Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks

Yair Amir, *Member, IEEE,* Claudiu Danilov, Danny Dolev, *Senior Member, IEEE*
Jonathan Kirsch, *Member, IEEE,* John Lane, *Member, IEEE,* Cristina Nita-Rotaru, *Member, IEEE,*
Josh Olsen, and David Zage, *Member, IEEE*

Abstract—This paper presents the first hierarchical Byzantine fault-tolerant replication architecture suitable to systems that span multiple wide area sites. The architecture confines the effects of any malicious replica to its local site, reduces message complexity of wide area communication, and allows read-only queries to be performed locally within a site for the price of additional standard hardware. We present proofs that our algorithm provides safety and liveness properties. A prototype implementation is evaluated over several network topologies and is compared with a flat Byzantine fault-tolerant approach. The experimental results show considerable improvement over flat Byzantine replication algorithms, bringing the performance of Byzantine replication closer to existing benign fault-tolerant replication techniques over wide area networks.

Index Terms—Fault tolerance, scalability, wide area networks

1 Introduction

During the last few years, there has been considerable progress in the design of Byzantine fault-tolerant replication systems. Current state of the art protocols perform very well on small-scale systems that are usually confined to local area networks, which have small latencies and do not experience frequent network partitions. However, current solutions employ flat architectures that have several limitations: Message complexity limits their ability to scale, and strong connectivity requirements limit their availability on wide area networks, which usually have lower bandwidth, higher latency, and exhibit more frequent network partitions.

This paper presents Steward [1], the first hierarchical Byzantine fault-tolerant replication architecture suitable for systems that span multiple wide area sites, each consisting of several server replicas. Steward assumes no trusted component in the entire system other than a mechanism to pre-distribute private/public keys.

Steward uses Byzantine fault-tolerant protocols within each site and a lightweight, benign fault-tolerant protocol among wide area sites. Each site, consisting of several potentially malicious replicas, is converted into a single logical trusted participant in the wide area fault-tolerant protocol. Servers within a site run a Byzantine agreement protocol to agree upon the content of any message leaving the site for the global protocol.

Guaranteeing a consistent agreement within a site is not enough. The protocol needs to eliminate the ability of malicious replicas to misrepresent decisions that took place in their site. To that end, messages between servers at different sites carry a threshold signature attesting that enough servers at the originating site agreed with the content of the message. This allows Steward to save the space and computation associated with sending and verifying multiple individual signatures. Moreover, it allows for a practical key management scheme where all servers need to know only a single public key for each remote site and not the individual public keys of all remote servers.

Steward's hierarchical architecture reduces the message complexity on wide area exchanges from $O(N^2)$ (N being the total number of replicas in the system) to $O(S^2)$ (S being the number of wide area sites), considerably increasing the system's ability to scale. It confines the effects of any malicious replica to its local site, enabling the use of a benign fault-tolerant algorithm over the wide area network. This improves the availability of the system over wide area networks that are prone to partitions. Only a majority of connected sites is needed to make progress, compared with at least $2f+1$ servers (out of $3f+1$) in flat Byzantine architectures, where f is the upper bound on the number of malicious servers.

Steward allows read-only queries to be performed

- *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.*
- *Claudiu Danilov is with Boeing Phantom Works, P.O. Box 3707 MC 7L-49, Seattle, WA 98124. Email: claudiu.b.danilov@boeing.com.*
- *Danny Dolev is with the School of Engineering and Computer Science, The Hebrew University, Edmond Safra campus - Givat Ram, Jerusalem 91904, Israel. Email: dolev@cs.huji.ac.il.*
- *Cristina Nita-Rotaru and David Zage are with the Department of Computer Science, Purdue University, 305 N. University St., West Lafayette, IN 47907. Email: crisn@cs.purdue.edu, zage@purdue.edu.*
- *Josh Olsen is with the Donald Bren School of Information and Computer Sciences, University of California, Irvine, 6210 Donald Bren Hall, Irvine, CA 92697. Email: jolsen@ics.uci.edu.*

Manuscript received XX Month 2006; revised XX Month 2008; accepted XX Month 2008; published online XX Month. 2008.
For information on obtaining reprints of this article, please send email to: tdsc@computer.org, and reference IEEECS Log Number XX. Digitial Object Idenftifer no. XX.

Digital Object Indentifier 10.1109/TDSC.2008.53

locally within a site, enabling the system to continue serving read-only requests even in sites that are partitioned away. These local queries provide one-copy serializability [2], the common semantics provided by database products. Serializability is a weaker guarantee than the linearizability semantics [3] provided by some existing flat protocols (e.g., [4]). We believe serializability is the desired semantics in partitionable environments, because systems that provide linearizability can only answer queries in sites connected to a quorum. In addition, Steward can guarantee linearizability by querying a majority of the wide area sites, at the cost of higher latency and lower availability.

Steward provides the benefits described above by using an increased number of servers. More specifically, if the requirement is to protect against *any* f Byzantine servers in the system, Steward requires $3f + 1$ servers in each site. However, in return, it can overcome up to f malicious servers in *each* site. We believe this requirement is reasonable given the cost associated with computers today.

Steward's efficacy depends on using servers within a site that are unlikely to suffer correlated vulnerabilities. Multi-version programming [5], where independently coded software implementations are run on each server, can yield the desired diversity. Newer techniques [6], [7] can automatically and inexpensively generate variation. Steward remains vulnerable to attacks that compromise an entire site (e.g., by a malicious administrator with access to the site). This problem was addressed in [8].

The paper demonstrates that the performance of Steward with $3f + 1$ servers in *each site* is much better even compared with a flat Byzantine architecture with a smaller system of $3f + 1$ *total* servers spread over the same wide area topology. The paper further demonstrates that Steward exhibits performance comparable (though somewhat lower) with common benign fault-tolerant protocols on wide area networks.

We implemented the Steward system, and a DARPA red-team experiment has confirmed its practical survivability in the face of white-box attacks (where the red-team has complete knowledge of system design, access to its source code, and control of f replicas in each site). According to the rules of engagement, where a red-team attack succeeded only if it stopped progress or caused inconsistency, no attacks succeeded.

The main contributions of this paper are:

1) It presents the first hierarchical architecture and algorithm that scales Byzantine fault-tolerant replication to large, wide area networks.
2) It provides a complete proof of correctness for this algorithm, demonstrating its safety and liveness properties.
3) It presents a software artifact that implements the algorithm completely.
4) It shows the performance evaluation of the implementation software and compares it with the current state of the art. The experiments demonstrate that the hierarchical approach greatly outperforms existing solutions when deployed on large, wide area networks.

The remainder of the paper is organized as follows. We discuss previous work in several related research areas in Section 2. We provide background in Section 3. We present our system model in Section 4 and the service properties met by our protocol in Section 5. We describe our protocol, Steward, in Section 6. We present experimental results demonstrating the improved scalability of Steward on wide area networks in Section 7. We include a proof of safety and a proof roadmap of liveness in Section 8. We summarize our conclusions in Section 9. Appendix A contains complete pseudocode for our protocol, and complete correctness proofs can be found in Appendix B. The appendices appear in the electronic version of this paper, available from IEEE and at http://dsn.jhu.edu.

2 Related Work

Agreement and Consensus: At the core of many replication protocols is a more general problem, known as the agreement or consensus problem. A good overview of significant results is presented in [9]. The strongest fault model that researchers consider is the Byzantine model, where some participants behave in an arbitrary manner. If communication is not authenticated and nodes are directly connected, $3f+1$ participants and $f+1$ communication rounds are required to tolerate f Byzantine faults. If authentication is available, the number of participants can be reduced to $f + 2$ [10].

Fail-Stop Processors: Schlichting and Schneider [11] present the implementation and use of k-fail-stop processors, which consist of several potentially Byzantine processors. A k-fail-stop processor behaves like a fail-stop processor as long as no more than k processors are Byzantine faulty. Benign fault-tolerant protocols can thus safely run on top of these logical processors. Unlike Steward, in which a site is live unless f+1 of its computers fail, the k-fail-stop processor described in [11] halts when even one of its constituent processors fails.

Byzantine Group Communication: Related with our work are group communication systems resilient to Byzantine failures. Two such systems are Rampart [12] and SecureRing [13]. Both systems rely on failure detectors to determine which replicas are faulty. An attacker can slow correct replicas or the communication between them until a view is installed with less than two-thirds correct members, at which point safety may be violated. The ITUA system [14], [15], developed by BBN and UIUC, employs Byzantine fault-tolerant protocols to provide intrusion-tolerant group services. The approach taken considers all participants as equal and is able to tolerate up to less than a third of malicious participants.

Replication with Benign Faults: The two-phase commit (2PC) protocol [16] provides serializability in a distributed database system when transactions may span several sites. It is commonly used to synchronize transactions in a replicated database. Three-phase commit [17] overcomes some of the availability problems of 2PC, paying the price of an additional communication round. Paxos [18], [19] is a very robust algorithm for benign fault-tolerant replication and is described in Section 3.

