Prime: Byzantine Replication Under Attack

Yair Amir, *Member, IEEE,* Brian Coan, Jonathan Kirsch, *Member, IEEE,* John Lane, *Member, IEEE*

Abstract—Existing Byzantine-resilient replication protocols satisfy two standard correctness criteria, safety and liveness, even in the presence of Byzantine faults. The runtime performance of these protocols is most commonly assessed in the absence of processor faults and is usually good in that case. However, in some protocols faulty processors can significantly degrade performance, limiting the practical utility of these protocols in adversarial environments. This paper demonstrates the extent of performance degradation possible in some existing protocols that do satisfy liveness and that do perform well absent Byzantine faults. We propose a new performance-oriented correctness criterion that requires a consistent level of performance, even when the system exhibits Byzantine faults. We present a new Byzantine fault-tolerant replication protocol that meets the new correctness criterion and evaluate its performance in fault-free executions and when under attack.

Index Terms—Fault tolerance, Reliability, Performance

1 Introduction

Existing Byzantine fault-tolerant state machine replication protocols are evaluated against two standard correctness criteria: *safety* and *liveness*. Safety means that correct servers do not make inconsistent ordering decisions, while liveness means that each update to the replicated state is eventually executed. Most Byzantine replication protocols are designed to maintain safety in all executions, even when the network delivers messages with arbitrary delay. This is desirable because it implies that an attacker cannot cause inconsistency by violating network-related timing assumptions. However, the well-known FLP impossibility result [1] implies that no asynchronous Byzantine agreement protocol can always be both safe and live, and thus these systems ensure liveness only during periods of sufficient synchrony and connectivity [2] or in a probabilistic sense [3], [4].

When the network is sufficiently stable and there are no Byzantine faults, Byzantine fault-tolerant replication systems can satisfy much stronger performance guarantees than liveness. The literature has many examples of systems that have been evaluated in such benign executions and that achieve throughputs of thousands of update operations per second (e.g., [5], [6]). It has been a less common practice to assess the performance of Byzantine fault-tolerant replication systems when some of the processors actually exhibit Byzantine faults. In this paper we point out that in many systems, a small number of Byzantine processors can degrade performance to a level far below what would be achievable with only correct processors. Specifically, the Byzantine processors can cause the system to make progress at an extremely slow rate, even when the network is stable and could support much higher throughput. 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 firsthand in 2005, when DARPA conducted a red team experiment on our Steward system [7]. Steward survived all of the tests according to the metrics of safety and liveness, and most attacks did not impact performance. However, in one experiment, we observed that the system was slowed down to twenty percent of its potential performance. After analyzing the attack, we found that we could slow the system down to roughly one percent of its potential performance. This experience led us to a new way of thinking about Byzantine fault-tolerant replication systems. We concluded that liveness is a necessary but insufficient correctness criterion for achieving high performance when the system actually exhibits Byzantine faults. This paper argues that new, *performance-oriented* correctness criteria are needed to achieve a practical solution for Byzantine replication.

Preventing the type of performance degradation experienced by Steward requires addressing what we call a *Byzantine performance failure*. Previous work on Byzantine fault tolerance has focused on mitigating Byzantine failures in the value domain (where a faulty processor tries to subvert the protocol by sending incorrect or conflicting messages) and the time domain (where messages from a faulty processor do not arrive within protocol timeouts, if at all). Processors exhibiting performance failures operate arbitrarily but correctly enough to avoid being suspected as faulty. They can send valid messages slowly but without triggering protocol timeouts; re-order or drop certain messages, both of which could be caused by a faulty network; or, with malicious intent, take one of a number of possible actions that a correct processor

• *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.*

in the same circumstances might take. Thus, processors exhibiting performance failures are correct in the value and time domains yet have the potential to significantly degrade performance. The problem is magnified in wide-area networks, where timeouts tend to be large and it may be difficult to determine what type of performance should be expected. Note that a performance failure is not a new failure mode but rather is a strategy taken by an adversary that controls Byzantine processors.

In order to better understand the challenges associated with building Byzantine fault-tolerant replication protocols that can resist performance failures, we analyzed existing protocols to assess their vulnerability to performance degradation by malicious servers. We observed that most of the protocols (e.g., [5], [6], [7], [8], [9], [10], [11]) share a common feature: they rely on an elected leader to coordinate the agreement protocol. We call such protocols *leader based*. We found that leader-based protocols are vulnerable to performance degradation caused by a malicious leader.

Based on the understanding gained from our analysis, we developed Prime [12], the first Byzantine fault-tolerant state machine replication protocol capable of making a meaningful performance guarantee even when some of the servers are Byzantine. Prime meets a new, performance-oriented correctness criterion, called BOUNDED-DELAY. Informally, BOUNDED-DELAY bounds the latency between a correct server receiving a client operation and the correct servers executing the operation. The bound is a function of the network delays between the correct servers in the system. This is a much stronger performance guarantee than the eventual execution promised by existing liveness criteria.

Like many existing Byzantine fault-tolerant replication protocols, Prime is leader based. Unlike existing protocols, Prime bounds the amount of performance degradation that can be caused by the faulty servers, including by a malicious leader. Two main insights motivate Prime's design. First, most protocol steps do not need any messages from the faulty servers to complete. Faulty servers cannot delay these steps beyond the time it would take if only correct servers were participating in the protocol. Second, the leader should require a predictable amount of resources to fulfill its role as leader. In Prime, the resources required by the leader to do its job as leader are bounded as a function of the number of servers in the system and are independent of the offered load. The result is that the performance of the few protocol steps that do depend on the (potentially malicious) leader can be effectively monitored by the non-leader servers.

We present experimental results evaluating the performance of Prime in fault-free and under-attack executions. Our results demonstrate that Prime performs competitively with existing Byzantine fault-tolerant replication protocols in fault-free configurations and that Prime performs an order of magnitude better in under-attack executions in the configurations tested.

2 Case Study: BFT Under Attack

This section presents an attack analysis of Castro and Liskov's BFT protocol [5], a leader-based Byzantine fault-tolerant replication protocol. We chose BFT because (1) it is a widely used protocol to which other Byzantine-resilient 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 throughput in fault-free executions 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.

BFT assigns a total order to client operations. The protocol requires $3f+1$ servers, where f is the maximum number of servers that may be Byzantine. An elected leader coordinates the protocol. 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 operations so that progress can safely resume in a new view.

A client sends its operation directly to the leader. The leader proposes a sequence number for the operation by broadcasting a PRE-PREPARE message, which contains the view number, the proposed sequence number, and the operation itself. Upon receiving the PRE-PREPARE, a non-leader server accepts the proposed assignment by broadcasting a PREPARE message. When a server collects the PRE-PREPARE and $2f$ corresponding PREPARE messages, it broadcasts a COMMIT message. A server globally orders the operation when it collects $2f+1$ COMMIT messages. Each server executes globally ordered operations according to sequence number.

2.1 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 a client operation 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 operations they have not yet executed and (2) the duration of these timeouts.

A malicious leader can ignore operations sent directly by clients. If a client's timeout expires before receiving a reply to its operation, it broadcasts the operation to all servers, which forward the operation to the leader. Each non-leader server maintains a FIFO queue of pending operations (i.e., those operations it has forwarded to the leader but has not yet executed). A server places a timeout on the execution of the first operation in its queue; that is, it expects to execute the operation within the timeout period. If the timeout expires, the server suspects the leader is faulty and votes to replace it. When a server executes the first operation in its 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 operation that is not the first in its 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. BFT does not provide a mechanism for reducing timeout values.

To retain its role as leader, the leader must prevent $f+1$ correct servers from voting to replace it. Thus, assuming a timeout value of T, a malicious leader can use the following attack to cause delay: (1) Choose a set, S, of $f+1$ correct servers, (2) For each server $i \in S$, maintain a FIFO queue of the operations forwarded by i, and (3) For each such queue, send a PRE-PREPARE containing the first operation on the queue every $T - \epsilon$ time units. This guarantees that the $f+1$ correct servers in S execute the first operation on their queue each timeout period. If these operations are all different, the fastest the leader would need to introduce operations 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 operation per timeout.

This attack exploits the fact that non-leader servers place timeouts only on the first operation in their queues. To understand the ramifications of placing a timeout on *all* pending operations, we consider a hypothetical protocol that is identical to BFT except that non-leader servers place a timeout on all pending operations. Suppose non-leader server i simultaneously forwards n operations to the leader. If server i sets a timeout on all n operations, then i will suspect the leader if the system fails to execute n operations per timeout period. Since the system has a maximal throughput, if n is sufficiently large, i 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 the global ordering.

Recent protocols [13], [14] attempt to mitigate the PRE-PREPARE attack by rotating the leader (an idea suggested in [15]). While these protocols allow good long-term throughput and avoid the scenario in which a faulty leader can degrade performance indefinitely, they do not guarantee that individual operations will be ordered in a timely manner. Prime takes a different approach, guaranteeing that the system eventually settles on a leader that is forced to propose an ordering on *all* operations in a timely manner. To meet this requirement, the leader needs only a bounded amount of incoming and outgoing bandwidth, which would not be the case if servers placed a timeout on all operations in BFT.

We note that BFT can be configured to use an optimization in which clients disseminate operations to all servers and the leader sends PRE-PREPARE messages containing hashes of the operations, thus limiting (but not bounding) the bandwidth requirements of the leader. However, when this optimization is used, faulty clients can repeatedly cause performance degradation by disseminating their operations only to $f+1$ of the $2f+1$ correct servers. This causes f correct servers to learn the ordering of an operation without having the operation itself, which prevents them from executing subsequent operations until they recover the missing operations.

2.2 Attack 2: Timeout Manipulation

BFT ensures safety regardless of network synchrony. However, while attacks that impact message delay (e.g., denial of service attacks) cannot cause inconsistency, they 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 server is the leader, then that leader will be able to slow the system down to a throughput of roughly $f+1$ operations per timeout T, where T is potentially very large, using the attack described in the previous section. This vulnerability stems from BFT's inability to reduce the timeout and adapt to the network conditions after the system stabilizes.

One might try to overcome this problem in several ways, such as by resetting the timeout when the system reaches a view in which progress occurs, or by adapting the timeout using a multiplicative increase and additive decrease mechanism. In the former approach, if the timeout is set too low originally, then it will be reset just when it reaches a large enough value. This may cause the system to experience long periods during which new operations cannot be executed, because leaders (even correct ones) continue to be suspected until the timeout becomes large enough again. The latter approach may be more effective but will be slow to adapt after periods of instability. As explained in Section 5.4, Prime adapts to changing network conditions and dynamically determines an acceptable level of timeliness based on the current latencies between correct servers.

