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. Digital Object Identifier no. XX.

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

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.

```
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.

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-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,

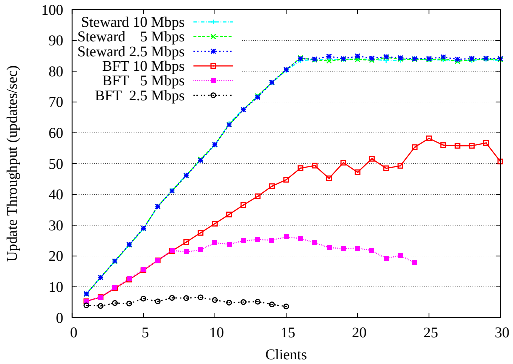


Fig. 4: Write Update Throughput

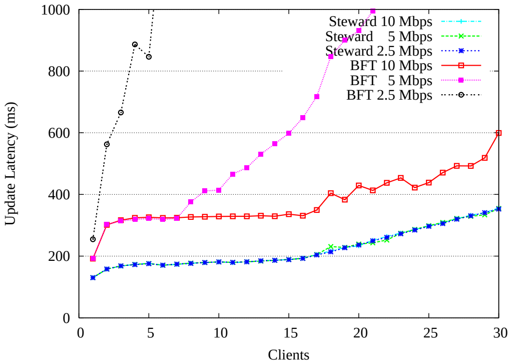


Fig. 5: Write Update Latency

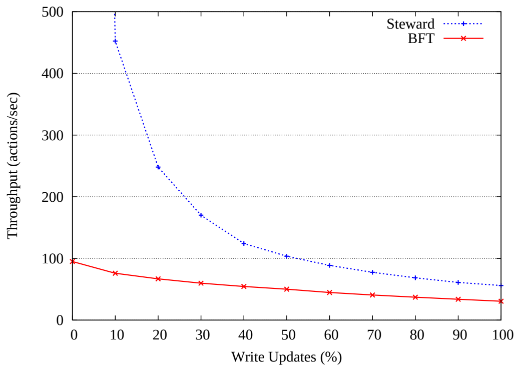


Fig. 6: Update Mix Throughput - 10 Clients

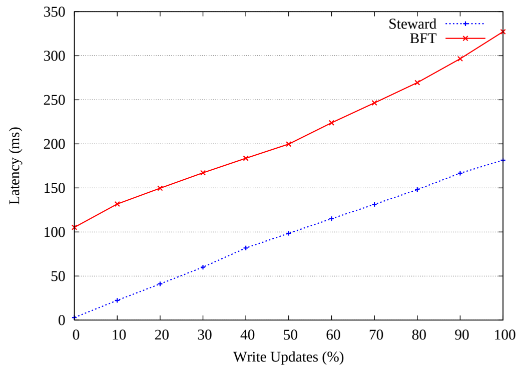


Fig. 7: Update Mix Latency - 10 Clients

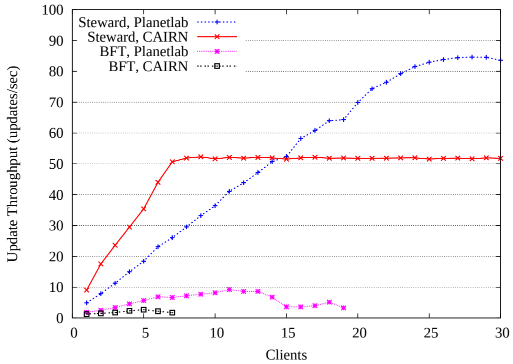


Fig. 8: WAN Emulation - Write Update Throughput

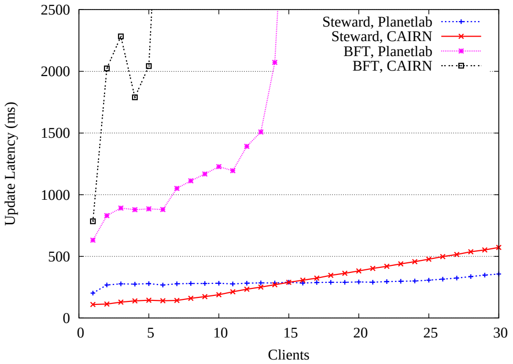


Fig. 9: WAN Emulation - Write Update Latency

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.

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.

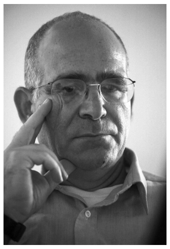
[41] "Planetlab," http://www.planet-lab.org/.

[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.

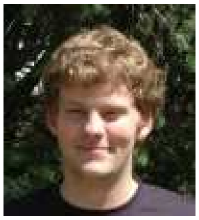


Danny Dolev (SM'89) received his B.Sc. degree in mathematics and physics from the Hebrew University, Jerusalem in 1971. His M.Sc. thesis in Applied Mathematics was completed in 1973, at the Weizmann Institute of Science, Israel. His Ph.D. thesis was on Synchronization of Parallel Processors (1979). He 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.

APPENDIX A
COMPLETE PSEUDOCODE

In this section we provide complete pseudocode for Steward. We then use this pseudocode in Appendix B to prove the safety and liveness of our protocol.