Replication with Byzantine Faults: The first practical Byzantine fault-tolerant replication protocol was Castro and Liskov's BFT [4], which is described in Section 3. Yin et al. [20] propose separating the agreement component that orders requests from the execution component that processes requests, which allows utilization of the same agreement component for many different replication tasks and reduces the number of execution replicas to $2f + 1$. Martin and Alvisi [21] recently introduced a two-round Byzantine consensus algorithm, which uses $5f + 1$ servers in order to overcome f faults. This approach trades lower availability ($4f + 1$ out of $5f + 1$ connected servers are required, instead of $2f + 1$ out of $3f + 1$ as in BFT), for increased performance. The solution is appealing for local area networks with high connectivity. While we considered using it within the sites in our architecture, we feel the increased hardware cost outweighs the benefit of using one less intra-site round. The ShowByz system of Rodrigues et al. [22] seeks to support a large-scale deployment consisting of multiple replicated objects. ShowByz modifies BFT quorums to tolerate a larger fraction of faulty replicas, reducing the likelihood of any group being compromised at the expense of protocol liveness. Zyzzyva [23] uses speculative execution to reduce the cost of Byzantine fault-tolerant replication when there are no faulty replicas. Since Zyzzyva employs fewer wide area protocol rounds and has lower message complexity than BFT, we expect it to perform better than BFT when deployed on a wide area network. However, since Zyzzyva is a flat protocol, the leader sends more messages than the leader site representative in Steward.

Quorum Systems with Byzantine Fault Tolerance: Quorum systems obtain Byzantine fault tolerance by applying quorum replication methods. Examples of such systems include Phalanx [24], [25] and Fleet [26], [27]. Fleet provides a distributed repository for Java objects. It relies on an object replication mechanism that tolerates Byzantine failures of servers, while supporting benign clients. Although the approach is relatively scalable with the number of servers, it suffers from the drawbacks of flat Byzantine replication solutions. The Q/U protocol of Abd-El-Malek et al. [28] uses quorum replication techniques to achieve state machine replication, requiring $5f + 1$ servers to tolerate f faults. It can perform well when write contention is low, but suffers decreased throughput when concurrent updates are attempted on the same object.

Alternate Architectures: An alternate hierarchical approach to scale Byzantine replication to wide area networks can be based on having a few trusted nodes that are assumed to be working under a weaker adversary model. For example, these trusted nodes may exhibit crashes and recoveries but not penetrations. A Byzantine replication algorithm in such an environment can use this knowledge in order to optimize performance. Correia et al. [29] and Veríssimo [30] propose such a hybrid approach, where synchronous, trusted nodes provide strong global timing guarantees. Both the hybrid approach and the approach proposed in this paper can scale Byzantine replication to wide area networks. The hybrid approach makes stronger assumptions, while our approach pays more hardware and computational costs.

3 Background

Our work requires concepts from fault tolerance, Byzantine fault tolerance, and threshold cryptography. To facilitate the presentation of our protocol, Steward, we first provide an overview of three representative works in these areas: Paxos, BFT, and RSA threshold signatures.

Paxos: Paxos [18], [19] is a well-known fault-tolerant protocol that allows a set of distributed servers, exchanging messages via asynchronous communication, to totally order client requests in the benign-fault, crash-recovery model. Paxos uses an elected *leader* to coordinate the agreement protocol. If the leader crashes or becomes unreachable, the other servers elect a new leader; a *view change* occurs, allowing progress to (safely) resume in the new view under the reign of the new leader. Paxos requires at least $2f + 1$ servers to tolerate f faulty servers. Since servers are not Byzantine, only a single reply needs to be delivered to the client.

In the common case, in which a single leader exists and can communicate with a majority of servers, Paxos uses two asynchronous communication rounds to globally order client updates. In the first round, the leader assigns a sequence number to a client update and sends a *Proposal* message containing this assignment to the rest of the servers. In the second round, any server receiving the Proposal sends an *Accept* message, acknowledging the Proposal, to the rest of the servers. When a server receives a majority of matching Accept messages – indicating that a majority of servers have accepted the Proposal – it *orders* the corresponding update.

BFT: The BFT [4] protocol addresses the problem of replication in the Byzantine model where a number of servers can exhibit arbitrary behavior. Similar to Paxos, BFT uses an elected leader to coordinate the protocol and proceeds through a series of views. BFT extends Paxos into the Byzantine environment by using an additional communication round in the common case to ensure consistency both in and across views and by constructing strong majorities in each round of the protocol. Specifically, BFT uses a flat architecture and requires acknowledgments from $2f + 1$ out of $3f + 1$ servers to mask the behavior of f Byzantine servers. A client must wait for $f + 1$ identical responses to be guaranteed that at least one correct server assented to the returned value.

In the common case, BFT uses three communication rounds. In the first round, the leader assigns a sequence number to a client update and proposes this assignment to the rest of the servers by broadcasting a *Pre-prepare* message. In the second round, a server accepts the proposed assignment by broadcasting an acknowledgment, *Prepare*. When a server collects a *Prepare Certificate* (i.e., it receives the Pre-Prepare and $2f$ Prepare messages with the same view number and sequence number as the Pre-prepare), it begins the third round by broadcasting a *Commit* message. A server *commits* the corresponding update when it receives $2f+1$ matching commit messages.

Threshold digital signatures: Threshold cryptography [31] distributes trust among a group of participants to protect information (e.g., threshold secret sharing [32]) or computation (e.g., threshold digital signatures [33]). A (k, n) threshold digital signature scheme allows a set of servers to generate a digital signature as a single logical entity despite $k-1$ Byzantine faults. It divides a private key into n shares, each owned by a server. Each server uses its key share to generate a partial signature on a message m and sends the partial signature to a *combiner* server, which combines the partial signatures into a threshold signature on m. The threshold signature, which is verified using the public key corresponding to the divided private key, is only valid if it is the result of combining k valid partial signatures on m.

Shoup [33] proposed a practical threshold digital signature scheme that allows participants to generate threshold signatures based on the standard RSA [34] digital signature. The scheme provides verifiable secret sharing [35], which allows participants to verify that the partial signatures contributed by other participants are valid (i.e., they were generated with a share from the initial key split).

4 System Model

Servers are implemented as deterministic state machines [36], [37]. All correct servers begin in the same initial state and transition between states by applying updates as they are ordered. The next state is completely determined by the current state and the next update to be applied.

We assume a Byzantine fault model. Servers are either *correct* or *faulty*. Correct servers do not crash. Faulty servers may behave arbitrarily. Communication is asynchronous. Messages can be delayed, lost, or duplicated. Messages that do arrive are not corrupted.

Servers are organized into wide area *sites*, each having a unique identifier. Each server belongs to one site and has a unique identifier within that site. The network may partition into multiple disjoint *components*, each containing one or more sites. During a partition, servers from sites in different components are unable to communicate with each other. Components may subsequently re-merge. Each site S_i has at least $3*(f_i)+1$ servers, where f_i is the maximum number of servers that may be faulty within S_i. For simplicity, we assume in what follows that in each site there are at most f faulty servers. Clients are distinguished by unique identifiers.

We employ digital signatures, and we make use of a cryptographic hash function to compute message digests. Client updates are properly authenticated and protected against modifications. We assume that all adversaries, including faulty servers, are computationally bounded such that they cannot subvert these cryptographic mechanisms. We also use a $(2f+1, 3f+1)$ threshold digital signature scheme. Each site has a public key, and each server receives a share with the corresponding proof that can be used to demonstrate the validity of the server's partial signatures. We assume that threshold signatures are unforgeable without knowing $2f+1$ or more shares.

5 Service Properties

Our protocol assigns global, monotonically increasing sequence numbers to updates, to establish a global, total order. Below we define the safety and liveness properties of the Steward protocol. We say that:

- *a client proposes* an update when the client sends the update to a server in the local site, and the server receives it.
- *a server executes* an update with sequence number i when it applies the update to its state machine. A server executes update i only after having executed all updates with a lower sequence number in the global total order.
- *two servers are connected* or *a client and server are connected* if any message that is sent between them will arrive in a bounded time. The protocol participants need not know this bound beforehand.
- *two sites are connected* if every correct server in one site is connected to every correct server in the other site.
- *a client is connected to a site* if it can communicate with all servers in that site.

We define the following two safety conditions:

Definition 5.1: S1 - Safety: If two correct servers execute the i^{th} update, then these updates are identical.

Definition 5.2: S2 - Validity: Only an update that was proposed by a client may be executed.

Read-only queries can be handled within a client's local site and provide one-copy serializability semantics [2]. Alternatively, a client can specify that its query should be linearizable [3], in which case replies are collected from a majority of wide area sites.

Since no asynchronous Byzantine replication protocol can always be both safe and live [38], we provide liveness under certain synchrony conditions. We introduce the following terminology to encapsulate these synchrony conditions and our progress metric:

1) A *site is stable* with respect to time T if there exists a set, S, of $2f+1$ servers within the site, where, for all times $T' > T$, the members of S are (i) correct

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

and (ii) connected. We call the members of S *stable servers*.

