Suspect-Leader protocol works (described below), this can cause a correct leader to be suspected when it should not be.

Our blacklisting mechanism mitigates attacks in which a malicious server sends inconsistent PO-ARU messages. Before a correct server checks if $\langle$PRE-PREPARE, $*, *, p_{pp}, *\rangle_{\sigma_*}$ covers $\langle$PROOF-MATRIX, $p_{pm}, *\rangle_{\sigma_*}$, where p_{pp} and p_{pm} denote proof matrices, it first performs the following procedure: for each server i in $\mathcal{R}-\mathcal{B}$, if $p_{pp}[i]$ is not consistent with $p_{pm}[i]$, add i to $\mathcal{B}$. This procedure blacklists any servers whose inconsistent PO-ARU messages may have otherwise caused the correct server to falsely suspect a correct leader. Intuitively, when testing if a PRE-PREPARE message covers a PROOF-MATRIX message, correct servers are able to ignore inconsistent PO-ARU messages before they cause a correct leader to appear malicious.

Pre-Prepare Flooding: Prime's mechanism for detecting malicious leaders requires a simple addition to the global ordering phase to ensure timely global ordering. Upon receiving a PRE-PREPARE, pp, a correct server broadcasts it. This guarantees that all correct servers receive pp within one round from the time that the first correct server receives it, at which point no faulty server can delay the correct servers from globally ordering pp. Flooding PRE-PREPAREs forces a malicious leader to delay sending PRE-PREPAREs to all correct servers in order to add unbounded delay to the global ordering phase. In practice, the rate at which the leader sends PRE-PREPAREs can be configured so that this flooding requires a small bandwidth overhead.

Suspect-Leader Protocol: Since the leader requires bounded resources to perform its role as leader, if the network is stable, the leader can be expected to send up-to-date PRE-PREPAREs in a timely manner. To leverage this, we require a mechanism whereby non-leader servers can (1) dynamically determine how fast a timely leader should perform, (2) monitor the performance of the current leader, and (3) suspect the leader if it is not performing fast enough. Each time a server sends a PROOF-MATRIX message, pm, it computes the delay between sending pm and receiving a PRE-PREPARE covering pm. We call this delay the *turn-around-time* (abbreviated TAT) provided by the leader. The goal of SUSPECT-LEADER is to force any leader that stays in power to provide a timely TAT to at least one correct server.

Each correct server, i, locally decides whether to suspect the leader by computing two values, *TAT_acceptable* and *TAT_leader*. TAT_acceptable is a standard against which server i judges the current leader, and TAT_leader is a measure of the current leader's performance. Server i suspects the leader if TAT_leader $>$ TAT_acceptable.

TAT_acceptable and TAT_leader are computed so that, when PRIME-STABILITY holds, SUSPECT-LEADER meets two key properties. L^* is the maximum latency between any two correct servers after the network stabilizes. Δ_{pp} is a value greater than the maximum time between a correct server sending successive PRE-PREPARE messages. K_{Lat} (see Section 2) accounts for latency variability. Let $B = 2K_{Lat}L^* + \Delta_{pp}$. We now state the properties.

PROPERTY 4.1: Any server that retains a role as leader must provide a turn-around time to at least one correct server that is no more than B.

PROPERTY 4.2: There exists a set of at least $f + 1$ correct servers (the permanent leaders) that will not be suspected by any correct server if elected leader.

Intuitively, Property 4.1 ensures that a faulty leader will be suspected unless it provides a timely TAT to at least one correct server. We consider a TAT, $t \leq B$, to be timely because B is within a constant factor of the TAT that the slowest correct server might provide. This factor is a function of the latency variability that SUSPECT-LEADER is configured to tolerate. Note that malicious servers cannot affect the value of B. Property 4.2 ensures that view changes cannot occur indefinitely. Prime does not guarantee that the slowest f correct servers will not be suspected because slow faulty leaders cannot be distinguished from slow correct leaders.

Figure 4 contains pseudocode for SUSPECT-LEADER. Server i initializes its data structures at the beginning of each new view (Block A). The remaining blocks run in parallel. In Block B, server i uses a simple ping protocol to measure the RTT to each other server, j. Server i sends this measured RTT to j. Using this value, j computes the maximum TAT that i would compute for j if j were the leader, and stores it in TATs_If_Leader[i]. In Block C, server i uses TATs_If_Leader[] to compute an upper bound, α, on the value of TAT_Leader that any correct server will compute for i if it were leader. Each server broadcasts its value of α and stores the values that it receives in TAT_Leader_UBs[]. In Block D, each non-leader server broadcasts the maximum TAT that the leader has provided it in the current view and stores the values that it receives in Reported_TATs[]. In Block E, each server computes TAT_acceptable using TAT_Leader_UBs[], computes TAT_Leader using Reported_TATs[], and compares these values to decide whether to suspect the leader.

Proof of Property 4.1: From Block B of Figure 4, at least $2f + 1$ cells in server i's TATs_If_Leader[] vector eventually contain values, v, sent by correct servers. By definition, each $v \leq B$. Since at most f servers are faulty, at least one of the $f + 1$ highest values in Sorted_TATs (line C2) is from a correct server and thus less than or equal to B. Server i computes α as the minimum of these $f + 1$ highest values (line C3), and thus $\alpha \leq B$. Figure 5 (left side) depicts this argument graphically.