3 System Model and Service Properties

We consider a system consisting of N servers and M clients (collectively called *processors*), which communicate by passing messages. Each server is uniquely identified from the set $\mathcal{R} = \{1, 2, \ldots, N\}$, and each client is uniquely identified from the set $\mathcal{S} = \{N+1, N+2, \ldots, N+M\}$. We assume a Byzantine fault model in which processors are either *correct* or *faulty*; correct processors follow the protocol specification exactly, while faulty processors can deviate from the protocol specification arbitrarily by sending any message at any time, subject to the cryptographic assumptions stated below. We assume that $N \geq 3f+1$, where f is an upper bound on the number of servers that may be faulty. For simplicity, we describe the protocol for the case when $N = 3f+1$. Any number of clients may be faulty.

We assume an asynchronous network, in which message delay for any message is unbounded. The system meets our safety criteria in all executions in which f or fewer servers are faulty. The system guarantees our liveness and performance properties only in subsets of the executions in which message delay satisfies certain constraints. For some of our analysis, we will be interested in the subset of executions that model Diff-Serv [16] with two traffic classes. To facilitate this modeling, we allow each correct processor to designate each message that it sends as either TIMELY or BOUNDED.

All messages sent between processors are digitally signed. We denote a message, m, signed by processor i as $\langle m \rangle_{\sigma_i}$. We assume that digital signatures are unforgeable without knowing a processor's private key. We also make use of a collision-resistant hash function, D, for computing message digests. We denote the digest of message m as D(m).

A client submits an *operation* to the system by sending it to one or more servers. Operations are classified as read-only (*queries*) and read/write (*updates*). Each client operation is signed. Each server produces a sequence of operations, $\{o_1, o_2, \ldots\}$, as its output. The output reflects the order in which the server executes client operations. When a server outputs an operation, it sends a reply containing the result of the operation to the client.

Safety: Server replies for operations submitted by correct clients are correct according to linearizability [17], as modified to cope with faulty clients [18]. Prime establishes a total order on client operations, as encapsulated in the following safety property:

DEFINITION 3.1: *Safety-S1* – In all executions in which f or fewer servers are faulty, the output sequences of two correct servers are identical, or one is a prefix of the other.

Liveness and Performance Properties: Before defining Prime's liveness and performance properties, which we call PRIME-LIVENESS and BOUNDED-DELAY, we first require several definitions:

DEFINITION 3.2: A *stable set* is a set of correct servers, *Stable*, such that $|Stable| \geq 2f+1$. We refer to the members of *Stable* as the *stable servers*.

DEFINITION 3.3: *Bounded-Variance(T, S, K)* – For each pair of servers, s and r, in S, there exists a value, Min_Latency(s, r), unknown to the servers, such that if s sends a message in traffic class T to r, it will arrive with delay $\Delta_{s,r}$, where Min_Latency(s, r) $\leq \Delta_{s,r} \leq$ Min_Latency(s, r) $* K$.

DEFINITION 3.4: *Eventual-Synchrony(T, S)* – Any message in traffic class T sent from server $s \in S$ to server $r \in S$ will arrive within some unknown bounded time.

We now specify the degrees of network stability needed for Prime to meet PRIME-LIVENESS and BOUNDED-DELAY, respectively:

DEFINITION 3.5: *Stability-S1* – Let T_{timely} be a traffic class containing all messages designated as TIMELY. Then there exists a stable set, S, a known network-specific constant, K_{Lat}, and a time, t, after which Bounded-Variance(T_{timely}, S, K_{Lat}) holds.

DEFINITION 3.6: *Stability-S2* – Let T_{timely} and $T_{bounded}$ be traffic classes containing messages designated as TIMELY and BOUNDED, respectively. Then there exists a stable set, S, a known network-specific constant, K_{Lat}, and a time, t, after which Bounded-Variance(T_{timely}, S, K_{Lat}) and Eventual-Synchrony($T_{bounded}$, S) hold.

We now state Prime's liveness and performance properties:

DEFINITION 3.7: PRIME-LIVENESS – If *Stability-S1* holds for a stable set, S, and no more than f servers are faulty, then if a server in S receives an operation from a correct client, the operation will eventually be executed by all servers in S.

DEFINITION 3.8: BOUNDED-DELAY – If *Stability-S2* holds for a stable set, S, and no more than f servers are faulty, then there exists a time after which the latency between a server in S receiving a client operation and all servers in S executing that operation is upper bounded.

Discussion: The degree of stability needed for liveness in Prime (i.e., *Stability-S1*) is incomparable with the degree of stability needed in BFT and similar protocols. BFT requires *Eventual-Synchrony* [2] to hold for *all* protocol messages, while Prime requires *Bounded-Variance* (a stronger degree of synchrony) but only for messages in the TIMELY traffic class. As described in Section 5, the TIMELY messages account for a small fraction of the total traffic, are only sent periodically, and have a small, bounded size. Prime is live even when messages in the BOUNDED traffic class arrive completely asynchronously.

In theory, it is possible for a strong network adversary capable of controlling the network variance to construct scenarios in which BFT is live and Prime is not. These scenarios occur when the variance for TIMELY messages becomes greater than K_{Lat}, yet the delay is still bounded. This can be made less likely to occur in practice by increasing K_{Lat}, although at the cost of giving a faulty leader more leeway to cause delay (see Section 5.4).

In practice, we believe both *Stability-S1* and *Stability-S2* can be made to hold. In well-provisioned local-area networks, network delay is often predictable and queuing is unlikely to occur. On wide-area networks, one could use a quality of service mechanism such as Diff-Serv [16], with one low-volume class for TIMELY messages and a second class for BOUNDED messages, to give *Bounded-Variance* sufficient coverage, provided enough bandwidth is available to pass the TIMELY messages without queuing. The required level of bandwidth is tunable and independent of the offered load; it is based only on the number of servers in the system and the rate at which the periodic messages are sent. Thus, in a well-engineered system, *Bounded-Variance* should hold for TIMELY messages, regardless of the offered load.

Finally, we remark that resource exhaustion denial of service attacks may cause *Stability-S2* to be violated for the duration of the attack. 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. Handling resource exhaustion attacks at the system-wide level is a difficult problem that is orthogonal and complementary to the solution strategies considered in this work.

4 Prime: Design and Overview

In any execution where the network is well behaved, Prime eventually bounds the time between when a client submits an operation to a correct server and when all of the correct servers execute the operation. The non-leader servers monitor the performance of the leader and replace any leader that is performing too slowly.

In existing leader-based protocols, variation in workload can give rise to legitimate variations in a leader's performance, which in turn make it hard or impossible for non-leaders to judge the leader's performance. In contrast, for any fixed system size, Prime assigns the leader a fixed amount of work (to do in its role as leader), independent of the load on the system. Thus, it is much easier for the non-leaders to judge the performance of the leader. Note that a leader in Prime is also assigned tasks unrelated to its role as leader. It must give priority to its leader tasks so that its performance as leader will be independent of system workload.

Prime reduces the amount of work assigned to the leader by offloading most of the tasks required to establish an order on client operations to other servers. Most offloaded tasks are done by all of the servers as a group, with the leader participating but playing no special role. In any execution where the network is well behaved, the progress of these tasks has bounded delay regardless of the behavior of Byzantine servers (including a Byzantine leader).

One offloaded task is handled differently. Much of the initial work to order and propagate each client operation is done by a sub-protocol coordinated by the server to which the operation was submitted. If the coordinator of this sub-protocol is correct, the sub-protocol will have bounded delay in executions where the network is well behaved. If instead the coordinator of the sub-protocol is Byzantine, there is no requirement on whether or when the sub-protocol completes (and, in turn, whether or when the submitted operation is ordered).

The view change protocol in Prime is also designed so that the leader has a fixed-size task and hence can have its performance accurately assessed by its peers.

4.1 Operation Ordering

A client requests the ordering of an operation by submitting it to a server. A server incrementally constructs a server-specific ordering of those operations that clients submit directly to it, and it assumes responsibility for disseminating the operations to the other servers. Each server periodically broadcasts a bounded-size *summary message* that indicates how much of each server's server-specific ordering this server has learned about.

The only thing that the current leader of the main protocol must do to build the global ordering of client operations is to incrementally construct an interleaving of the server-specific orderings. A correct leader periodically sends an *ordering message* containing the most recent summary message from each server. The ordering messages are of fixed size, and the extension to the global order is implicit in the set of ordering messages sent by the leader. The ordering message describes for each server a (possibly empty) window of additional operations from that server's server-specific order to add to the global order. The specified window always starts with the earliest operation from each server that has not yet been added to the global order. The window adds only those operations known widely enough among the correct servers so that eventually all correct servers will be able to learn what the operations are.

Because the leader's job of extending the global order requires a small, bounded amount of work, the non-leader servers can judge the leader's performance effectively. The non-leaders measure the round-trip times to each other to determine how long it should take between sending a summary to the leader and receiving a corresponding ordering message; we call this the *turnaround time* provided by the leader. Non-leaders also monitor that the leader is sending current summary messages. When a non-leader server sends a summary message to the leader, it can expect the leader's next ordering message to reflect at least as much information about the server-specific orderings as is contained in the summary.

4.2 Overview of Sub-Protocols

We now describe the sub-protocols in Prime.

Client Sub-Protocol: The Client sub-protocol defines how a client injects an operation into the system and collects replies from servers once the operation has been executed.

Preordering Sub-Protocol: The Preordering sub-protocol implements the server-specific orderings that are later interleaved by the leader to construct the global ordering. At all times a separate instance of the sub-protocol, coordinated by each server in the system, runs in order to process operations submitted to that server. The sub-protocol has three main functions. First, it is used to disseminate to $2f+1$ servers each client operation. Second, it binds each operation to a unique *preorder identifier*; we say that a server *preorders* an operation when it learns the operation's unique binding. Third, it summarizes each server's knowledge of the server-specific orderings by generating summary messages. A summary generated by server i contains a value, x, for each server j such that x is the longest gap-free prefix of the server-specific ordering generated by j that is known to i.

Global Ordering Sub-Protocol: The Global Ordering sub-protocol runs periodically and is used to incrementally extend the global order. The sub-protocol is coordinated by the current leader and, like BFT [5], establishes a total order on PRE-PREPARE messages. Instead of sending a PRE-PREPARE message containing client operations (or even operation identifiers) like in BFT, the leader in Prime sends a PRE-PREPARE message that contains a vector of at most $3f+1$ summary messages, each from a different server. The summaries contained in the totally ordered sequence of PRE-PREPARE messages induce a total order on the preordered operations.