2) The *system is stable* with respect to time T if there exists a set, S, of a majority of sites, where, for all times $T' > T$, the sites in S are (i) stable with respect to T and (ii) connected. We call the sites in S the *stable sites*.
3) *Global progress* occurs when some stable server executes an update.

We now define our liveness property:

DEFINITION 5.3: L1 - GLOBAL LIVENESS: If the system is stable with respect to time T, then if, after time T, a stable server receives an update which it has not executed, then global progress eventually occurs.

6 PROTOCOL DESCRIPTION

Steward leverages a hierarchical architecture to scale Byzantine replication to the high-latency, low-bandwidth links characteristic of wide area networks. Instead of running a single, relatively costly Byzantine fault-tolerant protocol among all *servers* in the system, Steward runs a more lightweight benign fault-tolerant protocol among all *sites* in the system, which reduces the number of messages and communication rounds on the wide area network compared to a flat Byzantine solution.

Steward's hierarchical architecture results in two levels of protocols: global and local. The global, Paxos-like protocol is run among wide area sites. Since each site consists of a set of potentially malicious servers (instead of a single trusted participant, as Paxos assumes), Steward employs several intra-site Byzantine fault-tolerant protocols to mask the effects of malicious behavior at the local level. Servers within a site agree upon the contents of messages to be used by the global protocol and generate a threshold signature for each message, preventing a malicious server from misrepresenting the site's decision and confining malicious behavior to the local site. In this way, each site emulates the behavior of a correct Paxos participant in the global protocol.

Similar to the elected coordinator scheme used in BFT, the local protocols in Steward are run in the context of a *local view*, with one server, the *site representative*, serving as the coordinator of a given view. Besides coordinating the local agreement and threshold-signing protocols, the representative (1) disseminates messages in the global protocol originating from the local site to the other site representatives and (2) receives global messages and distributes them to the local servers. If the representative is suspected to be faulty, the other servers in the site run a local view change protocol to replace the representative and install a new view.

While Paxos uses an elected leader server to coordinate the protocol, Steward uses an elected *leader site* to coordinate the global protocol; the global protocol runs in the context of a *global view*, with one leader site in charge of each view. If the leader site is partitioned away, the non-leader sites run a global view change protocol to elect a new one and install a new global view. The representative of the leader site drives the global protocol by invoking the local protocols needed to construct the messages sent over the wide area network.

In the remainder of this section, we present the local and global protocols that Steward uses to totally order client updates. We first describe the data structures used by our protocols. We then present the common case operation of Steward, followed by the view change protocols, which are run when failures occur. We then present the timeout mechanisms that Steward uses to ensure liveness. Due to space limitations, we include pseudocode associated with normal-case operation only. Complete pseudocode can be found in Appendix A.

6.1 Data Structures

Each server maintains separate variables for the global, Paxos-like protocol and the local, intra-site, Byzantine fault-tolerant protocols. Within the global context, a server maintains the state of its current global view and a *Global_History*, reflecting the status of those updates it has globally ordered or is attempting to globally order. Within the local context, a server maintains the state of its current local view. In addition, each server at the leader site maintains a *Local_History*, reflecting the status of those updates for which it has constructed, or is attempting to construct, a Proposal. Upon receiving a message, a server first runs a validity check on the message to ensure that it contains a valid RSA signature and does not originate from a server known to be faulty. If a message is valid, it can be applied to the server's data structures provided it does not conflict with any data contained therein.

6.2 The Common Case

In this section, we trace the flow of an update through the system as it is globally ordered during common case operation (i.e., when no leader site or site representative election occurs). The common case makes use of two local, intra-site protocols: THRESHOLD-SIGN (Fig. 1) and ASSIGN-SEQUENCE (Fig. 2), which we describe below. Pseudocode for the global ordering protocol (ASSIGN-GLOBAL-ORDER) is listed in Fig. 3.

The common case works as follows:

1) A client sends an update to a server in its local site. The update is uniquely identified by a pair consisting of the client's identifier and a client-generated logical timestamp. A correct client proposes an update with timestamp $i + 1$ only after it receives a reply for an update with timestamp i. The client's local server forwards the update to the local representative, which forwards the update to the representative of the leader site. If the client does not receive a reply within its timeout period, it broadcasts the update to all servers in its site.

```
THRESHOLD-SIGN(Data_s data, int server_id):
A1. Partial_Sig ← GENERATE_PARTIAL_SIG(data, server_id)
A2. Local Broadcast: Partial_Sig

B1. Upon receiving a set, PSig_Set, of 2f+1 Partial_Sigs:
B2.   signature ← COMBINE(PSig_Set)
B3.   if VERIFY(signature)
B4.      return signature
B5.   else
B6.      for each S in PSig_Set
B7.         if NOT VERIFY(S)
B8.            REMOVE(S, PSig_Set)
B9.            ADD(S.server_id, Corrupted_Servers_List)
B9.            Corrupted_Server ← CORRUPTED(S)
B10.           Local Broadcast: Corrupted_Server
B11.           Wait for more Partial_Sig messages
```

Fig. 1: THRESHOLD-SIGN Protocol, used to generate a threshold signature on a message. The message can then be used in a global protocol.

```
ASSIGN-SEQUENCE(Update u):
A1. Upon invoking:
A2.   Local Broadcast: Pre-Prepare(gv, lv, Global_seq, u)
A3.   Global_seq++

B1. Upon receiving Pre-Prepare(gv, lv, seq, u):
B2.   Apply Pre-Prepare to Local_History
B3.   Local Broadcast: Prepare(gv, lv, seq, Digest(u))

C1. Upon receiving Prepare(gv, lv, seq, digest):
C2.   Apply Prepare to Local_History
C3.   if Prepare_Certificate_Ready(seq)
C4.      pc ← Local_History[seq].Prepare_Certificate
C5.      pre-prepare ← pc.Pre-Prepare
C6.      unsigned_prop ← ConstructProposal(pre-prepare)
C7.      invoke THRESHOLD-SIGN(unsigned_prop, Server_id)

D1. Upon THRESHOLD-SIGN returning signed_proposal:
D2.   Apply signed_proposal to Global_History
D3.   Apply signed_proposal to Local_History
D4.   return signed_proposal
```

Fig. 2: ASSIGN-SEQUENCE Protocol, used to bind an update to a sequence number and create a threshold-signed Proposal.

2) When the representative of the leader site receives an update, it invokes the ASSIGN-SEQUENCE protocol to assign a global sequence number to the update; this assignment is encapsulated in a *Proposal* message. The site then generates a threshold signature on the constructed Proposal using THRESHOLD-SIGN, and the representative sends the signed Proposal to the representatives of all other sites for global ordering.
3) When a representative receives a signed Proposal, it forwards this Proposal to the servers in its site. Upon receiving a Proposal, a server constructs a site acknowledgment (i.e., an *Accept* message) and invokes THRESHOLD-SIGN on this message. The representative combines the partial signatures and then sends the resulting threshold-signed Accept message to the representatives of the other sites.
4) The representative of a site forwards the incoming Accept messages to the local servers. A server globally orders the update when it receives $\lfloor S/2 \rfloor$ Accept messages from distinct sites (where S is the number of sites) and the corresponding Proposal. The server at the client's local site that originally received the update sends a reply back to the client.

We now highlight the details of the THRESHOLD-SIGN and ASSIGN-SEQUENCE protocols.

Threshold-Sign: The THRESHOLD-SIGN intra-site protocol (Fig. 1) generates a $(2f+1, 3f+1)$ threshold signature on a given message.[1] Upon invoking the protocol, a server generates a Partial_Sig message, containing a partial signature on the message to be signed and a verification proof that other servers can use to confirm that the partial signature was created using a valid share. The Partial_Sig message is broadcast within the site. Upon receiving 2f+1 partial signatures on a message, a server combines the partial signatures into a threshold signature on that message, which is then verified using the site's public key. If the signature verification fails, one or more partial signatures used in the combination were invalid, in which case the verification proofs provided with the partial signatures are used to identify incorrect shares, and the servers that sent these incorrect shares are classified as malicious. Further messages from the corrupted servers are ignored, and the proof of corruption (the invalid Partial_Sig message) is broadcast to the other servers in the site.

```
ASSIGN-GLOBAL-ORDER():
A1. Upon receiving or executing an update, or becoming
    globally or locally constrained:
A2.   if representative of leader site
A3.      if (globally_constrained and locally_constrained
          and In_Window(Global_seq))
A4.         u ← Get_Next_To_Propose()
A5.         if (u ≠ NULL)
A6.            invoke ASSIGN-SEQUENCE(u)

B1. Upon ASSIGN-SEQUENCE returning Proposal:
B2.   SEND to all sites: Proposal

C1. Upon receiving Proposal(site_id, gv, lv, seq, u):
C2.   Apply Proposal to Global_History
C3.   if representative
C4.      Local Broadcast: Proposal
C5.   unsigned_accept ← ConstructAccept(Proposal)
C6.   invoke THRESHOLD-SIGN(unsigned_accept, Server_id)

D1. Upon THRESHOLD-SIGN returning signed_accept:
D2.   Apply signed_accept to Global_History
D3.   if representative
D4.      SEND to all sites: signed_accept

E1. Upon receiving Accept(site_id, gv, lv, seq, Digest(u)):
E2.   Apply Accept to Global_History
E3.   if representative
E4.      Local Broadcast: Accept
E5.   if Globally_Ordered_Ready(seq)
E6.      global_ord_update ← ConstructOrderedUpdate(seq)
E7.      Apply global_ord_update to Global_History
```

Fig. 3: ASSIGN-GLOBAL-ORDER Protocol. The protocol runs among all sites and is similar to Paxos.