Server i stores the values of α computed by each other server. Thus, at least $2f + 1$ of the cells in server i's TAT_Leader_UBs[] vector eventually contain α values from correct servers (each of which is no more than B). Using a parallel argument as above, at least one of the

```
/* Initialization, run at start of new view */
A1. For i = 1 to N, TATs_If_Leader[i] ← ∞
A2. For i = 1 to N, TAT_Leader_UBs[i] ← ∞
A3. For i = 1 to N, Reported_TATs[i] ← 0
A4. ping_seq ← 0

/* RTT Measurement Task, run at server i */
B1. Periodically:
B2.   BROADCAST: ⟨RTT-PING, view, ping_seq, i⟩σi
B3.   ping_seq++
B4. Upon receiving ⟨RTT-PING, view, seq, j⟩σj:
B5.   SEND to server j: ⟨RTT-PONG, view, seq, i⟩σi
B6. Upon receiving ⟨RTT-PONG, view, seq, j⟩σj:
B7.   rtt ← Measured RTT for pong message
B8.   SEND to server j: ⟨RTT-MEASURE, view, rtt, i⟩σi
B9. Upon receiving ⟨RTT-MEASURE, view, rtt, j⟩σj:
B10.  t ← rtt * K_Lat + Δ_pp
B11.  if t < TATs_If_Leader[j]
B12.    TATs_If_Leader[j] ← t

/* TAT_Leader Upper Bound Task, run at server i */
C1. Periodically:
C2.   Sorted_TATs ← SORT-ASCENDING TATs_If_Leader[]
C3.   α ← Sorted_TATs[2f+1]
C4.   BROADCAST: ⟨TAT-UB, view, α, i⟩σi
C5. Upon receiving ⟨TAT-UB, view, tat_ub, j⟩σj:
C6.   if tat_ub < TAT_Leader_UBs[j]
C7.     TAT_Leader_UBs[j] ← tat_ub

/* TAT Measurement Task, run at server i */
D1. Periodically:
D2.   max_tat ← Maximum TAT measured this view
D3.   BROADCAST: ⟨TAT-MEASURE, view, max_tat, i⟩σi
D4. Upon receiving ⟨TAT-MEASURE, view, tat, j⟩σj
D5.   if tat > Reported_TATs[j]
D6.     Reported_TATs[j] ← tat

/* Suspect Leader Task */
E1. Periodically:
E2.   Sorted_TAT_UBs ← SORT-ASCENDING TAT_Leader_UBs[]
E3.   TAT_acceptable ← Sorted_TAT_UBs[2f+1]
E4.   Sorted_TATs ← SORT-ASCENDING Reported_TATs[]
E5.   TAT_leader ← Sorted_TATs[f+1]
E6.   if TAT_leader > TAT_acceptable
E7.     Suspect Leader
```

Fig. 4: SUSPECT_LEADER Protocol, used to determine whether a server should suspect the leader. View numbers refer to the view in the global ordering protocol.

$f+1$ highest values in Sorted_TAT_UBs (line E2) is from a correct server and thus less than or equal to B. Server i computes TAT_acceptable as the minimum of these $f+1$ highest values (line E3), and thus TAT_acceptable $\leq B$. The preceding argument is illustrated on the right side of Figure 5.

If a malicious leader remains in power, there are at least $f+1$ servers (at least one of which is correct) for which TAT_leader $\leq$ TAT_acceptable always holds. Thus, at least one correct server collects TAT-MEASURE messages from $f+1$ servers (at least one of which is correct) with values v such that $v \leq$ TAT_acceptable. Therefore, the malicious leader is providing a TAT, t, such that $t \leq$ TAT_Acceptable $\leq B$, to at least one correct server.

Proof of Property 4.2: Since TAT_acceptable is the $(2f+1)$st lowest value in TAT_Leader_UBs[], at least $f+1$ correct servers sent values for α such that $\alpha \leq$ TAT_acceptable. Each permanent leader, l, has a set of at least $f+1$ correct servers that, if l is elected, will report TATs, t, with $t \leq \alpha \leq$ TAT_acceptable. Thus, any correct server will compute TAT_leader $\leq$ TAT_acceptable and will not suspect l.

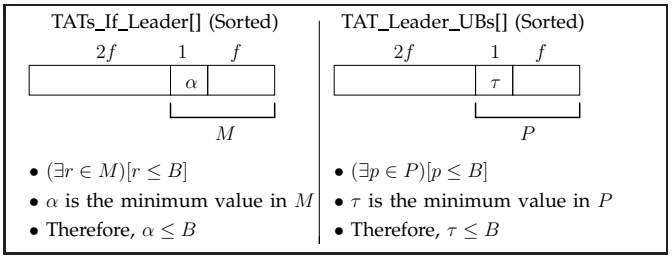

Fig. 5: The value of TAT_acceptable (τ) computed at any correct server converges to a value where $\tau \leq B$. The sets M (left side) and P (right side) contain the $f+1$ highest values in their respective vectors. M must eventually contain at least one value, v, reported by a correct server, where $v \leq B$. Thus, $\alpha \leq B$. Using a parallel argument, the right side shows that $\tau \leq B$.

5 THE PRIME VIEW CHANGE PROTOCOL

In order for the BOUNDED-DELAY property to be useful in practice, the time at which it begins to hold (after the network stabilizes) should not be able to be set arbitrarily far into the future by the faulty servers. As we now illustrate, achieving this requirement necessitates a different style of view change protocol than the one used by BFT (and other existing leader-based protocols).