To ensure that client operations known only to faulty processors will not be globally ordered, we define an operation as *eligible for execution* when the collection of summaries in a PRE-PREPARE message indicate that the operation has been preordered by at least $2f+1$ servers. An operation that is eligible for execution is known to enough correct servers so that all correct servers will eventually be able to execute it, regardless of the behavior of faulty servers and clients. Totally ordering a PRE-PREPARE extends the global order to include those operations that become eligible for the first time.

Reconciliation Sub-Protocol: The Reconciliation sub-protocol proactively recovers globally ordered operations known to some servers but not others. Because correct servers can only execute the gap-free prefix of globally ordered operations, this prevents faulty servers from blocking execution at some correct servers by intentionally failing to disseminate operations to them.

Suspect-Leader Sub-Protocol: The Suspect-Leader sub-protocol determines whether the leader is extending the global ordering in a timely manner. The servers measure the round-trip times to each other in order to compute two values. The first is an acceptable turnaround time that the leader should provide, computed as a function of the latencies between the correct servers in the system. The second is a measure of the turnaround time actually being provided by the leader since its election. Suspect-Leader guarantees that a leader will be replaced unless it provides an acceptable turnaround time to at least one correct server, and that at least $f+1$ correct servers will not be suspected.

Leader Election Sub-Protocol: When the current leader is suspected to be faulty by enough servers, the non-leader servers vote to elect a new leader. Each leader election is associated with a unique *view number*; the resulting configuration, in which one server is the leader and the rest are non-leaders, is called a *view*. The view number is increased by one and the new leader is chosen by rotation.

View Change Sub-Protocol: When a new leader is elected, the View Change sub-protocol wraps up in-progress activity from prior views by deciding which updates to the global order to process and which can be safely abandoned.

Sub-Protocol	Message Type	Traffic Class	Synchrony for Liveness?
Client	CLIENT-OP	BOUNDED	No
	CLIENT-REPLY	BOUNDED	No
Preordering	PO-REQUEST	BOUNDED	No
	PO-ACK	BOUNDED	No
	PO-SUMMARY	BOUNDED	No
Global Ordering	PRE-PREPARE (from leader only)	TIMELY	Yes
	PRE-PREPARE (flooded)	BOUNDED	No
	PREPARE	BOUNDED	No
	COMMIT	BOUNDED	No
Reconciliation	RECON	BOUNDED	No
	INQUIRY	BOUNDED	No
	CORRUPTION-PROOF	BOUNDED	No
Suspect-Leader	SUMMARY-MATRIX	TIMELY	Yes
	RTT-PING	TIMELY	Yes
	RTT-PONG	TIMELY	Yes
	RTT-MEASURE	BOUNDED	No
	TAT-UB	BOUNDED	No
	TAT-MEASURE	BOUNDED	No
Leader Election	NEW-LEADER	BOUNDED	No
	NEW-LEADER-PROOF	BOUNDED	No
View Change	REPORT	BOUNDED	No
	PC-SET	BOUNDED	No
	VC-LIST	BOUNDED	No
	REPLAY-PREPARE	BOUNDED	No
	REPLAY-COMMIT	BOUNDED	No
	VC-PROOF	TIMELY	Yes
	REPLAY	TIMELY	Yes

TABLE 1: Traffic class of each Prime message type.

5 PRIME: TECHNICAL DETAILS

This section describes the technical details of Prime. Due to space limitations, we only present the details of the Preordering, Global Ordering, Reconciliation, and Suspect-Leader sub-protocols. Details of the remaining sub-protocols can be found in Appendix A. Table 1 lists the types and traffic classes of all of Prime's messages.

5.1 The Preordering Sub-Protocol

The Preordering sub-protocol binds each client operation to a unique *preorder identifier*. The preorder identifier consists of a pair of integers, $(i,\ seq)$, where i is the identifier of the server that introduces the operation for preordering, and seq is a *preorder sequence number*, a local variable at i incremented each time it introduces an operation for preordering. Note that the preorder sequence number corresponds to server i's server-specific ordering.

Operation Dissemination and Binding: Upon receiving a client operation, o, server i broadcasts a $\langle$PO-REQUEST, $seq_i,\ o,\ i\rangle_{\sigma_i}$ message. The PO-REQUEST disseminates the client's operation and proposes that it be bound to the preorder identifier $(i,\ seq_i)$. When a server, j, receives the PO-REQUEST, it broadcasts a $\langle$PO-ACK, $i,\ seq_i,\ D(o),\ j\rangle_{\sigma_j}$ message if it has not previously received a PO-REQUEST from i with preorder sequence number seq_i.

A set consisting of a PO-REQUEST and $2f$ matching PO-ACK messages from different servers constitutes a *preorder certificate*. The preorder certificate proves that the preorder identifier $(i,\ seq_i)$ is uniquely bound to client operation o. We say that a server that collects a preorder certificate *preorders* the corresponding operation. Prime guarantees that if two servers bind operations o and o' to preorder identifier $(i,\ seq_i)$, then $o = o'$.

Summary Generation and Exchange: Each correct server maintains a local vector, *PreorderSummary[]*, indexed by server identifier. At correct server j, *PreorderSummary[i]* contains the maximum sequence number, n, such that j has preordered all operations bound to preorder identifiers $(i,\ seq)$, with $1 \leq seq \leq n$. For example, if server 1 has *PreorderSummary[]* $=$ $\{2,\ 1,\ 3,\ 0\}$, then server 1 has preordered the client

Let m_1 = $\langle$PO-SUMMARY, $vec_1,\ i\rangle_{\sigma_i}$
Let m_2 = $\langle$PO-SUMMARY, $vec_2,\ i\rangle_{\sigma_i}$

1. m_1 is **at least as up-to-date as** m_2 when
   - $(\forall j \in \mathcal{R})[vec_1[j] \geq vec_2[j]]$.
2. m_1 is **more up-to-date than** m_2 when
   - m_1 is at least as up to date as m_2, and
   - $(\exists j \in \mathcal{R})[vec_1[j] > vec_2[j]]$.
3. 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.

Fig. 1: Terminology used by the Preordering sub-protocol.

operations bound to preorder identifiers $(1, 1)$ and $(1,\ 2)$ from server 1; $(2,\ 1)$ from server 2; $(3,\ 1)$, $(3,\ 2)$, and $(3,\ 3)$ from server 3; and the server has not yet preordered any operations introduced by server 4.

Each correct server, i, periodically broadcasts the current state of its *PreorderSummary* vector by sending a $\langle$PO-SUMMARY, $vec,\ i\rangle_{\sigma_i}$ message. Note that the PO-SUMMARY message serves as a cumulative acknowledgement for preordered operations and is a short representation of every operation the sender has contiguously preordered (i.e., with no holes) from each server.

A key property of the Preordering sub-protocol is that if an operation is introduced for preordering by a correct server, the faulty servers cannot delay the time at which the operation is cumulatively acknowledged (in PO-SUMMARY messages) by at least $2f+1$ correct servers. This property holds because the rounds are driven by message exchanges between correct servers.

Each correct server stores the most *up-to-date* and *consistent* PO-SUMMARY messages that it has received from each server; these terms are defined formally in Fig. 1. Intuitively, two PO-SUMMARY messages from server i, containing vectors vec and vec', are consistent if either all of the entries in vec are greater than or equal to the corresponding entries in vec', or vice versa. Note that correct servers will never send inconsistent PO-SUMMARY messages, since entries in the *PreorderSummary* vector never decrease. Therefore, a pair of inconsistent PO-SUMMARY messages from the same server constitutes proof that the server is malicious. Each correct server, i, maintains a *Blacklist* data structure that stores the server identifiers of any servers from which i has collected inconsistent PO-SUMMARY messages.

The collected PO-SUMMARY messages are stored in a local vector, *LastPreorderSummaries[]*, indexed by server identifier. In Section 5.2, we show how the leader uses the contents of its own *LastPreorderSummaries* vector to propose an ordering on preordered operations. In Section 5.4, we show how the non-leaders' *LastPreorder-Summaries* vectors determine what they expect to see in the leader's ordering messages, thus allowing them to monitor the leader's performance.

5.2 The Global Ordering Sub-Protocol

Like BFT, the Global Ordering sub-protocol uses three message rounds: PRE-PREPARE, PREPARE, and COMMIT. In Prime, the PRE-PREPARE messages contain and propose a global order on *summary matrices*, not client operations. Each summary matrix is a vector of $3f+1$ PO-SUMMARY messages. The term "matrix" is used because each PO-SUMMARY message sent by a correct server itself contains a vector, with each entry reflecting the operations that have been preordered from each server. Thus, row i in summary matrix sm (denoted $sm[i]$) either contains a PO-SUMMARY message generated and signed by server i, or a special *empty* PO-SUMMARY message, containing a null vector of length $3f+1$, indicating that the server has not yet collected a PO-SUMMARY from server i. When indexing into a summary matrix, we let $sm[i][j]$ serve as shorthand for $sm[i].vec[j]$.

A correct leader, l, of view v periodically broadcasts a $\langle$PRE-PREPARE, $v,\ seq,\ sm,\ l\rangle_{\sigma_l}$ message, where seq is a global sequence number (analogous to the one assigned to PRE-PREPARE messages in BFT) and sm is the leader's *LastPreorderSummaries* vector. When correct server i receives a $\langle$PRE-PREPARE, $v,\ seq,\ sm,\ l\rangle_{\sigma_l}$ message, it takes the following steps. First, server i checks each PO-SUMMARY message in the summary matrix to see if it is consistent with what i has in its *LastPreorderSummaries* vector. Any server whose PO-SUMMARY is inconsistent is added to i's *Blacklist*. Second, i decides if it will respond to the PRE-PREPARE message using similar logic to the corresponding round in BFT. Specifically, i responds to the message if (1) v is the current view number and (2) i has not already processed a PRE-PREPARE in view v with the same sequence number but different content.

If i decides to respond to the PRE-PREPARE, it broadcasts a $\langle$PREPARE, $v,\ seq,\ D(sm),\ i\rangle_{\sigma_l}$ message, where v and seq correspond to the fields in the PRE-PREPARE and $D(sm)$ is a digest of the summary matrix found in the PRE-PREPARE. A set consisting of a PRE-PREPARE and $2f$ matching PREPARE messages constitutes a *prepare certificate*. Upon collecting a prepare certificate, server i broadcasts a $\langle$COMMIT, $v,\ seq,\ D(sm),\ i\rangle$ message. We say that a server *globally orders* a PRE-PREPARE when it collects $2f+1$ COMMIT messages that match the PRE-PREPARE.