Assign-Sequence: The ASSIGN-SEQUENCE local protocol (Fig. 2) is used in the leader site to construct a Proposal message. The protocol takes as input an update that was returned by the Get_Next_To_Propose procedure, which is invoked by the representative of the leader site during ASSIGN-GLOBAL-ORDER (Fig. 3, line A4). Get_Next_To_Propose considers the next sequence number for which an update should be ordered and returns either (1) an update that has already been bound to that sequence number, or (2) an update that is not bound to any sequence number. This ensures that the constructed Proposal cannot be used to violate safety and, if globally ordered, will result in global progress.

[1] We could use an $(f+1, 3f+1)$ threshold signature at the cost of an additional round in ASSIGN-SEQUENCE.

ASSIGN-SEQUENCE consists of three rounds. The first two are similar to the corresponding rounds of BFT, and the third round consists of an invocation of THRESHOLD-SIGN. During the first round, the representative binds an update, u, to a sequence number, seq, by creating and sending a Pre-Prepare(gv, lv, seq, u) message, where gv and lv are the current global and local views, respectively. A Pre-Prepare causes a conflict if either a binding (seq, u') or (seq', u) exists in a server's data structures. When a non-representative receives a Pre-Prepare that does not cause a conflict, it broadcasts a matching Prepare(gv, lv, seq, Digest(u)) message. At the end of the second round, when a server receives a Pre-Prepare and $2f$ matching Prepare messages for the same views, sequence number, and update (i.e., when it collects a *Prepare_Certificate*), it invokes THRESHOLD-SIGN on a Proposal($site\_id$, gv, lv, seq, u). If there are $2f+1$ correct, connected servers in the site, THRESHOLD-SIGN returns a threshold-signed Proposal to all servers.

6.3 View Changes

Several types of failure may occur during system execution, such as the corruption of a site representative or the partitioning away of the leader site. Such failures require delicate handling to preserve safety and liveness.

To ensure that the system can make progress despite server or network failures, Steward uses timeout-triggered *leader election* protocols at both the local and global levels of the hierarchy to select new protocol coordinators. Each server maintains two timers, Local_T and Global_T, which expire if the server does not execute a new update (i.e., make global progress) within the local or global timeout period. When the Local_T timers of $2f + 1$ servers within a site expire, the servers replace the current representative. Similarly, when, in a majority of sites, the Global_T timers of $2f+1$ local servers expire, the sites replace the current leader site.

While the leader election protocols guarantee progress if sufficient synchrony and connectivity exist, Steward uses *view change* protocols at both levels of the hierarchy to ensure *safe* progress. The presence of benign or malicious failures introduces a window of uncertainty with respect to pending decisions that may (or may not) have been made in previous views. For example, the new coordinator may not be able to definitively determine if some server globally ordered an update for a given sequence number. However, our view change protocols guarantee that *if* any server globally ordered an update for that sequence number in a previous view, the new coordinator will collect sufficient information to ensure that it respects the established binding in the new view.

Steward uses a *constraining* mechanism to enforce this behavior. Before participating in the global ordering protocol, a correct server must become both *locally constrained* and *globally constrained* by completing the LOCAL-VIEW-CHANGE and GLOBAL-VIEW-CHANGE protocols. The local constraints ensure continuity across local views (when the site representative changes), and the global constraints ensure continuity across global views (when the leader site changes). Since a faulty leader site representative may ignore the constraints imposed by previous views, *all* servers in the leader site become constrained, preventing a faulty server from causing them to act in an inconsistent way.

We now provide relevant details of our leader election and view change protocols. We focus primarily on the function of each protocol in ensuring safety and liveness, rather than on the inner-workings of each protocol.

Leader Election: Steward uses two Byzantine fault-tolerant leader election protocols, one in each level of the hierarchy. Each site runs the LOCAL-VIEW-CHANGE protocol to elect its representative, and the system runs the GLOBAL-LEADER-ELECTION protocol to elect the leader site. Both protocols provide two important properties necessary for liveness: If the system is stable and does not make global progress, (1) views are incremented consecutively, and (2) stable servers remain in each view for approximately one timeout period. These properties allow stable protocol coordinators to remain in power long enough for global progress to be made.

Local View Changes: Since the sequencing of Proposals occurs at the leader site (using the ASSIGN-SEQUENCE local protocol), replacing the representative of the leader site requires a Byzantine fault-tolerant reconciliation protocol to preserve the consistency of the sequencing. Steward uses the CONSTRUCT-LOCAL-CONSTRAINT local protocol for this purpose. As a result of the protocol, servers have enough information about pending Proposals to preserve safety in the new local view. Specifically, it prevents two conflicting Proposals, P1(gv, lv, seq, u) and P2(gv, lv, seq, u'), with $u \neq u'$, from being constructed in the same global view.

Global View Changes: The GLOBAL-VIEW-CHANGE protocol is triggered after a leader site election. It makes use of two local protocols, CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT, used at the leader site and non-leader sites, respectively. The leader site representative invokes CONSTRUCT-ARU, which generates an *Aru_Message*, containing the sequence number up to which at least $f + 1$ correct servers in the leader site have globally ordered all previous updates. The representative sends the Aru_Message to all other site representatives. Upon receiving this message, a non-leader site representative invokes CONSTRUCT-GLOBAL-CONSTRAINT, which generates a *Global_Constraint_Message* reflecting the state of the site's knowledge above the sequence number contained in the Aru_Message. Servers in the leader site use the Global_Constraint messages from a majority of sites to become *globally constrained*, which restricts the Proposals they will generate in the new view to preserve safety.

6.4 Timeouts

Steward uses timeouts to detect failures. Our protocols do not assume synchronized clocks; however, we do assume that the drift of the clocks at different servers is small. This assumption is valid considering today's technology. If a server does not execute updates, a local and, eventually, a global timeout will occur. These timeouts cause the server to "assume" that the current local and/or global coordinator has failed. Accordingly, the server attempts to elect a new local/global coordinator by suggesting new views. Intuitively, coordinators are elected for a *reign*, during which each server expects to make progress. If a server does not make progress, its Local_T timer expires, and it attempts to elect a new representative. Similarly, if a server's Global_T timer expires, it attempts to elect a new leader site. To provide liveness, Steward changes coordinators using three timeout values, which cause the coordinators of the global and local protocols to be elected at different rates. This guarantees that, during each global view, correct representatives at the leader site can communicate with correct representatives at all stable non-leader sites. We now describe the three timeouts.

Non-Leader Site Local Timeout (T1): Local_T is set to this timeout at servers in non-leader sites. When Local_T expires at all stable servers in a site, they preinstall a new local view. T1 must be long enough for servers in the non-leader site to construct Global_Constraint messages, which requires at least enough time to complete THRESHOLD-SIGN.

Leader Site Local Timeout (T2): Local_T is set to this timeout at servers in the leader site. T2 must be long enough to allow the representative to communicate with all stable sites. Observe that all non-leader sites do not need to have correct representatives at the same time; Steward makes progress as long as each leader site representative can communicate with at least one correct server at each stable non-leader site. We accomplish this by choosing T1 and T2 so that, during the reign of a representative at the leader site, $f + 1$ servers reign for complete terms at each non-leader site. The reader can think of the relationship between the timeouts as follows: The time during which a server is representative at the leader site *overlaps* with the time that $f+1$ servers are representatives at the non-leader sites. Therefore, we require that $T2 \geq (f+2)*T1$. The factor $f+2$ accounts for the possibility that Local_T is already running at some of the non-leader-site servers when the leader site elects a new representative.

Global Timeout (T3): Global_T is set to this timeout at all servers, regardless of whether they are in the leader site. At least two correct representatives in the leader site must serve complete terms during each global view. Thus, we require that $T3 \geq (f+3)*T2$. From the relationship between T1 and T2, each of these representatives will be able to communicate with a correct representative at each stable site. If the timeouts are sufficiently long and the system is stable, the first correct server to serve a full reign as leader site representative will complete GLOBAL-VIEW-CHANGE. The second correct server will be able to globally order and execute a new update.