5.1 BFT's View Change Protocol

To facilitate a comparison between Prime's view change protocol and the ones used by existing protocols, we review the BFT view change protocol. A newly elected leader collects state from $2f+1$ servers in the form of VIEW-CHANGE messages, processes these messages, and subsequently broadcasts a NEW-VIEW message. The NEW-VIEW contains the set of $2f+1$ VIEW-CHANGE messages, as well as a set of PRE-PREPARE messages that *replay* pending updates that may have been ordered by some, but not all, correct servers in a previous view; the VIEW-CHANGE messages allow the non-leader servers to verify that the leader constructed the set of PRE-PREPARE messages properly. We refer to the contents of the NEW-VIEW as the *constraining state* for this view.

Although the VIEW-CHANGE and NEW-VIEW messages are logically single messages, they may be large, and thus the non-leader servers cannot determine exactly how long it should take for the leader to receive and disseminate the necessary state. A non-leader server sets a timeout on suspecting the leader when it learns of the leader's election, and it expires the timeout if it does not receive the NEW-VIEW or does not execute the first update on its queue within the timeout period. The timeout used for suspecting the current leader doubles with every view change, guaranteeing that correct leaders eventually have enough time to complete the protocol.

5.2 Motivation and Protocol Overview

The view change protocol outlined above is insufficient for Prime. Doubling the timeouts greatly increases the power of the faulty servers; if the timeout grows very

high during unstable periods, then a faulty leader can cause the view change to take much longer than it would take with a correct leader. If Prime were to use such a protocol, then the faulty servers could delay the time at which BOUNDED-DELAY begins to hold by increasing the duration of the view changes in which they are leader. The amount of the delay would be a function of how many view changes occurred in the past, which can be manipulated by causing view changes during unstable periods (e.g., by using a denial of service attack).

To overcome this issue, Prime uses a fundamentally different approach for its view change protocol: whereas BFT's protocol is entirely coordinated by the leader, Prime's view change protocol is designed to rely on the leader as little as possible. The key observation is that the leader neither needs to collect view change state from $(2f + 1)$ servers nor disseminate constraining state to the non-leader servers in order to fulfill its role as leader. Instead, the leader can constrain non-leader servers simply by sending a single physical message that identifies which view change state messages should constitute the constraining state. Thus, instead of being responsible for state collection, processing, and dissemination, the leader is only responsible for making a single decision and sending a single message (which we call the leader's REPLAY message). The challenge is to construct the view change protocol in a way that will allow non-leader servers to force the leader to send a valid REPLAY message in a timely manner.

How can a single physical message identify the many view change state messages that constitute the constraining state? Each server disseminates its view change state using a Byzantine fault-tolerant reliable broadcast protocol (e.g., [17]). The reliable broadcast protocol guarantees that all servers that collect view change state from any server i in view v collect exactly the same state; in addition, if any correct server collects view change state from server i in view v, then all correct servers eventually will do so. Given these properties, the leader's REPLAY message simply needs to contain a list of $2f + 1$ server identifiers in order to unambiguously identify the constraining state. For example, if the leader's REPLAY message contains the list $\langle 1, 3, 4\rangle$, then the view change state disseminated by servers 1, 3, and 4 should be used to become constrained. As described below, the REPLAY message also contains a proof that all of the referenced view change state messages will eventually be delivered to all correct servers.

A critical property of the reliable broadcast protocol used for view change state dissemination is that it cannot be slowed down by the faulty servers. Correct servers only need to send and receive messages from one another in order to complete the protocol. Therefore, the state dissemination phase takes as much time as is required for correct servers to pass the necessary information between one another, and no more.

If the leader is faulty, it can send a REPLAY message whose list contains faulty servers, from which it may be impossible to collect view change state. Thus, the protocol requires that the leader's list be verifiable, which we achieve by using a threshold signature protocol. Once a server finishes collecting view change state from $2f+1$ servers, it announces a list containing their server identifiers. A server submits a partial signature on a list L if it has finished collecting view change state from the $2f + 1$ servers in L. The servers combine $2f + 1$ matching partial signatures into a threshold signature on L; we refer to the pair consisting of L and its threshold signature as a *VC-Proof*. At least one correct server (in fact, $f+1$ correct servers) must have submitted a partial signature on L, which, by the properties of reliable broadcast, implies that all correct servers will eventually finish collecting view change state from the servers in L. Thus, by including a VC-Proof in its REPLAY, the leader can convince the non-leader servers that they will eventually collect the state from the servers in the list.

The last remaining challenge is to ensure that the leader sends its REPLAY message in a timely manner. The key property of the protocol is that the leader can immediately use a VC-Proof to generate the REPLAY message, *even if it has not yet collected view change state from the servers in the list*. Thus, after a non-leader server sends a VC-Proof to the leader, it can expect to receive the REPLAY message in a timely fashion. We integrate the computation of this turnaround time (i.e., the time between sending a VC-Proof to the leader and receiving a valid REPLAY message) into the normal-case SUSPECT-LEADER protocol to monitor the leader's behavior. By using SUSPECT-LEADER to ensure that the leader terminates the view change in a timely manner, we avoid the use of a timeout and its associated vulnerabilities. Table 1 summarizes Prime's view change protocol.