Obtaining a Global Order on Client Operations: At any time at any correct server, the current outcome of the Global Ordering sub-protocol is a totally ordered stream of PRE-PREPARE messages: $T = \langle T_1, T_2, \ldots, T_x\rangle$. The stream at one correct server may be a prefix of the stream at another correct server, but correct servers do not have inconsistent streams.

We now explain how a correct server obtains a total order on client operations from its current local value of T. Let *mat* be a function that takes a PRE-PREPARE message and returns the summary matrix that it contains. Let M, a function from PRE-PREPARE messages to sets of preorder identifiers, be defined as:

$$\begin{aligned}M(T_y) = \{(i,\ seq) : i \in \mathcal{R} \wedge seq \in \mathcal{N} \wedge \\ |\{j : j \in \mathcal{R} \wedge mat(T_y)[j][i] \geq seq\}| \geq 2f+1\}\end{aligned}$$

Observe that any preorder identifier in $M(T_y)$ has been associated with a specific operation by at least $2f+1$ servers, of which at least $f+1$ are correct. The Preordering sub-protocol guarantees that this association is unique. The Reconciliation sub-protocol guarantees that any operation in $M(T_y)$ is known to enough correct servers so that in any sufficiently stable execution, any correct server that does not yet have the operation will eventually receive it (see Section 5.3). Note also that since PO-SUMMARY messages are cumulative acknowledgements, if $M(T_y)$ contains a preorder identifier $(i,\ seq)$, then $M(T_y)$ also contains all preorder identifiers of the form $(i,\ seq')$ for $1 \leq seq' < seq$.

Let L be a function that takes as input a set of preorder identifiers, P, and outputs the elements of P ordered lexicographically by their preorder identifiers, with the first element of the preorder identifier having the higher significance. Letting || denote concatenation and \ denote set difference, the final total order on clients operations is obtained by:

$$\begin{aligned} C_1 &= L(M(T_1)) \\ C_q &= L(M(T_q) \setminus M(T_{q-1})) \\ C &= C_1 \mid\mid C_2 \mid\mid \ldots \mid\mid C_x \end{aligned}$$

Intuitively, when a PRE-PREPARE is globally ordered, it expands the set of preordered operations that are *eligible for execution* to include all operations o for which the summary matrix in the PRE-PREPARE proves that at least $2f+1$ servers have preordered o. Thus, the set difference operation in the definition of the C_q components causes only those operations that have not already become eligible for execution to be executed.

Pre-Prepare Flooding: We now make a key observation about the Global Ordering sub-protocol: If all correct servers receive a copy of a PRE-PREPARE message, then there is nothing the faulty servers can do to prevent the PRE-PREPARE from being globally ordered in a timely manner. Progress in the PREPARE and COMMIT rounds is based on collecting sets of $2f+1$ messages. Therefore, since there are at least $2f+1$ correct servers, the correct servers are not dependent on messages from the faulty servers to complete the global ordering.

We leverage this property by having a correct server broadcast a PRE-PREPARE upon receiving it for the first time. This guarantees that all correct servers receive the PRE-PREPARE within one round from the time that the first correct server receives it, after which no faulty server can delay the correct servers from globally ordering it. The benefit of this approach is that it forces a malicious leader to delay sending a PRE-PREPARE to *all* correct servers in order to add unbounded delay to the Global Ordering sub-protocol. As described in Section 5.4, the Suspect-Leader sub-protocol results in the replacement of any leader that fails to send a timely PRE-PREPARE to at least one correct server. This property, combined with PRE-PREPARE flooding, will be used to ensure timely ordering.

In the course of PRE-PREPARE flooding, if a correct server, i, receives two PRE-PREPARE messages with the same view number and sequence number but different proof matrices, server i adds the leader to its *Blacklist* and invokes the Leader Election sub-protocol (see Appendix A). By periodically broadcasting the conflicting PRE-PREPARE messages, i will cause all correct servers to blacklist and suspect the leader, ensuring that a new leader is eventually elected.

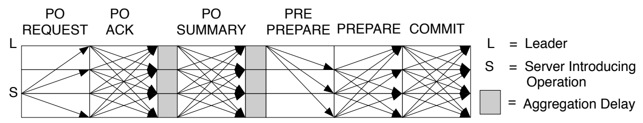

Fig. 2: Fault-free operation of Prime ($f = 1$).

Summary of Normal-Case Operation: To summarize the Preordering and Global Ordering sub-protocols, Fig. 2 follows the path of a client operation through the system during normal-case operation. The operation is first preordered in two rounds (PO-REQUEST and PO-ACK), after which its preordering is cumulatively acknowledged (PO-SUMMARY). When the leader is correct, it includes, in its next PRE-PREPARE, the set of at least $2f+1$ PO-SUMMARY messages that prove that at least $2f+1$ servers have preordered the operation. The PRE-PREPARE flooding step (not shown) runs in parallel with the PREPARE step. The client operation will be executed once the PRE-PREPARE is globally ordered. Note that in general, many operations are being preordered in parallel, and globally ordering a PRE-PREPARE will make many operations eligible for execution.

5.3 The Reconciliation Sub-Protocol

The Reconciliation sub-protocol ensures that all correct servers will eventually receive any operation that becomes eligible for execution. Conceptually, the protocol operates on the totally ordered sequence of operations defined by the total order $C = C_1 \mid\mid C_2 \mid\mid \ldots \mid\mid C_x$. Recall that each C_j is a sequence of preordered operations that became eligible for execution with the global ordering of pp_j, the PRE-PREPARE globally ordered with global sequence number j. From the way C_j is created, for each preordered operation $(i,\ seq)$ in C_j, there exists a set, $R_{i,seq}$, of at least $2f+1$ servers whose PO-SUMMARY messages cumulatively acknowledged $(i,\ seq)$ in pp_j. The protocol operates by having $2f+1$ deterministically chosen servers in $R_{i,seq}$ send *erasure encoded parts* of the PO-REQUEST containing $(i,\ seq)$ to those servers that have not cumulatively acknowledged preordering it.

Prime uses a Maximum Distance Separable erasure-resilient coding scheme [19], in which the PO-REQUEST is encoded into $2f+1$ parts, each $1/(f+1)$ the size of the original message, such that any $f+1$ parts are sufficient to decode. Each of the $2f+1$ servers in $R_{i,seq}$ sends one part. Since at most f servers are faulty, this guarantees that a correct server will receive enough parts to be able to decode the PO-REQUEST. The servers run the reconciliation procedure speculatively, when they first receive a PRE-PREPARE message, rather than when they globally order it. This proactive approach allows operations to be recovered in parallel with the remainder of the Global Ordering sub-protocol.

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

5.4 The Suspect-Leader Sub-Protocol

There are two types of performance attacks that can be mounted by a malicious leader. First, it can send PRE-PREPARE messages at a rate slower than the one specified by the protocol. Second, even if the leader sends PRE-PREPARE messages at the correct rate, it can intentionally include a summary matrix that does not contain the most up-to-date PO-SUMMARY messages that it has received. This can prevent or delay preordered operations from becoming eligible for execution.

The Suspect-Leader sub-protocol is designed to defend against these attacks. The protocol consists of three mechanisms that work together to enforce timely behavior from the leader. The first provides a means by which non-leader servers can tell the leader which PO-SUMMARY messages they expect the leader to include in a subsequent PRE-PREPARE message. The second mechanism allows the non-leader servers to periodically measure how long it takes for the leader to send a PRE-PREPARE containing PO-SUMMARY messages at least as up-to-date as those being reported. We call this time the *turnaround time* provided by the leader. The third mechanism is a *distributed monitoring protocol* in which the non-leader servers can dynamically determine, based on the current network conditions, how quickly the leader should be sending up-to-date PRE-PREPARE messages and decide, based on each server's measurements of the leader's performance, whether to suspect the leader. We now describe each mechanism in more detail.

5.4.1 *Reporting the Latest* PO-SUMMARY *Messages*

If the leader is to be expected to send PRE-PREPARE messages with the most up-to-date PO-SUMMARY messages, then each correct server must tell the leader which PO-SUMMARY messages it believes are the most up-to-date. This explicit notification is necessary because

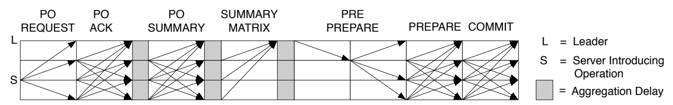

Fig. 3: Operation of Prime with a malicious leader that performs well enough to avoid being replaced ($f = 1$).

the reception of a particular PO-SUMMARY by a correct server does not imply that the leader will receive the same message—the server that originally sent the message may be faulty. Therefore, each correct server, i, periodically sends the leader the complete contents of its *LastPreorderSummaries* vector in a $\langle$SUMMARY-MATRIX, $sm,\ i\rangle_{\sigma_i}$ message.

Upon receiving a SUMMARY-MATRIX, a correct leader updates its *LastPreorderSummaries* vector by adopting any of the PO-SUMMARY messages in the SUMMARY-MATRIX that are more up-to-date than what the leader currently has in its data structure. Since SUMMARY-MATRIX messages have a bounded size dependent only on the number of servers in the system, the leader requires a small, bounded amount of incoming bandwidth and processing resources to learn about the most up-to-date PO-SUMMARY messages in the system. Furthermore, since PRE-PREPARE messages also have a bounded size independent of the offered load, the leader requires a bounded amount of outgoing bandwidth to send timely, up-to-date PRE-PREPARE messages.

5.4.2 *Measuring the Turnaround Time*

The preceding discussion suggests a way for non-leader servers to monitor the leader's performance effectively. Given that a correct leader can send timely, up-to-date PRE-PREPARE messages, a non-leader server can measure the time between sending a SUMMARY-MATRIX message, SM, to the leader and receiving a PRE-PREPARE that contains PO-SUMMARY messages that are at least as up-to-date as those in SM. This is the turnaround time provided by the leader. As described below, Suspect-Leader's distributed monitoring protocol forces any server that retains its role as leader to provide a timely turnaround time to at least one correct server. Combined with PRE-PREPARE flooding (see Section 5.2), this ensures that all eligible client operations will be globally ordered in a timely manner.

Fig. 3 depicts the maximum amount of delay that can be added by a malicious leader that performs well enough to avoid being replaced. The leader ignores PO-SUMMARY messages and sends its PRE-PREPARE to only one correct server. PRE-PREPARE flooding ensures that all correct servers receive the PRE-PREPARE within one round of the first correct server receiving it.