We compute our timeout values based on the global view. If the system is stable, all stable servers will move to the same global view. Timeouts T1, T2, and T3 are deterministic functions of the global view, guaranteeing that the relationships described above are met at *every* stable server. Timeouts double every S global views, where S is the number of sites. Thus, if there is a time after which message delays do not increase, then our timeouts eventually grow long enough for global progress to be made. We note that, when failures occur, Steward may require more time than flat Byzantine fault-tolerant replication protocols to reach a configuration where progress will occur. The global timeout must be large enough so that a correct leader site representative will complete GLOBAL-VIEW-CHANGE, which may require waiting for several local view changes to complete. In contrast, flat protocols do not incur this delay. However, Steward's hierarchical architecture yields an $O(S)$ wide area message complexity for view change messages, compared to $O(N)$ for flat architectures.

7 Performance Evaluation

To evaluate the performance of our hierarchical architecture, we implemented a complete prototype of our protocol including all necessary communication and cryptographic functionality. We focus only on the networking and cryptographic aspects of our protocols and do not consider disk writes.

Testbed and Network Setup: We selected a network topology consisting of 5 wide area sites and assumed at most 5 Byzantine faults in each site, in order to quantify the performance of our system in a realistic scenario. This requires 16 replicated servers in each site.

Our experimental testbed consists of a cluster with twenty 3.2 GHz, 64-bit Intel Xeon computers. Each computer can compute a 1024-bit RSA signature in 1.3 ms and verify it in 0.07 ms. For n=16, k=11, 1024-bit threshold cryptography which we use for these experiments, a computer can compute a partial signature and verification proof in 3.9 ms and combine the partial signatures in 5.6 ms. The leader site was deployed on 16 machines, and the other 4 sites were emulated by one computer each. An emulating computer performed the role of a representative of a complete 16 server site. Thus, our testbed was equivalent to an 80 node system distributed across 5 sites. Upon receiving a message, the emulating computers busy-waited for the time it took a 16 server site to handle that packet and reply to it, including intra-site communication and computation. We determined busy-wait times for each type of packet by benchmarking individual protocols on a fully deployed, 16 server site. We used the Spines [39], [40] messaging system to emulate latency and throughput constraints on the wide area links.

We compared the performance results of the above system with those of the Castro-Liskov implementation of BFT [4] on the same network setup with five sites,

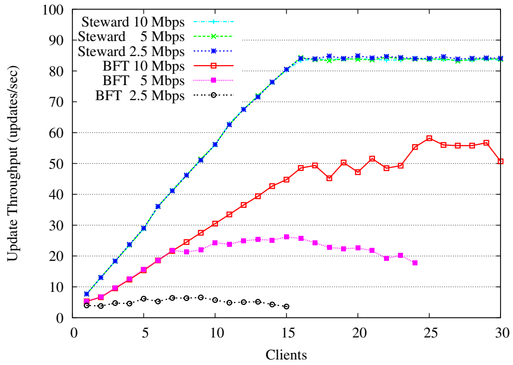

Fig. 4: Write Update Throughput

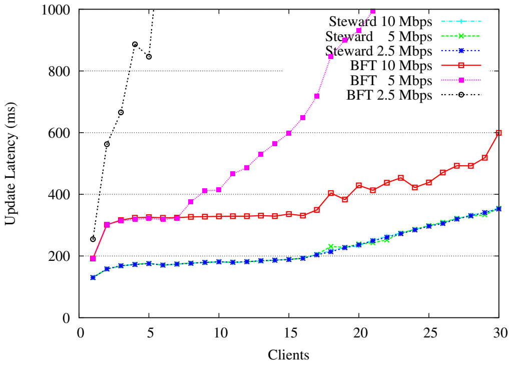

Fig. 5: Write Update Latency

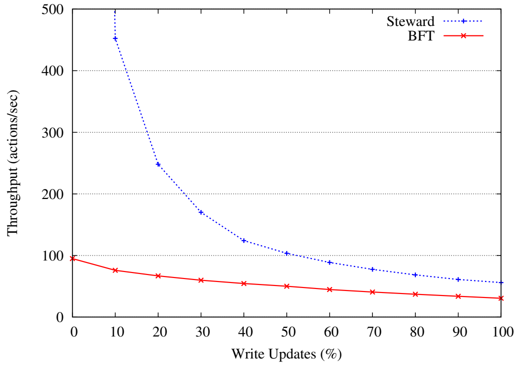

Fig. 6: Update Mix Throughput - 10 Clients

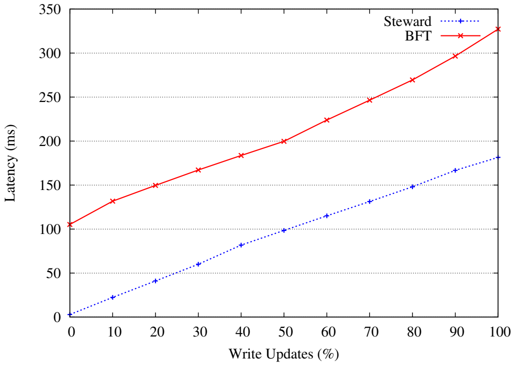

Fig. 7: Update Mix Latency - 10 Clients

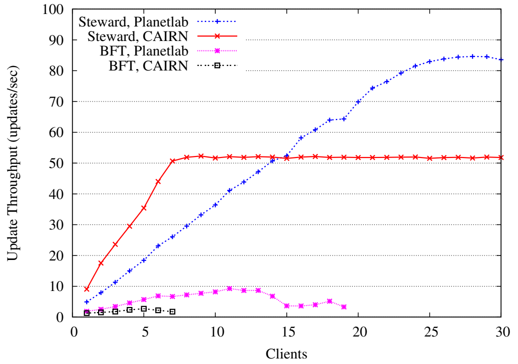

Fig. 8: WAN Emulation - Write Update Throughput

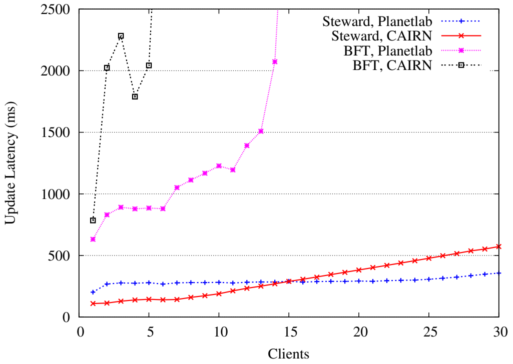

Fig. 9: WAN Emulation - Write Update Latency

run on the same cluster. Instead of using 16 servers in each site, for BFT we used a *total* of 16 servers across the entire network. This allows for up to 5 Byzantine failures in the entire network for BFT, instead of up to 5 Byzantine failures in each site for Steward. Since BFT is a flat solution where there is no correlation between faults and the sites in which they can occur, we believe this comparison is fair. We distributed the BFT servers such that four sites contain 3 servers each, and one site contains 4 servers. All the write updates and read-only queries in our experiments carried a payload of 200 bytes, representing a common SQL statement.

Our protocols use RSA signatures for authentication. Although our ASSIGN-SEQUENCE protocol can use vectors of MACs for authentication (as BFT can), the benefit of using MACs compared to signatures is limited because the latency for global ordering is dominated by the wide area network latency. In addition, digital signatures provide non-repudiation, which can be used to detect malicious servers.

In order to support our claim that our results reflect fundamental differences between the Steward and BFT protocols, and not differences in their implementations, we confirmed that BFT's performance matched our similar intra-site agreement protocol, ASSIGN-SEQUENCE. Since BFT performed slightly better than ASSIGN-SEQUENCE, we attribute Steward's performance advantage over BFT to its hierarchical architecture and resultant wide area message savings. Note that in our five-site test configuration, BFT sends over twenty times more wide area messages per update than Steward. This message savings is consistent with the difference in performance between Steward and BFT shown in the experiments that follow.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Bandwidth Limitation: We first investigate the benefits of the hierarchical architecture in a symmetric configuration with 5 sites, where all sites are connected to each other with 50 ms latency links (emulating crossing the continental US).

In the first experiment, clients inject write updates. Fig. 4 shows how limiting the capacity of wide area links affects update throughput. As we increase the number of clients, BFT's throughput increases at a lower slope than Steward's, mainly due to the additional wide area crossing for each update. Steward can process up to 84 updates/sec in all bandwidth cases, at which point it is limited by CPU used to compute threshold signatures. At 10, 5, and 2.5 Mbps, BFT achieves about 58, 26, and 6 updates/sec, respectively. In each of these cases, BFT's throughput is bandwidth limited. We also notice a reduction in the throughput of BFT as the number of clients increases. We attribute this to a cascading increase in message loss, caused by the lack of flow control in BFT. For the same reason, we were not able to run BFT with more than 24 clients at 5 Mbps, and 15 clients at 2.5 Mbps. We believe that adding a client queuing mechanism would stabilize the performance of BFT to its maximum achieved throughput.

Fig. 5 shows that Steward's average update latency slightly increases with the addition of clients, reaching 190 ms at 15 clients in all bandwidth cases. As client updates start to be queued, latency increases linearly. BFT exhibits a similar trend at 10 Mbps, where the average update latency is 336 ms at 15 clients. As the bandwidth decreases, the update latency increases heavily, reaching 600 ms at 5 Mbps and 5 seconds at 2.5 Mbps, at 15 clients.