5.3 Detailed Protocol Description

Preliminaries: When a server learns that a new leader has been elected in view v, we say that it *preinstalls* view v. As described above, the Prime view change protocol uses an asynchronous Byzantine fault-tolerant reliable broadcast protocol for state dissemination. We assume that the identifiers used in the reliable broadcast are of the form $\langle i, v, seq\rangle$, where v is the preinstalled view number and $seq = j$ means that this message is the j^{th} message reliably broadcast by server i in view v. Using these tags guarantees that all correct servers agree on the messages reliably broadcast by each server in each view. We refer to the last global sequence number that a server has executed as that server's *execution ARU*.

State Dissemination Phase: A server's view change state consists of the server's execution ARU and a set of Prepare-Certificates for global sequence numbers for which the server has sent a COMMIT message but which it has not yet globally ordered. We refer to this set as the server's *PC-Set*. Upon preinstalling view v, server i reliably broadcasts a $\langle \text{REPORT}, v, execARU, numSeq, i\rangle_{\sigma_i}$ message, where v is the preinstalled view number, execARU is server i's execution ARU, and numSeq is

Phase	Action	Phase Completed Upon	Action Taken Upon Phase Completion	Progress Driven By
State Dissemination	All: Reliably broadcast REPORT and PC-SET messages	Collecting complete state from $2f+1$ servers	Broadcast VC-LIST	Correct Servers
Proof Generation	All : Upon collecting complete state from servers in VC-LIST, broadcast VC-PARTIAL-SIG (up to N times)	Combining $2f+1$ matching partial signatures	Broadcast VC-PROOF, Run SUSPECT-LEADER	Correct Servers
Replay	Leader: Upon receiving VC-PROOF, broadcast REPLAY message All: Agree on REPLAY	Committing REPLAY and collecting associated state	Execute all updates in replay window	Leader, monitored by SUSPECT-LEADER

TABLE 1: Summary of Prime's view change protocol.

the size of server i's PC-Set. Server i then reliably broadcasts each Prepare-Certificate in its PC-Set in a $\langle \text{PC-SET}, v, pc, i \rangle_{\sigma_i}$ message, where v is the preinstalled view number and pc is the Prepare-Certificate being disseminated.

A server will accept a REPORT message from server i in view v as valid if the message's tag is $\langle i, v, 0 \rangle$; that is, the REPORT message must be the first message reliably broadcast by server i in view v. The numSeq field in the REPORT tells the receiver how many Prepare-Certificates to expect. These must have tags of the form $\langle i, v, j \rangle$, where $1 \leq j \leq numSeq$.

Each server stores REPORT and PC-SET messages as they are reliably delivered. We say that server i has *collected complete state* from server j in view v when i has (1) reliably delivered j's REPORT message, (2) reliably delivered the numSeq PC-SET messages described in j's report, and (3) executed a global sequence number at least as high as the one contained in j's report. To meet the third condition, we assume that a reconciliation protocol runs in the background. In practice, correct servers will reserve some amount of their outgoing bandwidth for fulfilling reconciliation requests from other servers. Upon collecting complete state from a set S of $2f+1$ servers, server i broadcasts a $\langle \text{VC-LIST}, v, L, i \rangle_{\sigma_i}$ message, where v is the preinstalled view number and L is the list of server identifiers of the servers in S.

Proof Generation Phase: Each server stores VC-LIST messages as they are received. When server i has a $\langle \text{VC-LIST}, v, ids, \ j \rangle_{\sigma_j}$ message in its data structures for which it has collected complete state from all servers in ids, it broadcasts a $\langle \text{VC-PARTIAL-SIG}, v, ids, startSeq, pSig, i \rangle_{\sigma_i}$ message, where v is the preinstalled view number, ids is the list of server identifiers, startSeq is the global sequence number at which the leader should begin ordering in view v, and pSig is a partial signature computed on the tuple $\langle v, ids, startSeq \rangle$. startSeq is the sequence number directly after the replay window. It can be computed deterministically as a function of the REPORT messages collected from the servers in ids.

Upon collecting $(2f+1)$ matching VC-PARTIAL-SIG messages, server i take the following steps. First, it combines the partial signatures to generate a VC-Proof, p, which is a threshold signature on the tuple $\langle v, ids, startseq \rangle$. Second, it broadcasts a $\langle \text{VC-PROOF}, v, ids, startSeq, p, i \rangle_{\sigma_i}$ message. Third, it begins running SUSPECT-LEADER, treating the VC-PROOF message just as it would a PROOF-MATRIX in computing the maximum turnaround time provided by the leader in the current view (see Figure 4, Block D); specifically, server i starts a timer to compute the turnaround time between sending the VC-PROOF to the leader and receiving a valid REPLAY message (see below) for view v. Thus, the leader is forced to send the REPLAY message in a timely fashion, in the same way that it is forced to send timely PRE-PREPARE messages during normal-case operation.

Replay Phase: When the leader, l, receives a VC-PROOF message for view v, it broadcasts a $\langle \text{REPLAY}, v, ids, startSeq, p, l \rangle_{\sigma_l}$ message. By sending a REPLAY message, the leader proposes an ordering on the entire replay set implied by the contents of the VC-PROOF message. Specifically, for each sequence number, seq, between the maximum execution ARU found in the report messages of the servers in ids and startSeq, seq is either (1) bound to the Prepare-Certificate for that sequence number from the highest view, if one or more Prepare-Certificates were reported by the servers in ids, or (2) bound to a No-op, if no Prepare-Certificate for that sequence number was reported. It is critical to note that the leader itself may not yet have collected complete state from the servers in ids. Nevertheless, it can commit to using the state sent by the servers in ids in order to complete the replay phase.