In order to define the notion of turnaround time more formally, we first define the *covers* predicate:

Let $pp = \langle$PRE-PREPARE, $*,\ *,\ sm, *\rangle_{\sigma_*}$
Let $SM = \langle$SUMMARY-MATRIX, $sm',\ *\rangle_{\sigma_*}$

Then *covers*$(pp,\ SM,\ i)$ is true at server i iff:
- $\forall j \in (\mathcal{R} \setminus \mathit{Blacklist}_i)$, $sm[j]$ is at least as up-to-date as $sm'[j]$.

Thus, server i is satisfied that a PRE-PREPARE *covers* a SUMMARY-MATRIX, SM, if, for all servers not in i's blacklist, each PO-SUMMARY in the PRE-PREPARE is at least as up-to-date (see Fig. 1) as the corresponding PO-SUMMARY in SM.

We can now define the turnaround time provided by the leader for a SUMMARY-MATRIX message, SM, sent by server i, as the time between i sending SM to the leader and i receiving a PRE-PREPARE that (1) covers SM, and (2) is for the next global sequence number for which i expects to receive a PRE-PREPARE. Note that the second condition establishes a connection between receiving an up-to-date PRE-PREPARE and actually being able to execute client operations once the PRE-PREPARE is globally ordered. Without this condition, a leader could provide fast turnaround times without this translating into fast global ordering.

5.4.3 The Distributed Monitoring Protocol

Before describing Suspect-Leader's distributed monitoring protocol, we first define what it means for a turnaround time to be timely. Timeliness is defined in terms of the current network conditions and the rate at which a correct leader would send PRE-PREPARE messages. In the definition that follows, we let L^*_{timely} denote the maximum latency for a TIMELY message sent between any two correct servers; Δ_{pp} denote a value greater than the maximum time between a correct server sending successive PRE-PREPARE messages; and K_{Lat} be a known network-specific constant accounting for latency variability.

PROPERTY 5.1: If Stability-S1 holds, then any server that retains a role as leader must provide a turnaround time to at least one correct server that is no more than $B = 2K_{Lat}L^*_{timely} + \Delta_{pp}$.

Property 5.1 ensures that a faulty leader will be suspected unless it provides a timely turnaround time to at least one correct server. We consider a turnaround time, $t \leq B$, to be timely because B is within a constant factor of the turnaround time that the slowest correct server might provide. The 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, and that increasing the value of K_{Lat} gives the leader more power to cause delay.

Of course, it is important to make sure that Suspect-Leader is not overly aggressive in the timeliness it requires from the leader. The following property ensures that this is the case:

PROPERTY 5.2: If Stability-S1 holds, then there exists a set of at least $f + 1$ correct servers that will not be suspected by any correct server if elected leader.

Property 5.2 ensures that when the network is sufficiently stable, 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.

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

/* TAT Measurement Task, run at server i */
B1. Periodically:
B2.   max_tat $\leftarrow$ Maximum TAT measured this view
B3.   BROADCAST: $\langle$TAT-MEASURE, view, max_tat, i$\rangle_{\sigma_i}$
B4. Upon receiving $\langle$TAT-MEASURE, view, tat, j$\rangle_{\sigma_j}$
B5.   if tat $>$ Reported_TATs[j]
B6.      Reported_TATs[j] $\leftarrow$ tat
B7.   $TAT_{leader}$ $\leftarrow (f+1)^{st}$ lowest val in Reported_TATs

/* RTT Measurement Task, run at server i */
C1. Periodically:
C2.   BROADCAST: $\langle$RTT-PING, view, ping_seq, i$\rangle_{\sigma_i}$
C3.   ping_seq++
C4. Upon receiving $\langle$RTT-PING, view, seq, j$\rangle_{\sigma_j}$:
C5.   SEND to server j: $\langle$RTT-PONG, view, seq, i$\rangle_{\sigma_i}$
C6. Upon receiving $\langle$RTT-PONG, view, seq, j$\rangle_{\sigma_j}$:
C7.   rtt $\leftarrow$ Measured RTT for pong message
C8.   SEND to server j: $\langle$RTT-MEASURE, view, rtt, i$\rangle_{\sigma_i}$
C9. Upon receiving $\langle$RTT-MEASURE, view, rtt, j$\rangle_{\sigma_j}$:
C10.  t $\leftarrow$ rtt * $K_{Lat} + \Delta_{pp}$
C11.  if t $<$ TATs_If_Leader[j]
C12.     TATs_If_Leader[j] $\leftarrow$ t

/* TAT_Leader Upper Bound Task, run at server i */
D1. Periodically:
D2.   $\alpha \leftarrow (f+1)^{st}$ highest val in TATs_If_Leader
D3.   BROADCAST: $\langle$TAT-UB, view, $\alpha$, i$\rangle_{\sigma_i}$
D4. Upon receiving $\langle$TAT-UB, view, tat_ub, j$\rangle_{\sigma_j}$:
D5.   if tat_ub $<$ TAT_Leader_UBs[j]
D6.      TAT_Leader_UBs[j] $\leftarrow$ tat_ub
D7.   $TAT_{acceptable}$ $\leftarrow (f+1)^{st}$ highest val in TAT_Leader_UBs