Increasing the update size would increase the percentage of wide area bandwidth used to carry data in both Steward and BFT. Since BFT has higher protocol overhead per update, this would benefit BFT to a larger extent. However, Steward's hierarchical architecture would still result in a higher data throughput, because the update must only be sent on the wide area $O(S)$ times, whereas BFT would need to send it $O(N)$ times. A similar benefit can be achieved by using batching techniques, which reduces the protocol overhead per update. We demonstrate the impact of batching in [8].

Adding Read-only Queries: Our hierarchical architecture enables read-only queries to be answered locally. To demonstrate this benefit, we conducted an experiment where 10 clients send random mixes of read-only queries and write updates. We compared the performance of Steward (which provides one-copy serializability) and BFT (which provides linearizability) with 50 ms, 10 Mbps links, where neither was bandwidth limited. Fig. 6 and Fig. 7 show the average throughput and latency, respectively, of different mixes of queries and updates. When clients send only queries, Steward achieves about 2.9 ms per query, with a throughput of over 3,400 queries/sec. Since queries are answered locally, their latency is dominated by two RSA signatures, one at the originating client and one at the servers answering the query. Depending on the mix ratio, Steward performs 2 to 30 times better than BFT.

BFT's read-only query latency is about 105 ms, and its throughput is 95 queries/sec. This is expected, as read-only queries in BFT need to be answered by at least $f + 1$ servers, some of which are located across wide area links. BFT requires at least $2f + 1$ servers in each site to guarantee that it can answer queries locally. Such a deployment, for 5 faults and 5 sites, would require at least 55 servers, which would dramatically increase communication for updates and reduce BFT's performance.

Wide Area Scalability: To demonstrate Steward's scalability on real networks, we conducted experiments measuring its performance on two emulated networks based on real wide area topologies. The first experiment was run on an emulated Planetlab [41] topology consisting of five sites spanning several continents, and the second experiment emulated a WAN setup across the US, called CAIRN [42]. Fig. 8 and Fig. 9 show the average write update throughput and latency measured in both experiments, which we now describe.

We first selected five sites on the Planetlab network, measured the latency and available bandwidth between all sites, and emulated the network topology on our cluster. We ran the experiment on our cluster because Planetlab machines lack sufficient computational power. The five sites were located in the US (University of Washington), Brazil (Rio Grande do Sul), Sweden (Swedish Institute of Computer Science), Korea (KAIST) and Australia (Monash University). The network latency varied between 59 ms (US - Korea) and 289 ms (Brazil - Korea). Available bandwidth varied between 405 Kbps(Brazil - Korea) and 1.3 Mbps (US - Australia).

As seen in Fig. 8, Steward is able to achieve its maximum throughput of 84 updates/sec with 27 clients. Fig. 9 shows that the latency increases from about 200 ms for one client to about 360 ms for 30 clients. BFT is bandwidth limited to about 9 updates/sec. The update latency is 631 ms for one client and several seconds with more than 6 clients.

In the next experiment, we emulated the CAIRN topology using the Spines messaging system, and we ran Steward and BFT on top of it. The main characteristic of CAIRN is that East and West Coast sites were connected through a single 38 ms, 1.86 Mbps link. Since Steward runs a lightweight fault-tolerant protocol between wide area sites, we expect it to achieve performance comparable to existing benign fault-tolerant replication protocols. We compare the performance of our hierarchical Byzantine architecture on the CAIRN topology with that of two-phase commit protocols [16] on the same topology.

Fig. 8 shows that Steward achieved a throughput of about 51 updates/sec in our tests, limited mainly by the bandwidth of the link between the East and West Coasts in CAIRN. In comparison, an upper bound of two-phase commit protocols presented in [43] was able to achieve 76 updates/sec. We believe that the difference in performance is caused by the presence of additional digital signatures in the message headers of Steward, adding 128 bytes to the 200 byte payload of each update. Fig. 8 and Fig. 9 show that BFT achieved a maximum throughput of 2.7 updates/sec and an update latency of over a second, except when there was a single client.

8 Proofs of Correctness

In this section we first prove that Steward meets the safety property listed in Section 5. Due to space limitations, we provide a proof roadmap for liveness, and we state certain lemmas without proof. Complete proofs are presented in Appendix B.

8.1 Proof of Safety

We prove Safety by showing that two servers cannot globally order conflicting updates for the same sequence number. We use two main claims. In the first claim, we show that any two servers which globally order an update in the same global view for the same sequence number will globally order the same update. We show that a leader site cannot construct conflicting Proposal messages in the same global view. A conflicting Proposal has the same sequence number as another Proposal, but it has a *different* update. Since globally ordering two different updates for the same sequence number in the same global view would require two different Proposals from the same global view, and since only one Proposal can be constructed within a global view, all servers that globally order an update for a given sequence number in the same global view must order the same update.

In the second claim, we show that any two servers which globally order an update in different global views for the same sequence number must order the same update. We show that a leader site from a later global view cannot construct a Proposal conflicting with one used by a server in an earlier global view to globally order an update for that sequence number. Since no Proposals can be created that conflict with the one that has been globally ordered, no correct server can globally order a different update with the same sequence number. Since a server only executes an update once it has globally ordered an update for all previous sequence numbers, two servers executing the i^{th} update must execute the same update.

We now proceed to prove the first main claim:

Claim *8.1:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Then if any other server globally orders an update for sequence number seq in global view gv, it will globally order u.

To prove this claim, we use the following lemma, which shows that conflicting Proposal messages cannot be constructed in the same global view:

Lemma *8.1:* Let P1(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S for sequence number seq. Then no other Proposal message P2(gv, lv', seq, u') for $lv' \geq lv$, with $u' \neq u$, can be constructed.

We prove Lemma 8.1 with a series of lemmas. We begin by proving that two servers cannot collect conflicting Prepare Certificates or construct conflicting Proposals in the same global and local view.

Lemma *8.2:* Let PC1(gv, lv, seq, u) be a Prepare Certificate collected by some server in leader site S. Then no server in S can collect a different Prepare Certificate, PC2(gv, lv, seq, u'), with ($u \neq u'$). Moreover, if some server in S collects a Proposal P1(gv, lv, seq, u), then no server in S can construct a Proposal P2(gv, lv, seq, u'), with ($u \neq u'$).

Proof: We assume that both Prepare Certificates were collected and show that this leads to a contradiction. PC1 contains a Pre-Prepare(gv, lv, seq, u) and $2f$ Prepare(gv, lv, seq, Digest(u)) messages from distinct servers. Since there are at most f faulty servers in S, at least $f + 1$ of the messages in PC1 were from correct servers. PC2 contains similar messages, but with u' instead of u. Since any two sets of $2f+1$ messages intersect on at least one correct server, there exists a correct server that contributed to both PC1 and PC2. Assume, without loss of generality, that this server contributed to PC1 first (either by sending the Pre-Prepare message or by responding to it). If this server was the representative, it would not have sent the second Pre-Prepare message, because, from Figure 2 line A3, it increments Global_seq and does not return to seq in this local view. If this server was a non-representative, it would not have contributed a Prepare in response to the second Pre-Prepare, since this would have generated a conflict. Thus, this server did not contribute to PC2, a contradiction.

To construct Proposal P2, at least $f + 1$ correct servers would have had to send partial signatures on P2, after obtaining a Prepare Certificate PC2 reflecting the binding of seq to u' (Figure 2, line C7). Since some server collected PC1, no server can have collected such a Prepare Certificate, implying that P2 could not have been constructed. □

We now show that two conflicting Proposal messages cannot be constructed in the same global view, even across local view changes. We maintain the following invariant:

INVARIANT 8.1: Let P(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S for sequence number seq in global view gv. We say that Invariant 8.1 holds with respect to P if the following conditions hold in leader site S in global view gv:

1) There exists a set of at least $f + 1$ correct servers with a Prepare Certificate PC(gv, lv', seq, u) or a Proposal(gv, lv', seq, u), for $lv' \geq lv$, in their Local_History[seq] data structure, or a Globally_Ordered_Update(gv', seq, u), for $gv' \geq gv$, in their Global_History[seq] data structure.
2) There does not exist a server with any conflicting Prepare Certificate or Proposal from any view

(gv, lv'), with $lv' \geq lv$, or a conflicting Globally_Ordered_Update from any global view $gv' \geq gv$.

Lemma 8.3 shows that the invariant holds in the first global and local view in which any Proposal might have been constructed for a given sequence number. Lemma 8.4 shows that the invariant holds throughout the remainder of the global view, across local view changes. Finally, Lemma 8.5 shows that if the invariant holds, no Proposal message conflicting with the first Proposal that was constructed can be created. In other words, once a Proposal has been constructed for sequence number seq, there will always exist a set of at least $f+1$ correct servers which maintain and enforce the binding reflected in the Proposal.

Lemma *8.3:* Let P(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S for sequence number seq in global view gv. Then when P is constructed, Invariant 8.1 holds with respect to P, and it holds for the remainder of (gv, lv).