When a non-leader server receives a valid REPLAY message for view v, it floods it to the other servers, treating the message as it would a typical PRE-PREPARE message. The REPLAY message is then agreed upon using REPLAY-PREPARE and REPLAY-COMMIT messages, whose functions parallel those of typical PREPARE and COMMIT messages. The REPLAY message does not carry a global sequence number because only one may be agreed upon (and subsequently executed) within each view. A correct server does not send a REPLAY-COMMIT message until it has collected complete state from all servers in the list contained in the REPLAY message. Finally, when a server commits the REPLAY message, it executes all sequence numbers in the replay window in one batch.

Besides flooding the REPLAY message upon receiving it, a non-leader server also stops the timer on computing the turnaround time for the VC-PROOF, if one was set. Note that a non-leader server stops its timer as long as it receives *some* valid REPLAY message, not necessarily one containing the *VC-Proof* it sent to the leader. The properties of reliable broadcast ensure that the server

will eventually collect complete state from those servers in the list contained in the REPLAY message.

One subtle consequence of the fact that "any REPLAY message will do" is that a faulty leader that sends conflicting REPLAY messages might be able to convince two different correct servers to stop their timers, even though neither REPLAY will ever be executed. In this case, since the REPLAY messages are flooded, all correct servers will eventually receive the conflicting messages. Since the messages are signed, the two messages constitute proof of corruption and can be broadcast. A correct server will suspect the leader upon collecting this proof. Thus, the system will not remain in a view with such a faulty leader forever, and the detection time is a function of the latency between correct servers.

6 PROOF SKETCH OF BOUNDED-DELAY

In this section we show that in those executions in which PRIME-STABILITY holds, Prime meets the BOUNDED-DELAY property (see Definition 2.5). L^* and B are as defined in Section 4.2. Δ_{agg} is a value greater than the maximum time between a correct server sending any of the following messages successively: PO-ARU, PROOF-MATRIX, and PRE-PREPARE.

We first consider the maximum amount of delay that can be added by a malicious leader that performs well enough to stay in power. As discussed in Section 4, the time between a correct server initiating an update, u, and all correct servers sending PROOF-MATRIX messages containing at least $2f + 1$ PO-ARUs that cumulatively acknowledge the preordering of u is at most three rounds plus $2\Delta_{agg}$. The malicious servers cannot increase this time beyond what it would take if only correct servers were participating. By Property 4.1, a leader that stays in power must provide a TAT, $t \leq B$, to at least one correct server. By definition, $\Delta_{agg} \geq \Delta_{pp}$. Thus, $B \leq 2K_{Lat}L^* + \Delta_{agg}$. Since correct servers flood PRE-PREPARE messages, all correct servers receive the PRE-PREPARE within three rounds and one aggregation delay of when the PROOF-MATRIX messages are sent. All correct servers globally order the PRE-PREPARE in two rounds from the time, t, the last correct server receives it. Reconciliation guarantees that all correct servers receive the update within one round of time t. Summing the total delays yields a maximum latency of $\beta = 6L^* + 2K_{Lat}L^* + 3\Delta_{agg}$.

If a malicious leader delays an update by more than B, it will be suspected and a view change will occur. View changes require a finite (and, in practice, small) amount of state to be exchanged among correct servers, and thus they complete in finite time. As described in Section 5, faulty leaders will quickly be suspected if they do not terminate the view change in a timely manner. SUSPECT-LEADER guarantees that at most $2f$ view changes can occur before the system settles on a leader that will remain in power forever. Therefore, there is a time after which the bound of β holds for any update initiated by a stable server.

7 PERFORMANCE EVALUATION

To evaluate the performance of Prime, we implemented the protocol and compared it to an available implementation of BFT. We show results for configurations with 4 servers ($f = 1$) and 7 servers ($f = 2$). Prime has similar performance to BFT when the systems are run in a benign environment, which is commonly the only environment in which Byzantine fault-tolerant replication systems are benchmarked. When strong attacks are mounted against both systems, Prime outperforms BFT by more than an order of magnitude.

Testbed and Network Setup: We used a system consisting of 7 servers, organized in a fully connected graph. Each server ran on a 3.2 GHz, 64-bit Intel Xeon computer. RSA signatures provided authentication and non-repudiation. Each computer can compute a 1024-bit RSA signature in 1.3 ms and verify it in 0.07 ms. We emulated the overhead of Cauchy-based Reed-Solomon erasure codes [16] used for reconciliation. Servers and clients sent unicast messages. We used the netem utility to place delay and bandwidth constraints on the links between the servers. We added 50 ms delay (emulating a US-wide deployment) to each link and limited the aggregate outgoing bandwidth of each server to 10 Mbps. Clients were evenly distributed among the servers and no delay or bandwidth constraints were set between the client and its server.

Attack Strategies: Our experimental results during attack show the minimum performance that must be achieved in order for a malicious leader to remain in power. Our measurements do not reflect the time required for view changes, during which a new leader is installed. Since a view change takes a finite, and, in practice, relatively small, amount of time, malicious leaders must cause performance degradation without being detected in order to have a prolonged effect on throughput. Therefore, we focus on the attack scenario where a malicious leader stays in power indefinitely while degrading performance.