/* Suspect Leader Task */
E1. Periodically:
E2.   if $TAT_{leader}$ $>$ $TAT_{acceptable}$
E3.      Suspect Leader
```

Fig. 4: The Suspect-Leader distributed monitoring protocol.

We now present Suspect-Leader's distributed monitoring protocol, which allows non-leader servers to dynamically determine how fast a turnaround time the leader should provide and to suspect the leader if it is not providing a fast enough turnaround time to at least one correct server. Pseudocode is contained in Fig. 4.

The protocol is organized as several tasks that run in parallel, with the outcome being that each server decides whether or not to suspect the current leader. This decision is encapsulated in the comparison of two values: TAT_{leader} and $TAT_{acceptable}$ (see Fig. 4, lines E1-E3). TAT_{leader} is a measure of the leader's performance in the current view and is computed as a function of the turnaround times measured by the non-leader servers. $TAT_{acceptable}$ is a standard against which the server judges the current leader and is computed as a function of the round-trip times between correct servers. A server decides to suspect the leader if $TAT_{leader} > TAT_{acceptable}$.

As seen in Fig. 4, lines A1-A4, the data structures used in the distributed monitoring protocol are reinitialized at the beginning of each new view. Thus, a newly elected leader is judged using fresh measurements, both of what turnaround time it is providing and what turnaround time is acceptable given the current network conditions. The following two sections describe how TAT_{leader} and $TAT_{acceptable}$ are computed.

Computing TAT_{leader}: Each server keeps track of the maximum turnaround time provided by the leader in the current view and periodically broadcasts the value in a TAT-MEASURE message (Fig. 4, lines B1-B3). The values reported by other servers are stored in a vector, *Reported_TATs*, indexed by server identifier. TAT_{leader} is computed as the $(f+1)^{st}$ lowest value in *Reported_TATs* (line B7). Since at most f servers are faulty, TAT_{leader} is therefore a value v such that the leader is providing a turnaround time $t \leq v$ to at least one correct server.

Computing $TAT_{\textbf{acceptable}}$: Each server periodically runs a ping protocol to measure the RTT to every other server (Fig. 4, lines C1-C5). Upon computing the RTT to server j, server i sends the RTT measurement to j in an RTT-MEASURE message (line C8). When j receives the RTT measurement, it can compute the maximum turnaround time, t, that i would compute if j were the leader (line C10). Note that t is a function of the latency variability constant, K_{Lat}, as well as the rate at which a correct leader would send PRE-PREPARE messages. Server j stores the minimum such t in *TATs_If_Leader[i]*.

Each server, i, can use the values stored in *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. This upper bound is computed as the $(f+1)^{st}$ highest value in *TATs_If_Leader* (line D2). The servers periodically exchange their α values by broadcasting TAT-UB messages, storing the values in *TAT_Leader_UBs* (lines D5-D6). $TAT_{acceptable}$ is computed as the $(f+1)^{st}$ highest value in *TAT_Leader_UBs*.

We prove that Suspect-Leader meets Properties 5.1 and 5.2 in Appendix B.

6 Analysis

In this section we show that in those executions in which *Stability-S2* holds, Prime provides BOUNDED-DELAY (see Definition 3.8). As before, we let L^*_{timely} and $L^*_{bounded}$ denote the maximum message delay between correct servers for TIMELY and BOUNDED messages, respectively, and we let $B = 2K_{Lat}L^*_{timely} + \Delta_{pp}$. We also let Δ_{agg} denote a value greater than the maximum time between a correct server sending any of the following messages successively: PO-SUMMARY, SUMMARY-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 avoid being replaced. The time between a server receiving and introducing a client operation, o, for preordering and all correct servers sending SUMMARY-MATRIX messages containing at least $2f+1$ PO-SUMMARY messages that cumulatively acknowledge the preordering of o is at most three bounded 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 5.1, a leader that retains its role as leader must provide a TAT, $t \leq B$, to at least one correct server. By definition, $\Delta_{agg} \geq \Delta_{pp}$. Thus, $B \leq 2K_{Lat}L^*_{timely} + \Delta_{agg}$. Since correct servers flood PRE-PREPARE messages, all correct servers receive the PRE-PREPARE within three bounded rounds and one aggregation delay of when the SUMMARY-MATRIX messages are sent. All correct servers globally order the PRE-PREPARE in two bounded rounds from the time, t, the last correct server receives it. Reconciliation guarantees that all correct servers receive the PO-REQUEST containing the operation within one bounded round of time t. Summing the total delays yields a maximum latency of $\beta = 6L^*_{bounded} + 2K_{Lat}L^*_{timely} + 3\Delta_{agg}$.

If a malicious leader delays proposing an ordering, by more than B, on a summary matrix that proves that at least $2f+1$ servers preordered operation o, 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 A.3, a faulty leader will be suspected if it does not terminate the view change in a timely manner. Property 5.2 of Suspect-Leader guarantees that at most $2f$ view changes can occur before the system settles on a leader that will not be replaced. Therefore, there is a time after which the bound of β holds for any client operation received and introduced by a stable server.

7 Performance Evaluation

To evaluate the performance of Prime, we implemented the protocol and compared its performance to that of an available implementation of BFT. We show results evaluating the protocols in an emulated wide-area setting with 7 servers ($f = 2$). More extensive performance results, including results in a local-area setting, are presented in Appendix C.

Testbed and Network Setup: We used a system consisting of 7 servers, each running on a 3.2 GHz, 64-bit Intel Xeon computer. RSA signatures [20] provided authentication and non-repudiation. We used the netem utility [21] 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.

Clients submit one update operation to their local server, wait for proof that the update has been ordered, and then submit their next update. Updates contained 512 bytes of data. 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. Messages in BFT use message authentication codes for authentication.

Attack Strategies: Our experimental results during attack show the minimum performance that must be achieved in order for a malicious leader to avoid being replaced. 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

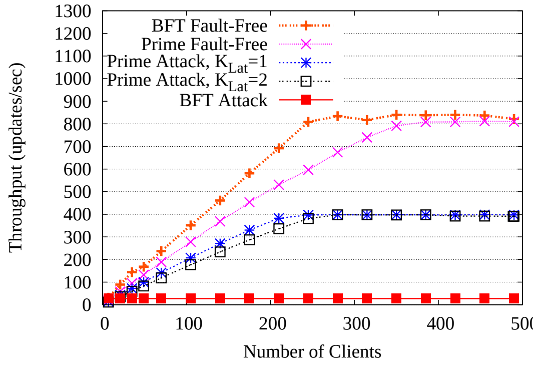

Fig. 5: Throughput of Prime and BFT as a function of the number of clients in a 7-server configuration.

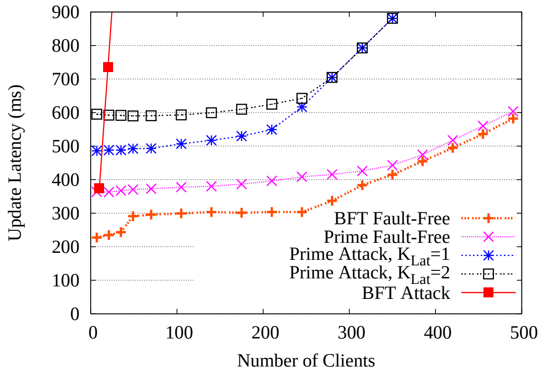

Fig. 6: Latency of Prime and BFT as a function of the number of clients in a 7-server configuration.

throughput. Therefore, we focus on the attack scenario where a malicious leader retains its role as leader indefinitely while degrading performance.

To attack Prime, the leader adds as much delay as possible (without being suspected) to the protocol, and faulty servers force as much reconciliation as possible. As described in Section 5.4, a malicious leader can add approximately two rounds of delay to the Global Ordering sub-protocol, plus an aggregation delay. The malicious servers force reconciliation by not sending their PO-REQUEST messages to f of the correct servers. Therefore, all PO-REQUEST messages originating from the faulty servers must be sent to these f correct servers using the Reconciliation sub-protocol (see Section 5.3). Moreover, the malicious servers only acknowledge each other's PO-REQUEST messages, forcing the correct servers to send reconciliation messages to them for all PO-REQUEST messages introduced by correct servers. Thus, all PO-REQUEST messages undergo a reconciliation step, which consumes approximately the same outgoing bandwidth as the dissemination of the PO-REQUEST messages during the Preordering sub-protocol.

To attack BFT, we use the attack described in Section 2.1. We present results for a very aggressive yet possible timeout (300 ms). This yields the most favorable performance for BFT under attack.

Results: Fig. 5 shows system throughput, measured in updates/sec, as a function of the number of clients in the emulated wide-area deployment. Fig. 6 shows the corresponding update latency, measured at the client. In the fault-free scenario, 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 fewer than 850 updates/sec, with about 250 clients. Prime reaches a similar plateau with about 350 clients. As seen in Fig. 6, BFT has a lower latency than Prime when the protocols are not under attack, 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 queuing at the servers.

The throughput results are different when the two protocols are attacked. With an aggressive timeout of 300 ms, BFT can order fewer than 30 updates/sec. With the default timeout of 5 seconds, BFT can only order 2 updates/sec (not shown). Prime plateaus at about 400 updates/sec due to the bandwidth overhead incurred by the Reconciliation sub-protocol. 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.

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 variability in latency (see Section 3). As K_{Lat} increases, a malicious leader can add more delay to the turnaround time without being detected. The amount of delay that can be added by a malicious leader is directly proportional to K_{Lat}. For example, if K_{Lat} were set to 10, the leader could add roughly 10 round-trip times of delay without being suspected. 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.

8 Related Work

This paper focused on leader-based Byzantine fault-tolerant protocols [5], [6], [7], [8], [9], [10], [11] that achieve replication via the state machine approach [22], [23]. 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 2.2. In contrast, Prime uses the Suspect-Leader sub-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. Aiyer et al. [15] first noted the problems that can be caused by a faulty primary and suggested rotating the primary to mitigate its attacks. Prime takes a different approach, enforcing timely behavior from any leader that remains in power and eventually settling on a leader that provides good performance. Singh et al. [24] demonstrate how the performance of different protocols can degrade under unfavorable network conditions.

More recently, the Aardvark system of Clement et al. [13] proposed building *robust* Byzantine replication systems that sacrifice some normal-case performance in order to ensure that performance remains acceptably high when the system exhibits Byzantine failures. The approaches taken by Prime and Aardvark are quite different. Prime aims to guarantee that *every* request known to correct servers will be executed in a timely manner, limiting the leader's responsibilities in order to enforce timeliness exactly where it is needed. Aardvark aims to guarantee that over sufficiently long periods, system throughput remains within a constant factor of what it would be if only correct servers were participating in the protocol. It achieves this by gradually increasing the level of work expected from the leader, which ensures that view changes take place. Aardvark guarantees high throughput when the system is saturated, but individual requests may take longer to execute (e.g., if they are introduced during the grace period that begins any view with a faulty primary). The Spinning protocol of Veronese et al. [14] constantly rotates the primary to reduce the impact of faulty servers.

Rampart [25] 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 requests to $2f+1$ servers before a coordinator assigns the global order. Drabkin et al. [26] observe the difficulty of setting timeouts in the context of group communication in malicious settings. Prime's Reconciliation sub-protocol uses erasure codes for efficient data dissemination. A similar approach was taken by Cachin and Tessaro [27] and Fitzi and Hirt [28].

Other Byzantine fault-tolerant protocols [3], [4], [29], [30] 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.

Byzantine quorum systems [31], [32], [33], [34] 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 state machine replication. The Q/U protocol [33] requires $5f + 1$ replicas for this purpose and suffers performance degradation when write contention occurs. The HQ protocol [34] 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 state machine replication is to use a hybrid architecture in which different parts of the system rely on different fault and/or timing assumptions [35], [36], [37]. The different components are therefore resilient to different types of attacks. We believe leveraging stronger timing assumptions may allow for more aggressive performance monitoring.

9 Conclusions

In this paper we pointed out the vulnerability of current leader-based intrusion-tolerant state machine replication protocols to performance degradation when under attack. We proposed the BOUNDED-DELAY correctness criterion to require consistent performance in all executions, even when the system exhibits Byzantine faults. We presented Prime, a new intrusion-tolerant state machine replication protocol, which meets BOUNDED-DELAY and is an important step towards making intrusion-tolerant replication resilient to performance attacks in malicious environments. Our experimental results show that Prime performs competitively with existing protocols in fault-free configurations and an order of magnitude better when under attack in the configurations tested.