Lemma *8.4:* Let P(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S for sequence number seq in global view gv. If Invariant 8.1 holds with respect to P at the beginning of a run of CONSTRUCT-LOCAL-CONSTRAINT, then it is never violated during the run.

Lemma *8.5:* Let P(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S for sequence number seq in global view gv. If Invariant 8.1 holds with respect to P at the beginning of a view (gv, lv'), with $lv' \geq lv$, then it holds throughout the view.

We can now prove Lemma 8.1:

Proof: By Lemma 8.3, Invariant 8.1 holds with respect to P throughout (gv, lv). By Lemma 8.4, the invariant holds with respect to P during and after CONSTRUCT-LOCAL-CONSTRAINT. By Lemma 8.5, the invariant holds at the beginning and end of view (gv, $lv+1$). Repeated applications of Lemma 8.4 and Lemma 8.5 shows that the invariant always holds in global view gv.

In order for P2 to be constructed, at least $f+1$ correct servers must send a partial signature on P2 after collecting a corresponding Prepare Certificate (Figure 2, line C3). Since the invariant holds throughout gv, at least $f+1$ correct servers do not collect such a Prepare Certificate and do not send such a partial signature. This leaves only $2f$ servers remaining, which is insufficient to construct the Proposal. Since a Proposal is needed to construct a Globally_Ordered_Update, no conflicting Globally_Ordered_Update can be constructed. □

Finally, we can prove Claim 8.1:

Proof: To globally order an update u in global view gv for sequence number seq, a server needs a Proposal(gv, *, seq, u) message and $\lfloor S/2 \rfloor$ corresponding Accept messages. By Lemma 8.1, all Proposal messages constructed in gv are for the same update, which implies that all servers which globally order an update in gv for seq globally order the same update. □

We now prove the second main claim:

Claim *8.2:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Then if any other server globally orders an update for sequence number seq in a global view gv', with $gv' > gv$, it will globally order u.

We prove Claim 8.2 using Lemma 8.6, which shows that, once an update has been globally ordered for a given sequence number, no conflicting Proposal messages can be generated for that sequence number in any future global view.

Lemma *8.6:* Let u be the first update globally ordered by any server for sequence number seq with corresponding Proposal P1(gv, lv, seq, u). Then no other Proposal message P2(gv', *, seq, u') for $gv' > gv$, with $u' \neq u$, can be constructed.

We prove Lemma 8.6 using a series of lemmas, and we maintain the following invariant:

INVARIANT 8.2: Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Let P(gv, lv, seq, u) be the first Proposal message constructed by any server in the leader site in gv for sequence number seq. We say that Invariant 8.2 holds with respect to P if the following conditions hold:

1) There exists a majority of sites, each with at least $f+1$ correct servers with a Prepare Certificate(gv, lv', seq, u), a Proposal(gv', *, seq, u), or a Globally_Ordered_Update(gv', seq, u), with $gv' \geq gv$ and $lv' \geq lv$, in its Global_History[seq] data structure.
2) There does not exist, at any site in the system, a server with any conflicting Prepare Certificate(gv', lv', seq, u'), Proposal(gv', *, seq, u'), or Globally_Ordered_Update(gv', seq, u'), with $gv' \geq gv$, $lv' \geq lv$, and $u' \neq u$.

Lemma 8.7 shows that Invariant 8.2 holds when the first update is globally ordered for sequence number seq and that it holds throughout the view in which it is ordered. Lemmas 8.8 and 8.9 then show that the invariant holds across global view changes. Finally, Lemma 8.10 shows that if Invariant 8.2 holds at the beginning of a global view after which an update has been globally ordered, then it holds throughout the view.

Lemma *8.7:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Let P(gv, lv, seq, u) be the first Proposal message constructed by any server in the leader site in gv for sequence number seq. Then when u is globally ordered, Invariant 8.2 holds with respect to P, and it holds for the remainder of global view gv.

Lemma *8.8:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Let P(gv, lv, seq, u) be the first Proposal message constructed by any server in the leader site in gv for sequence number seq. Assume Invariant 8.2 holds with respect to P, and let S be one of the (majority) sites maintained by the first condition of the invariant. Then if a run of CONSTRUCT-ARU or CONSTRUCT-GLOBAL-CONSTRAINT begins at S, the invariant is never violated during the run.

Lemma *8.9:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which u was globally ordered. Let P(gv, lv, seq, u) be the first Proposal message constructed by any server in the leader site in gv for sequence number seq. Then if Invariant 8.2 holds with respect to P at the beginning of a run of the Global_View_Change protocol, then it is never violated during the run. Moreover, if at least $f + 1$ correct servers in the leader site become globally constrained by completing the GLOBAL-VIEW-CHANGE protocol, the leader site will be in the set maintained by Condition 1 of Invariant 8.2.

Lemma *8.10:* Let u be the first update globally ordered by any server for sequence number seq, and let gv be the global view in which gv was globally ordered. Let P(gv, lv, seq, u) be the first Proposal message constructed by any server in the leader site in gv for sequence number seq. Then if Invariant 8.2 holds with respect to P at the beginning of a global view (gv', *), with $gv' > gv$, then it holds throughout the view.

Proof: We show that no conflicting Prepare Certificate, Proposal, or Globally_Ordered_Update can be constructed during global view gv that would cause the invariant to be violated. We assume that a conflicting Prepare Certificate PC is collected and show that this leads to a contradiction. This then implies that no conflicting Proposals or Globally_Ordered_Updates can be constructed.

If PC is collected, then some server collected a Pre-Prepare(gv', lv, seq, u') and $2f$ Prepare(gv', lv, seq, Digest(u')) for some local view lv and $u' \neq u$. At least $f + 1$ of these messages were from correct servers. Moreover, this implies that at least $f + 1$ correct servers were globally constrained. By Lemma 8.9, since at least $f + 1$ correct servers became globally constrained in gv', the leader site meets Condition 1 of Invariant 8.2, and it thus has at least $f+1$ correct servers with a Prepare Certificate, Proposal, or Globally_Ordered_Update binding seq to u. At least one such server contributed to the construction of PC. A correct representative would not send such a Pre-Prepare message because the Get_Next_To_Propose() routine would return the constrained update. Similarly, a correct server would see a conflict. Since no server can collect a conflicting Prepare Certificate, no server can construct a conflicting Proposal. Thus, no server can collect a conflicting Globally_Ordered_Update, since this would require a conflicting Proposal, and Invariant 8.2 holds throughout global view gv'. □

We can now prove Lemma 8.6:

Proof: By Lemma 8.7, Invariant 8.2 holds with respect to P1 throughout global view gv. By Lemma 8.9, the invariant holds with respect to P1 during and after the GLOBAL-VIEW-CHANGE protocol. By Lemma 8.10, the invariant holds at the beginning and end of global view $gv + 1$. Repeated application of Lemma 8.9 and Lemma 8.10 shows that the invariant always holds for all global views $gv' > gv$.

In order for P2 to be constructed, at least $f + 1$ correct servers must send a partial signature on P2 after collecting a corresponding Prepare Certificate (Figure 2, line C3). Since the invariant holds, at least $f + 1$ correct servers do not collect such a Prepare Certificate and do not send such a partial signature. This leaves only $2f$ servers remaining, which is insufficient to construct the Proposal. □

Finally, we can prove Claim 8.2:

Proof: We assume that two servers globally order conflicting updates with the same sequence number in two global views gv and gv' and show that this leads to a contradiction.

Without loss of generality, assume that a server globally orders update u in gv, with $gv < gv'$. This server collected a a Proposal(gv, *, seq, u) message and $\lfloor S/2 \rfloor$ corresponding Accept messages. By Lemma 8.6, any future Proposal message for sequence number seq contains update u, including the Proposal from gv'. This implies that another server that globally orders an update in gv' for sequence number seq must do so using the Proposal containing u, which contradicts the fact that it globally ordered u' for sequence number seq. □

SAFETY - S1 follows directly from Claims 8.1 and 8.2.

8.2 Proof Roadmap of Global Liveness

We prove Global Liveness by contradiction: we assume that global progress does not occur and show that, if the system is stable and a stable server receives an update which it has not executed, then the system will reach a state in which some stable server *will* execute an update and make global progress.

We first show that, if no global progress occurs, all stable servers eventually reconcile their global histories to the maximum sequence number through which any stable server has executed all updates. By definition, if any stable server executes an update beyond this point, global progress will have been made, and we will have reached a contradiction.

Once the above reconciliation completes, the system eventually reaches a state in which a stable representative of a stable leader site remains in power for sufficiently long to be able to complete the global view change protocol; this is a precondition for globally ordering a new update (which would imply global progress). To prove this, we first show that, eventually, the stable sites will move through global views together, and within each stable site, the stable servers will move through local views together. We then establish the relationships between the global and local timeouts, which show that the stable servers will eventually remain in their views long enough for global progress to be made.