We use the first attack on BFT described in Section 3. We present results for a very aggressive yet possible timeout (300 ms), yielding the most favorable performance for BFT under attack. To attack Prime, (1) the leader adds as much delay as possible (without being suspected) to the protocol, and (2) faulty servers force as much reconciliation as possible. A malicious leader can add approximately two rounds of delay to the global ordering phase (see Figure 2). The malicious servers force reconciliation by not sending their PO-REQUEST messages to f of the correct servers. Therefore, all updates originating from the faulty servers must be sent to these f correct servers using the reconciliation mechanism (Section 4). Moreover, the malicious servers only acknowledge each other's PO-REQUEST messages, forcing the correct servers to send reconciliation messages to them for all messages originating from correct servers. Thus, all messages undergo a reconciliation

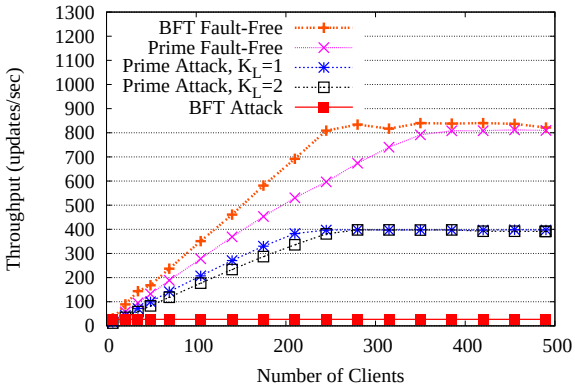

Fig. 6: 7 servers, Throughput (updates/sec) vs. number of clients, 50 ms Diameter

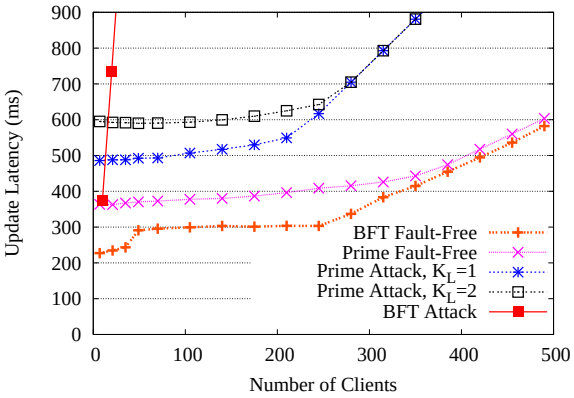

Fig. 7: 7 servers, Latency (ms) vs. number of clients, 50 ms Diameter

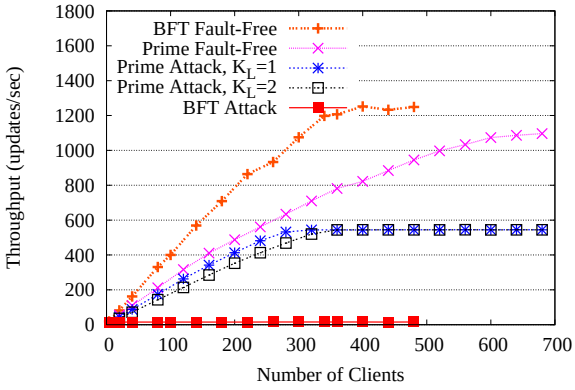

Fig. 8: 4 servers, Throughput (updates/sec) vs. number of clients, 50 ms Diameter

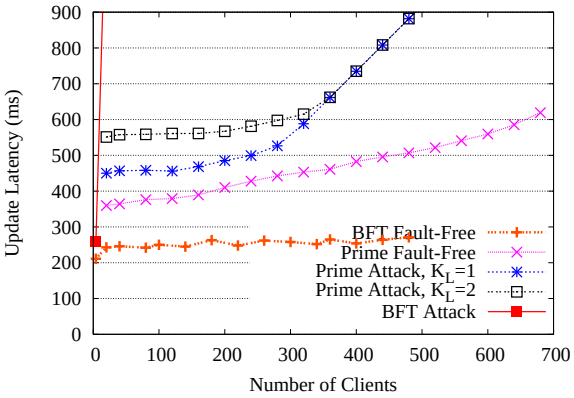

Fig. 9: 4 servers, Latency (ms) vs. number of clients, 50 ms Diameter

step, which consumes approximately the same outgoing bandwidth as update dissemination during preordering. This reduces the maximum achievable throughput by approximately half.

Performance Results: Figure 6 shows system throughput in updates per second as a function of the number of clients in the 7-server configuration. The clients send one write update (containing 512 bytes of data), wait for proof that the update has been ordered, and then submit their next update. BFT uses an optimization where clients send updates directly to all of the servers, and the BFT PRE-PREPARE message contains batches of update digests. When both protocols are not under attack, the throughput of BFT increases at a faster rate than the throughput of Prime, because BFT has fewer protocol rounds. BFT's performance plateaus due to bandwidth constraints at slightly less than 850 updates per second with about 250 clients. Prime reaches a similar plateau with about 350 clients.

Throughput results are much different when the two protocols are attacked. With an aggressive timeout of 300 ms, BFT can order less than 30 updates per second. With the default timeout of 5 sec, BFT can only order 2 updates per second (not shown). Prime plateaus at about 400 updates per second due to the bandwidth overhead of reconciliation. The slope of the curve corresponding to Prime under attack is less steep than when it is not under attack due to the delay added by the malicious leader. We include results with $K_{Lat} = 1$ and $K_{Lat} = 2$. K_{Lat} accounts for variabilities in latency (Section 2). As K_{Lat} increases, a malicious leader can add more delay to the turn-around time without being detected. Prime's throughput continues to increase until it becomes bandwidth constrained. BFT reaches its maximum throughput when there is one client per server. This throughput limitation, which occurs when only a small amount of the available bandwidth is used, is a consequence of judging the leader conservatively.