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," *J. 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*, 1983, pp. 27–30.

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

[5] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[6] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, pp. 7:1–7:39, 2009.

[7] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling Byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2010.

[8] 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*, 2003, pp. 253–267.

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

[10] 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*, 2007, pp. 66–80.

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

[12] 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*, 2008, pp. 197–206.

[13] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009, pp. 153–168.

[14] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, "Spin one's wheels? Byzantine fault tolerance with a spinning primary," in *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 135–144.

[15] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "BAR fault tolerance for cooperative services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005, pp. 45–58.

[16] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, "An architecture for differentiated services," RFC 2475, 1998.

[17] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[18] M. Castro, "Practical Byzantine fault tolerance," Ph.D. dissertation, Massachusetts Institute of Technology, Cambridge, Massachusetts, 2001.

[19] F. J. MacWilliams and N. J. A. Sloane, *The theory of error-correcting codes*. New York, New York: North-Holland, 1988.

[20] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[21] "The netem Utility," http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

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

[23] 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.

[24] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "BFT protocols under fire," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008, pp. 189–204.

[25] M. K. Reiter, "The Rampart Toolkit for building high-integrity services," in *Lecture Notes Computer Science, Vol. 938: Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*. Springer-Verlag, 1995, pp. 99–110.

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

[27] C. Cachin and S. Tessaro, "Asynchronous verifiable information dispersal," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems)*, 2005, pp. 191–202.

[28] M. Fitzi and M. Hirt, "Optimally efficient multi-valued Byzantine agreement," in *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing*, 2006, pp. 163–168.

[29] C. Cachin and J. A. Portiz, "Secure intrusion-tolerant replication on the Internet," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2002, pp. 167–176.

[30] 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*, 2006, pp. 568–577.

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

[32] D. Malkhi and M. K. Reiter, "Secure and scalable replication in Phalanx," in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, 1998, pp. 51–58.

[33] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005, pp. 59–74.

[34] 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 USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 177–190.

[35] P. E. Veríssimo, N. F. 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.

[36] 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*, 2004, pp. 174–183.

[37] M. Serafini and N. Suri, "The fail-heterogeneous architectural model," in *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, 2007, pp. 103–113.

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

[39] "The BFT Project Homepage," http://www.pmg.csail.mit.edu/bft.

[40] R. C. Merkle, "Secrecy, authentication, and public key systems." Ph.D. dissertation, Stanford University, 1979.

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. 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 is a member of the ACM.

Jonathan Kirsch received a Ph.D. in Computer Science from Johns Hopkins University in 2010. He holds the B.Sc. degree from Yale University (2004) and the M.S.E. degree from Johns Hopkins University (2007). He is currently a Research Scientist at the Siemens Technology to Business Center in Berkeley, CA. His research interests include fault-tolerant replication and survivable systems. He is a member of the ACM and the IEEE Computing Society.

John Lane is a Senior Research Scientist at LiveTimeNet, where he has worked since 2008. He received his Ph.D. in Computer Science from Johns Hopkins University in 2008. 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. His research interests include distributed systems, replication, and byzantine fault tolerance. He is a member of the ACM and the IEEE Computing Society.

APPENDIX A ADDITIONAL PRIME SUB-PROTOCOLS

This appendix presents the details of three of Prime's sub-protocols: the Client sub-protocol, the Leader Election sub-protocol, and the View Change sub-protocol.

A.1 Client Sub-Protocol

A client, c, injects an operation into the system by sending a $\langle$CLIENT-OP, o, seq, $c\rangle_{\sigma_c}$ message, where o is the operation and seq is a client-specific sequence number, incremented each time the client submits an operation, used to ensure exactly-once semantics. The client sets a timeout, during which it waits to collect $f+1$ matching $\langle$CLIENT-REPLY, seq, res, $i\rangle_{\sigma_i}$ messages from different servers, where res is the result of executing the operation and i is the server's identifier.

There are several communication patterns that the client can use to inject its operation into the system. First, the client can initially send to one server. If the timeout expires, the client can send to another server, or to $f+1$ servers to ensure that the operation reaches a correct server. The client can keep track of the response times resulting from sending to different servers and, when deciding to which server it should send its next operation, the client can favor those that have provided the best average response times in the past. This approach is preferable in fault-free executions or when the system is bandwidth limited but has many clients, because it consumes the least bandwidth and will result in the highest system throughput. However, although clients will eventually settle on servers that provide good performance, any individual operation might be delayed if the client communicates with a faulty server.

To ensure that an operation is introduced by a server in a timely manner, the client can initially send its CLIENT-OP message to $f+1$ servers. This prevents faulty servers from causing delay but may result in the operation being ordered $f+1$ times. This is safe, because servers use the sequence number in the CLIENT-OP message to ensure that the operation is executed exactly once. While providing low latency, this communication pattern may result in lower system throughput because the system does more work per client operation. For this reason, this approach is preferable for truly time-sensitive operations or when the system has only a small number of clients.

Finally, we also note that when clients and servers are located on the same machine and hence share fate, the client can simply send the CLIENT-OP to its local server. In this case, the client can wait for a single reply: If the client's server is correct, then the client obtains a correct reply, while if the client's server is faulty, the client is considered faulty.

A.2 Leader Election Sub-Protocol

The Suspect-Leader sub-protocol provides a mechanism by which a correct server can decide whether to suspect the current leader as faulty. This section describes the Leader Election sub-protocol, which enables the servers to actually elect a new leader once the current leader is suspected by enough correct servers.

When server i suspects the leader of view v to be faulty (see Fig. 4, line E3), it broadcasts a $\langle$NEW-LEADER, $v+1$, $i\rangle_{\sigma_i}$ message, suggesting that the servers move to view $v+1$ and elect a new leader. However, server i continues to participate in all aspects of the protocol, including Suspect-Leader. A correct server only stops participating in view v when it collects $2f+1$ NEW-LEADER messages for a later view.

When server i receives a set, S, of $2f+1$ NEW-LEADER messages for the same view, v', where v' is later than i's current view, server i broadcasts the set of messages in a $\langle$NEW-LEADER-PROOF, v', S, $i\rangle_{\sigma_i}$ message and moves to view v'; we say that the server *preinstalls* view v'. Any server that receives a NEW-LEADER-PROOF message for a view later than its current view, v, immediately stops participating in view v and preinstalls view v'. It also broadcasts the NEW-LEADER-PROOF message for view v'. This ensures that all correct servers preinstall view v' within one round of the first correct server preinstalling view v'. When a server preinstalls view v', it begins running the View Change sub-protocol described in Section A.3.

The reason why a correct server continues to participate in view v even after suspecting the leader of view v is to prevent a scenario in which a leader retains its role as leader (by sending timely, up-to-date PRE-PREPARE messages to enough correct servers) but the servers are unable to globally order the PRE-PREPARE messages. If a correct server could become silent in view v without knowing that a new leader will be elected, then if the leader does retain its role and the faulty servers become silent, the PRE-PREPARE messages would not be able to garner $2f+1$ PREPARE messages and ultimately be globally ordered. The approach taken by the Leader Election sub-protocol is similar to the one used by Zyzzyva [6], where correct servers continue to participate in a view until they collect $f+1$ I-HATE-THE-PRIMARY messages.

Note that the messages sent in the Leader Election sub-protocol are in the BOUNDED traffic class. In particular, they do not require synchrony for Prime to meet its liveness guarantee. The Leader Election sub-protocol uses the reception of messages, and not timeouts, to clock the progress of the protocol. As described in Section A.3, the View Change sub-protocol also uses the reception of messages to clock the progress of the protocol, except for the last step, where messages must be timely and the servers resume running Suspect-Leader to ensure that the protocol terminates without delay.

A.3 View Change Sub-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, Zyzzyva, and other existing leader-based protocols.

A.3.1 Background: 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 operations 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 operation 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.

A.3.2 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 different approach for its view change protocol. Whereas BFT's protocol is primarily 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., [38]). 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 longer.

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.

We note that instead of generating a threshold-signed proof, the leader could also include a set of $2f+1$ signed messages to prove the correctness of the REPLAY message. While this may be conceptually simpler and somewhat less computationally expensive, using threshold signatures has the desirable property that the resulting proof is compact and can fit in a single physical message, which may allow for more effective performance monitoring in bandwidth-constrained environments. Both types of proof provide the same level of guarantee regarding the correctness of the REPLAY message.

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 2 summarizes Prime's view change protocol.

A.3.3 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 all received up to*, or *execution ARU*, value.

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 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 takes 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 the Suspect-Leader distributed monitoring protocol, treating the VC-PROOF message just as it would a SUMMARY-MATRIX in computing the maximum turnaround time provided by the leader in the current view (see Algorithm 4, lines 9-15). 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 in the Global Ordering sub-protocol.

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,

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: All:	Upon receiving VC-PROOF, broadcast REPLAY message Agree on REPLAY	Committing REPLAY and collecting associated state	Execute all operations in replay window	Leader, monitored by Suspect-Leader

TABLE 2: Summary of Prime's view change protocol.

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-PREPARE 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 consequence of the fact that a correct server stops its timer after receiving any valid REPLAY message is that a faulty leader that sends conflicting REPLAY messages can 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 suspects the leader upon collecting this proof. Thus, the system will replace the faulty leader, and the detection time is a function of the latency between correct servers.