Finally, we show that a stable representative of a stable leader site will eventually be able to globally order (and execute) an update which it has not previously executed. We first show that the same update cannot be globally ordered on two different sequence numbers. This implies that when the representative executes an update, global progress will occur; no correct server has previously executed the update, since otherwise, by our reconciliation claim, all stable servers would have eventually executed the update and global progress would have occurred (which contradicts our assumption). We then show that each of the local protocols invoked during the global ordering protocol will complete in bounded finite time. Since the duration of our timeouts are a function of the global view, and stable servers preinstall consecutive global views, the stable servers will eventually reach a global view in which a new update can be globally ordered and executed.

9 Conclusion

This paper presented a hierarchical architecture that enables efficient scaling of Byzantine replication to systems that span multiple wide area sites, each consisting of several potentially malicious replicas. The architecture reduces the message complexity on wide area updates, increasing the system's scalability. By confining the effect of any malicious replica to its local site, the architecture enables the use of a benign fault-tolerant algorithm over the WAN, increasing system availability. Further increase in availability and performance is achieved by the ability to process read-only queries within a site.

We implemented Steward, a fully functional prototype that realizes our architecture, and evaluated its performance over several network topologies. The experimental results show considerable improvement over flat Byzantine replication algorithms, bringing the performance of Byzantine replication closer to existing benign fault-tolerant replication techniques over WANs.

Acknowledgment

Yair Amir thanks his friend Dan Schnackenberg for introducing him to this problem area and for conversations on this type of solution. He will be greatly missed.

This work was partially funded by DARPA grant FA8750-04-2-0232, and by NSF grants 0430271 and 0430276.

References

[1] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage, "Scaling byzantine fault-tolerant replication to wide area networks," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '06)*, 2006, pp. 105–114.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987.

[3] 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.

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance," *Proc. Symp. Operating Systems Design and Implementation (OSDI '99).* New Orleans, LA, USA: USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999, pp. 173–186.

[5] A. Avizeinis, "The n-version approach to fault-tolerant software," *IEEE Trans. Soft. Eng.*, vol. SE-11, no. 12, pp. 1491–1501, 1985.

[6] "Genesis: A framework for achieving component diversity, http://www.cs.virginia.edu/genesis/."

[7] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," *Proc. USENIX Security Symp. (USENIX-SS '06)*, 2006, pp. 105–120.

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Customizable fault tolerance for wide-area replication," *Proc. IEEE Int'l Symp. Reliable Distributed Systems (SRDS '07)*, Beijing, China, 2007, pp. 65–82.

[9] M. J. Fischer, "The consensus problem in unreliable distributed systems (a brief survey)," *Fundamentals of Computation Theory*, 1983, pp. 127–140.

[10] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal of Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[11] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *Computer Systems*, vol. 1, no. 3, pp. 222–238, 1983.

[12] "The Rampart Toolkit for building high-integrity services," in *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, 1995, pp. 99–110.

[13] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," *Proc. IEEE Hawaii Int'l Conf. System Sciences*, vol. 3, Kona, Hawaii, 1998, pp. 317–326.

[14] M. Cukier, T. Courtney, J. Lyons, H. V. Ramasamy, W. H. Sanders, M. Seri, M. Atighetchi, P. Rubel, C. Jones, F. Webber, P. Pal, R. Watro, and J. Gossett, "Providing intrusion tolerance with ITUA," *Supplement of IEEE Int'l Conf. Dependable Systems and Networks (DSN '02)*, 2002, pp. C-5-1–C-5-3.

[15] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the cost of providing intrusion tolerance in group communication systems," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '02)*, 2002, pp. 229–238.

[16] K. Eswaran, J. Gray, R. Lorie, and I. Taiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976.

[17] D. Skeen, "A quorum-based commit protocol," *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, 1982, pp. 69–80.

[18] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.

[19] Lamport, "Paxos made simple," *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, pp. 51–58, 2001.

[20] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault-tolerant services," *Proc. ACM SIGOPS Symp. Operating Systems Principles (SOSP '03)*, 2003, pp. 253–267.

[21] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Trans. Dependable Secur. Comput.*, vol. 3, no. 3, pp. 202–215, 2006.

[22] R. Rodrigues, P. Kouznetsov, and B. Bhattacharjee, "Large-scale byzantine fault tolerance: Safe but not always live," *Proc. 3rd workshop on Hot Topics in System Dependability (HotDep '07).* Berkeley, CA, USA: USENIX Association, 2007, p. 17.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *Proc. ACM SIGOPS Symp. Operating Systems Principles (SOSP '07)*, 2007, pp. 45–58.

[24] D. Malkhi and M. K. Reiter, "Secure and scalable replication in Phalanx," *Proc. IEEE Int'l Symp. Reliable Distributed Systems (SRDS '98)*, 1998, pp. 51–60.

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

[26] ——, "An architecture for survivable coordination in large distributed systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 12, no. 2, pp. 187–202, 2000.

[27] D. Malkhi, M. Reiter, D. Tulone, and E. Ziskind, "Persistent objects in the Fleet system," *The 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2001, pp. 126–136.

[28] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie, "Fault-scalable byzantine fault-tolerant services," *Proc. ACM Symp. Operating Systems Principles (SOSP '05)*, Brighton, UK, 2005, pp. 59–74.

[29] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo, "Efficient byzantine-resilient reliable multicast on a hybrid failure model," *Proc. IEEE Int'l Symp. Reliable Distributed Systems (SRDS '02)*, Suita, Japan, 2002, pp. 2–11.

[30] P. Veríssimo, "Uncertainty and predictability: Can they be reconciled?" *Future Directions in Distributed Computing*, ser. LNCS, no. 2584. Springer-Verlag, pp. 108–113, 2003.

[31] Y. G. Desmedt and Y. Frankel, "Threshold cryptosystems," *Proc. Advances in Cryptology (CRYPTO '89)*. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 307–315.

[32] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[33] V. Shoup, "Practical threshold signatures," *Lecture Notes in Computer Science*, vol. 1807, pp. 207–223, 2000.

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

[35] P. Feldman, "A practical scheme for non-interactive verifiable secret sharing," *Proc. 28th Ann. Symp. on Foundations of Computer Science*. Los Angeles, CA, 1987, pp. 427–437.

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

[37] 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.

[38] 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.

[39] "The spines project, http://www.spines.org/."

[40] Y. Amir and C. Danilov, "Reliable communication in overlay networks," *Proc. IEEE Int'l Conf. Dependable Systems and Networks (DSN '03)*, 2003, pp. 511–520.

[41] "Planetlab," http://www.planet-lab.org/.

[42] "The CAIRN Network," http://www.isi.edu/div7/CAIRN/.

[43] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu, "On the performance of consistent wide-area database replication," Tech. Rep. CNDS-2003-3, December 2003.

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.

Claudiu Danilov is an Advanced Computing Technologist at Boeing Phantom Works. Before he joined Boeing in 2006, he was an Assistant Research Scientist in the Department of computer Science, Johns Hopkins University. He received his BS degree in Computer Science in 1995 from Politehnica University of Bucharest, and MSE and PhD degrees in Computer Science from The Johns Hopkins University in 2000 and 2004. His research interests include wireless and mesh network protocols, distributed systems and survivable messaging systems. He is a creator of the Spines overlay network platform and the SMesh wireless mesh network.

Danny Dolev (SM'89) received his B.Sc. degree in mathematics and physics from the Hebrew University, Jerusalem in 1971. His M.Sc. thesis in Applied Mathematics was completed in 1973, at the Weizmann Institute of Science, Israel. His Ph.D. thesis was on Synchronization of Parallel Processors (1979). He was a Post-Doctoral fellow at Stanford University, 1979-1981, and IBM Research Fellow 1981-1982. He joined the Hebrew University in 1982. From 1987 to 1993 he held a joint appointment as a professor at the Hebrew University and as a research staff member at the IBM Almaden Research Center. He is currently a professor at the Hebrew University of Jerusalem. His research interests are all aspects of distributed computing, fault tolerance, and networking – theory and practice.

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

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

Cristina Nita-Rotaru is an Assistant Professor in the Computer Science department of the Purdue University and a member of Center for Education and Research in Information Assurance and Security at Purdue University. She received the BS and MSc degrees in Computer Science from Politechnica University of Bucharest, Romania, in 1995 and 1996, and the MSE and PhD degrees in Computer Science from The Johns Hopkins University in 2000 and 2003. Her research interests include secure distributed systems, network security protocols and security aspects in wireless networks. She is a member of the ACM and IEEE Computer Society.

Josh Olsen is a graduate student at the University of California, Irvine. He obtained his Bachelor of Science from Purdue University in 2005. His research interests include systems, security, and cryptography.

David Zage is a fourth year PhD student in the Computer Science Department at Purdue University under the supervision of Professor Cristina Nita-Rotaru. He obtained his Bachelor of Science from Purdue in 2004. He is a member of the Dependable and Secure Distributed Systems Laboratory ($DS2$). His research interests include distributed systems, fault tolerance, and security. He is a member of the ACM and IEEE Computer Society.