Figure 8 shows similar throughput trends in the 4-server configuration. When not under attack, both protocols plateau at higher throughputs than those shown in the 7-server configuration (Figure 6). Prime reaches a plateau of 1140 updates per second when there are 600 clients. In the 4-server configuration, each server sends a higher fraction of the executed updates than in the 7-server configuration. This places a relatively higher computational burden (due to RSA cryptography) on the servers in the 4-server configuration. Thus, there is a larger difference in performance when not under attack between Prime and BFT. When under attack, Prime outperforms BFT by a factor of 30.

Figure 7 shows update latency, at the client, as a function of the number of clients in the 7-server configuration. When the protocols are not under attack,

BFT has a lower latency than Prime, due to the differences in the number of protocol rounds. The latency of both protocols increases at different points before the plateau due to overhead associated with aggregation. The latency begins to climb steeply when the throughput plateaus due to update queueing at the servers. When under attack, the latency of Prime increases due to the two extra protocol rounds added by the leader. When $K_{Lat} = 2$, the leader can add approximately 100 ms more delay than when $K_{Lat} = 1$. The latency of BFT under attack climbs as soon as more than one client is added to each server, because the leader can order one update per server per timeout without being suspected. Figure 9 shows a similar trend in the 4-server configuration.

8 RELATED WORK

The protocols considered in this paper use the state machine approach [18], [19] to achieve replication, in which replicas execute a totally ordered stream of updates. This paper focused on leader-based Byzantine fault-tolerant SMR protocols. Other approaches are possible, such as using randomization, quorum systems, and hybrid architectures. A thorough analysis of how resilient these systems are to performance failures is an interesting avenue for future research.

Many Byzantine fault-tolerant SMR systems rely on a leader to coordinate the ordering protocol [5], [9], [10], [11], [12], [13], [14]. The consistency of these systems does not rely on synchrony assumptions, while liveness is guaranteed assuming the network meets certain stability properties. To ensure that the stability properties are eventually met in practice, they use exponentially growing timeouts during view changes. This makes these systems vulnerable to the type of performance degradation when under attack described in Section 3.2. In contrast, Prime uses the SUSPECT-LEADER protocol to allow correct servers to collectively decide whether the leader is performing fast enough by adapting to the network conditions once the system stabilizes.

Rampart [20] implements Byzantine atomic multicast over a reliable group multicast protocol. This is similar to how Prime uses preordering followed by global ordering. Both protocols disseminate updates to $2f+1$ servers before a coordinator assigns the global order. Drabkin et al. [21] observe the difficulty of setting protocol timeouts in the context of group communication in malicious settings.

Other Byzantine fault-tolerant protocols [3], [4], [8], [22] use randomization to circumvent the FLP impossibility result, guaranteeing termination with probability 1. These protocols incur a high number of communication rounds during normal-case operation (even those that terminate in an expected constant number of rounds). However, they do not rely on a leader to coordinate the ordering protocol, and thus may not suffer the same kinds of performance vulnerabilities when under attack. We believe it is an interesting open question to consider (1) whether their performance in fault-free configurations is sufficiently high, especially in high latency environments and (2) how resilient they are to performance degradation when under attack.

Byzantine quorum systems [23], [24], [25], [26] can also be used for replication. While early work in this area was restricted to a read/write interface, recent work uses quorum systems to provide SMR. The Q/U protocol [25] of Abd-El-Malek et al. requires $5f + 1$ replicas for this purpose and suffers performance degradation when write contention occurs. The HQ protocol [26] showed how to mitigate this cost by reducing the number of replicas to $3f + 1$. Since HQ uses BFT to resolve contention when it arises, it is vulnerable to the same types of performance degradation as BFT.

A different approach to SMR is to use a hybrid architecture in which different parts of the system rely on different fault and/or timing assumptions [27], [28], [29]. The different components are therefore resilient to different types of attacks. We believe leveraging stronger timing assumptions may allow for more aggressive performance monitoring.

The Θ-Model [30] assumes that messages in transit simultaneously experience a bounded ratio of end-to-end delays. PRIME-STABILITY assumes an eventual ratio of delays on each link between correct servers.

9 CONCLUSIONS

In this paper we brought to light the vulnerability of current leader-based Byzantine fault-tolerant SMR protocols to performance degradation when under attack. We proposed the BOUNDED-DELAY correctness criterion to complement current liveness criteria by requiring the leader to act timely in order to stay in power. We presented Prime, a new Byzantine fault-tolerant SMR protocol, which meets BOUNDED-DELAY and is a first step towards making intrusion-tolerant replication resilient to performance attacks in malicious environments. Our experimental results show that Prime performs competitively with BFT in fault-free configurations and an order of magnitude better when under attack.

REFERENCES

[1] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[2] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[3] M. Ben-Or, "Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols," in *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC '83)*, 1983, pp. 27–30.

[4] M. O. Rabin, "Randomized Byzantine generals," in *The 24th Annual IEEE Symposium on Foundations of Computer Science*, 1983, pp. 403–409.