Appendix B
Suspect-Leader Proofs

In this appendix we prove a series of claims that allow us to prove Properties 5.1 and 5.2 of the Suspect-Leader sub-protocol. We first prove the following two lemmas, which place an upper bound on the value of $TAT_{acceptable}$.

Lemma *B.1:* Once a correct server, i, receives an RTT-MEASURE message from each correct server in view v, it will compute upper bound values, α, such that $\alpha \leq B = 2K_{Lat}L^*_{timely} + \Delta_{pp}$.

Proof: From Fig. 4 line A1, each entry in *TATs_If_Leader* is initialized to infinity at the beginning of each view. Thus, when server i receives the first RTT-MEASURE message from each other correct server, j, in view v, it will store an appropriate measurement in *TATs_If_Leader*[j] (lines C11-C12). Therefore, since there are at least $2f+1$ correct servers, at least $2f+1$ cells in server i's *TATs_If_Leader* vector eventually contain values, v, based on measurements 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 *TATs_If_Leader* is from a correct server and thus less than or equal to B. Server i computes its upper bound, α, as the minimum of these $f+1$ highest values (line D2), and thus $\alpha \leq B$. □

Lemma *B.2:* Once a correct server, i, receives a TAT-UB message, m, from each correct server, j, where m was sent after j collected an RTT-MEASURE message from each correct server, it will compute $TAT_{acceptable} \leq B = 2K_{Lat}L^*_{timely} + \Delta_{pp}$.

Proof: By Lemma B.1, once a correct server, j, receives an RTT-MEASURE message from each correct server in view v, it will compute upper bound values

$\alpha \leq B$. Call the time at which a correct server receives these RTT-MEASURE messages t. Any α value sent by this server before t will be greater than or equal to the first α value sent after t: α is chosen as the $(f+1)^{st}$ highest value in *TATs_If_Leader*, and the values in *TATs_If_Leader* only decrease. Thus, for each server, k, server i will store the first α value that k sends after time t (lines D5-D6). This implies that 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. At least one of the $f+1$ highest values in *TAT_Leader_UBs* 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 D7), and thus $TAT_{acceptable} \leq B$. □

We can now prove Property 5.1:

Proof: A server retains its role as leader unless at least $2f+1$ servers suspect it. Thus, if a leader retains its role, there are at least $f+1$ servers (at least one of which is correct) for which $TAT_{leader} \leq TAT_{acceptable}$ always holds. Call this correct server i. During view v, server i eventually collects TAT-MEASURE messages from at least $2f+1$ correct servers. If the faulty servers either do not send TAT-MEASURE messages or report turnaround times of zero, then TAT_{leader} is computed as a value from a correct server. Otherwise, at least one of the $(f+1)$ lowest entries is from a correct server, and thus there exists a correct server being provided a turnaround time $t \leq TAT_{leader}$. In both cases, by Lemma B.2, there exists at least one correct server being provided a turnaround time $t \leq TAT_{leader} \leq TAT_{acceptable} \leq B$. □

Now that we have shown that malicious servers that retain their role as leader must provide a timely turnaround time to at least one correct server, it remains to be shown that Suspect-Leader is not overly aggressive, and that some correct servers will be able to avoid being replaced. This is encapsulated in Property 5.2. Before proving Property 5.2, we prove the following lemma:

Lemma *B.3:* If a correct server, i, sends an upper bound value, α, then if i is elected leader, any correct server will compute $TAT_{leader} \leq \alpha$.

Proof: At server i, *TATs_If_Leader[j]* stores the maximum turnaround time, *max_tat*, that j would compute if i were leader. Thus, when i is leader, j will send TAT-MEASURE messages that report a turnaround time no greater than *max_tat*. Since α is chosen as the $(f+1)^{st}$ highest value in *TATs_If_Leader*, $2f+1$ servers (at least $f+1$ of which are correct) will send TAT-MEASURE messages that report values less than or equal to α when i is leader. Since the entries in *Reported_TATs* are initialized to zero (line A3), TAT_{leader} will be computed as zero until TAT-MEASURE messages from at least $2f+1$ servers are received. Since any two sets of $2f+1$ servers intersect on one correct server, the $(f+1)^{st}$ lowest value in *Reported_TATs* will never be more than α. Thus if server i were leader, any correct server would compute $TAT_{leader} \leq \alpha$. □

We can now prove Property 5.2:

Proof: Since $TAT_{acceptable}$ is the $(f+1)^{st}$ highest α value in *TAT_Leader_UBs*, at least $2f+1$ servers (at least $f+1$ of which are correct) sent α values such that $\alpha \leq TAT_{acceptable}$. By Lemma B.3, when each such correct server is elected leader, all other correct servers will compute $TAT_{leader} \leq \alpha$. Since $\alpha \leq TAT_{acceptable}$, each of these correct servers will not be suspected. □

APPENDIX C
ADDITIONAL PERFORMANCE RESULTS

In this appendix we present additional performance results for both Prime and BFT. We first show the performance in a 4-server configuration over an emulated wide-area deployment. We then present results in a local-area network setting.

The experiments were run on the same cluster of machines described in Section 7. We used a cluster of 3.2 GHz, 64-bit Intel Xeon computers. RSA signatures [20] provided authentication and non-repudiation in Prime. Each computer can compute a 1024-bit RSA signature in 1.3 ms and verify it in 0.07 ms.

C.1 Wide-Area Network Deployment

We evaluated Prime and BFT in the same emulated wide-area deployment as the one described in Section 7, in which servers were connected by 50 ms, 10 Mbps links. Clients submitted updates containing 512 bytes of data.

Fig. 7 shows system throughput, measured in updates per second, as a function of the number of clients, and Fig. 8 shows the corresponding latency. The throughput trends for the 4-server configuration are similar to the trends in the 7-server configuration. When not under attack, both Prime and BFT plateau at higher throughputs than those shown in the 7-server configuration (Fig. 5). 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.

C.2 Local-Area Network Deployment

We also tested Prime and BFT on a local-area network in which servers communicated via a Gigabit switch. In the local-area deployment, we used updates containing null

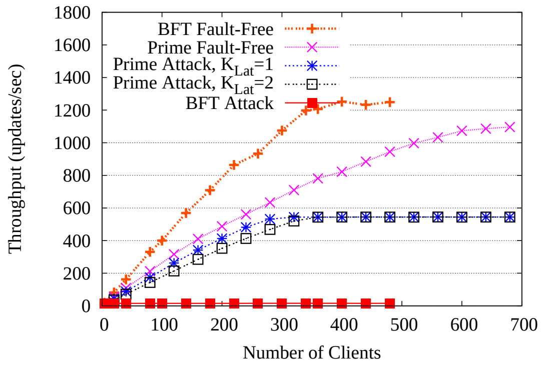

Fig. 7: Throughput of Prime and BFT as a function of the number of clients in a 4-server configuration. Servers were connected by 50 ms, 10 Mbps links.

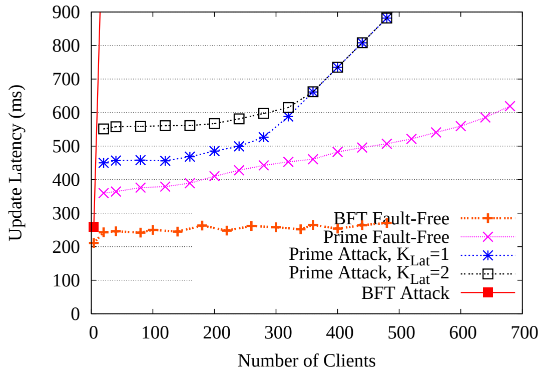

Fig. 8: Latency of Prime and BFT as a function of the number of clients in a 4-server configuration. Servers were connected by 50 ms, 10 Mbps links.

operations (i.e., 0 bytes of data) to match the way these protocols are commonly evaluated (e.g., [5], [13]). Taking into account signature overhead and other update-specific content, each update consumed a total of 162 bytes.

To attack Prime, we used the same attacks as those mounted in the wide-area setting. For BFT, we show results for two aggressive timeouts (5 ms and 10 ms). We used the original distribution of BFT [39] for all tests. Unfortunately, the original distribution becomes unstable when run at high throughputs, so we were unable to get results for BFT in a fault-free execution in the LAN setting. Results using an updated implementation were recently reported in [13] and [6], but we were unable to get the new implementation to build on our cluster. We base our analysis on the assumption that the newer implementation would obtain similar results on our own cluster.

Fig. 9 shows the throughput of Prime as a function of the number of clients in the LAN deployment, and Fig. 10 shows the corresponding latency. When not under attack, Prime becomes CPU constrained at a throughput of approximately 12,500 null operations per second. Latency remains below 100 ms with approximately 1200 clients.

When deployed on a LAN, our implementation of Prime uses Merkle trees [40] to amortize the cost of generating digital signatures over many messages. Although we could have used this technique for the WAN experiments, doing so does not significantly impact throughput or latency, because the system is bandwidth constrained rather than CPU constrained. Combined with the aggregation techniques built into Prime, a single digital signature covers many messages, significantly reducing the overhead of signature generation. In fact, since our implementation utilizes only a single CPU, and since verifying client signatures takes 0.07 ms, the maximum throughput that could be achieved is just over 14,000 updates per second (if the only operation performed were verifying client signatures). This implies that (1) signature aggregation is effective in improving peak throughput and (2) the peak throughput of Prime could be significantly improved by offloading cryptographic operations (specifically, signature verification) to a second processor (or to multiple cores), as is done in the recent implementation of the Aardvark protocol [13].

As Fig. 9 demonstrates, the performance of Prime under attack is quite different on a LAN compared to a WAN. We separated the delay attacks from the reconciliation attacks so their effects could be seen more clearly. Note that the reconciliation attack, which degraded throughput by approximately a factor of 2 in a wide-area environment, has very little impact on throughput on a LAN because the erasure encoding operations are inexpensive and bandwidth is plentiful.

In our implementation, the leader is expected to send a PRE-PREPARE every 30 ms. On a local-area network, the duration of this aggregation delay dominates any variability in network latency. Recall that in Suspect-Leader, a non-leader server computes the maximum turnaround time as $t = rtt * K_{Lat} + \Delta_{pp}$, where rtt is the measured round-trip time and Δ_{pp} is a value greater than the maximum time it might take a correct server to send a PRE-PREPARE (see Fig. 4, line C10). We ran Prime with two different values of Δ_{pp}: 40 ms and 50 ms. A malicious leader only includes a SUMMARY-MATRIX in its current PRE-PREPARE if it determines that including the SUMMARY-MATRIX in the next PRE-PREPARE (sent 30 ms in the future) would potentially cause the leader to be suspected, given the value of Δ_{pp}. Figures 9 and 10 show that the leader's attempts to add delay only increase latency slightly, by about 15 ms and 25 ms, respectively. As expected, the attacks do not impact peak throughput.

As noted above, the implementation of BFT that we tested does not work well when run at high speeds; the servers begin to lose messages due to a lack of sufficient flow control, and some of the servers crash. Therefore, we were unable to generate results for fault-free executions. Recently published results on a newer implementation report peak throughputs of approximately 60,000

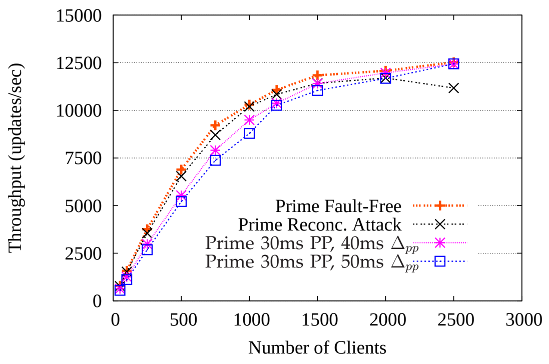

Fig. 9: Throughput of Prime as a function of the number of clients in a 7-server, local-area network configuration.

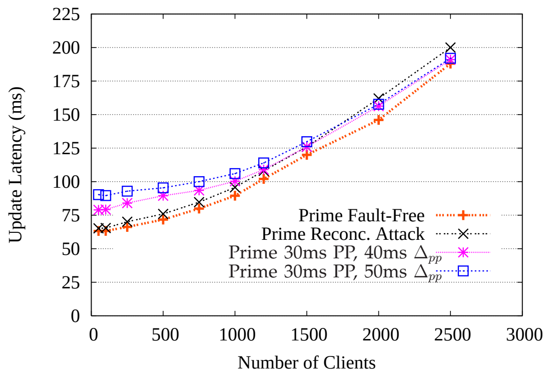

Fig. 10: Latency of Prime as a function of the number of clients in a 7-server, local-area network configuration.

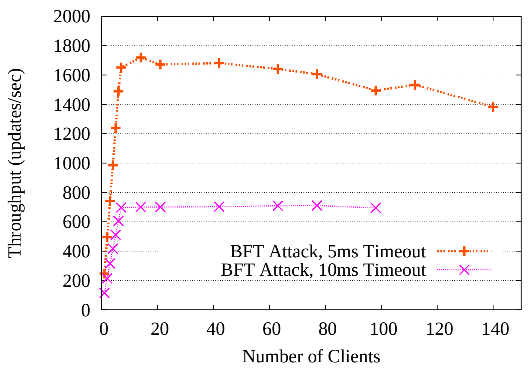

Fig. 11: Throughput of BFT in under-attack executions as a function of the number of clients in a 7-server, local-area network configuration.

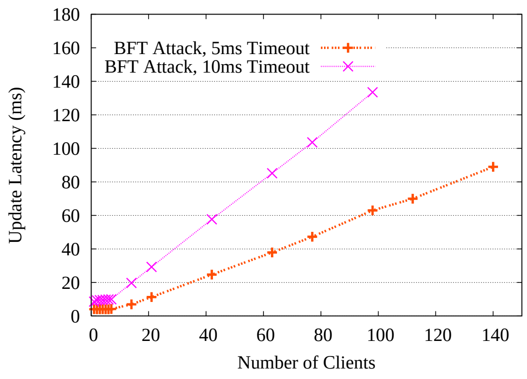

Fig. 12: Latency of BFT in under-attack executions as a function of the number of clients in a 7-server, local-area network configuration.

0-byte updates/sec and 32,000 updates/sec when client operations are authenticated using vectors of message authentication codes and digital signatures, respectively. Latency remains low, on the order of 1 ms or below, until the system becomes saturated. As noted in [18] and [13], when MACs are used for authenticating client operations, faulty clients can cause view changes in BFT when their operations are not properly authenticated. As explained above, if BFT used the same signature scheme as in Prime, it could only achieve peak throughputs higher than 14,000 updates/sec if it utilized more than one processor or core. While the peak throughputs of BFT and Prime are likely to be comparable in well-engineered implementations of both protocols, BFT is likely to have significantly lower operation latency than Prime in fault-free executions. This reflects the latency impact in Prime of both sending certain messages periodically and using more rounds requiring signed messages to be sent. Nevertheless, we believe the absolute latency values for Prime are likely to be low enough for many applications.

Figures 11 and 12 show the performance of BFT when under attack. With a 5 ms timeout, BFT achieved a peak throughput at approximately 1700 updates per second. With a 10 ms timeout, the peak throughput is approximately 750 updates/sec. As expected, throughput plateaus and latency begins to rise when there are more than 7 clients, when BFT is using only a small percentage of the CPU. As the graphs show, Prime's operation latency under attack will be less than BFT's once the number of clients exceeds approximately 100. When less aggressive timeouts are used in BFT, Prime's latency under attack will be lower than BFT's for smaller numbers of clients.