[5] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling Byzantine fault-tolerant replication to wide area networks," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*, Philadelphia, PA, USA, June 2006, pp. 105–114.

[6] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[7] L. M. R. Sampaio, F. V. Brasileiro, W. Cirne, and J. C. A. Figueiredo, "How bad are wrong suspicions? Towards adaptive distributed protocols," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)*, San Francisco, CA, USA, 2003, pp. 551–560.

[8] H. Moniz, N. F. Neves, M. Correia, and P. Veríssimo, "Randomized intrusion-tolerant asynchronous services," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*, 2006, pp. 568–577.

[9] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999, pp. 173–186.

[10] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for Byzantine fault-tolerant services," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, USA, October 2003, pp. 253–267.

[11] J.-P. Martin and L. Alvisi, "Fast Byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[12] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Customizable fault tolerance for wide-area replication," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*, Beijing, China, 2007, pp. 66–80.

[13] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*, 2007, pp. 45–58.

[14] J. Li and D. Mazieres, "Beyond one-third faulty replicas in Byzantine fault tolerant systems," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, 2007, pp. 131–144.

[15] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, AK, USA, June 2008, pp. 197–206.

[16] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," International Computer Science Institute, Tech. Rep. TR-95-048, August 1995.

[17] G. Bracha, "An asynchronous [(n - 1)/3]-resilient consensus protocol," in *Proceedings of the third annual ACM symposium on Principles of Distributed Computing (PODC '84)*, Vancouver, British Columbia, Canada, 1984, pp. 154–162.

[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[19] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[20] M. K. Reiter, "The Rampart Toolkit for building high-integrity services," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*. London, UK: Springer-Verlag, 1995, pp. 99–110.

[21] V. Drabkin, R. Friedman, and A. Kama, "Practical Byzantine group communication," in *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems (ICDCS '06)*, Lisboa, Portugal, 2006, p. 36.

[22] C. Cachin and J. A. Portiz, "Secure intrusion-tolerant replication on the internet," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, Bethesda, MD, USA, June 2002, pp. 167–176.

[23] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[24] D. Malkhi and M. K. Reiter, "Secure and scalable replication in Phalanx," in *Proceedings of the the 17th IEEE Symposium on Reliable Distributed Systems (SRDS '98)*, West Lafayette, IN, USA, 1998, pp. 51–58.

[25] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, 2005, pp. 59–74.

[26] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: A hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, Nov. 2006, pp. 177–190.

[27] P. Veríssimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch, "Intrusion-tolerant middleware: The road to automatic security," *IEEE Security & Privacy*, vol. 4, no. 4, pp. 54–62, 2006.

[28] M. Correia, N. F. Neves, and P. Veríssimo, "How to tolerate half less one Byzantine nodes in practical distributed systems," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)*, Florianpolis, Brazil, 2004, pp. 174–183.

[29] M. Serafini and N. Suri, "The fail-heterogeneous architectural model," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS '07)*, Beijing, China, 2007, pp. 103–113.

[30] J.-F. Hermant and J. Widder, "Implementing reliable distributed real-time systems with the *theta*-model," in *Proceedings of the 9th International Conference on Principles of Distributed Systems (OPODIS '05)*, Pisa, Italy, December 2005, pp. 334–350.

Yair Amir is a Professor in the Department of Computer Science, Johns Hopkins University, where he served as Assistant Professor since 1995, Associate Professor since 2000, and Professor since 2004. He holds a BS (1985) and MS (1990) from the Technion, Israel Institute of Technology, and a PhD (1995) from the Hebrew University of Jerusalem, Israel. Prior to his PhD, he gained extensive experience building C3I systems. He is a creator of the Spread and Secure Spread messaging toolkits, the Backhand and Wackamole clustering projects, the Spines overlay network platform, and the SMesh wireless mesh network. He has been a member of the program committees of the IEEE International Conference on Distributed Computing Systems (1999, 2002, 2005-07), the ACM Conference on Principles of Distributed Computing (2001), and the International Conference on Dependable Systems and Networks (2001, 2003, 2005). He is a member of the ACM and the IEEE Computer Society.

Brian Coan is Director of the Distributed Computing group at Telcordia, where he has worked since 1978. He works on providing reliable networking and information services in adverse network environments, possibly caused by cyber attacks. He received a PhD in computer science from MIT in the Theory of Distributed Systems Group in 1987. He holds the BSE Degree from Princeton University (1977) and the MS from Stanford (1979). He has twice served on the program committee of the ACM Conference on the Principles of Distributed Computing. He is a member of the ACM.

Jonathan Kirsch is a fifth year Ph.D. student in the Computer Science Department at the Johns Hopkins University under the supervision of Dr. Yair Amir. He received his B.Sc. degree in Computer Science from Yale University in 2004 and his M.S.E. in Computer Science from Johns Hopkins University in 2007. He is a member of the Distributed Systems and Networks Laboratory. His research interests include fault-tolerant replication and survivability. He is a member of the ACM and the IEEE Computing Society.

John Lane is a fifth year Ph.D. student in the Computer Science Department at Johns Hopkins University under the supervision of Dr. Yair Amir. He received his B.A. in Biology from Cornell University in 1992 and his M.S.E. in Computer Science from Johns Hopkins University in 2006. He is a member of the Distributed Systems and Networks Laboratory. His research interests include distributed systems, replication, and byzantine fault tolerance. He is a member of the ACM and the IEEE Computing Society.