```
Standard Abbreviations: lv = local view; gv = global view; u = update; seq = sequence number;
ctx = context; sig = signature; partial_sig = partial signature; t_sig = threshold signature

// Message from client
Update = (client_id, timestamp, client_update, sig)

// Messages used by THRESHOLD-SIGN
Partial_Sig = (server_id, data, partial_sig, verification_proof, sig)
Corrupted_Server = (server_id, data, Partial_sig, sig)

// Messages used by ASSIGN-SEQUENCE
Pre-Prepare = (server_id, gv, lv, seq, Update, sig)
Prepare = (server_id, gv, lv, seq, Digest(Update), sig)
Prepare_Certificate( gv, lv, seq, u ) = a set containing a Pre-Prepare(server_id, gv, lv, seq, u, sig) message
and a list of 2f distinct Prepare(*, gv, lv, seq, Digest(u), sig) messages

// Messages used by ASSIGN-GLOBAL-ORDER
Proposal = (site_id, gv, lv, seq, Update, t_sig)
Accept = (site_id, gv, lv, seq, Digest(Update), t_sig)
Globally_Ordered_Update(gv, seq, u) = a set containing a Proposal(site_id, gv, lv, seq, u, t_sig) message and a
list of distinct Accept(*, seq, gv, *, Digest(u), t_sig) messages from a majority-1 of sites

// Messages used by LOCAL-VIEW-CHANGE
New_Rep = (server_id, suggested_lv, sig)
Local_Preinstall_Proof = a set of 2f+1 distinct New_Rep messages

// Messages used by GLOBAL-VIEW-CHANGE
Global_VC = (site_id, gv, t_sig)
Global_Preinstall_Proof = a set of distinct Global_VC messages from a majority of sites

// Messages used by CONSTRUCT-ARU, CONSTRUCT-LOCAL-CONSTRAINT, and CONSTRUCT-GLOBAL-CONSTRAINT
Request_Local_State = (server_id, gv, lv, seq)
Request_Global_State = (server_id, gv, lv, seq)
Local_Server_State = (server_id, gv, lv, invocation_aru, a set of Prepare Certificates, a set of Proposals,
sig)
Global_Server_State = (server_id, gv, lv, invocation_aru, a set of Prepare Certificates, a set of Proposals, a
set Globally_Ordered_Updates, sig)
Local_Collected_Server_State = (server_id, gv, lv, a set of 2f+1 Local_Server_State messages, sig)
Global_Collected_Server_State = (server_id, gv, lv, a set of 2f+1 Global_Server_State messages, sig)

//Messages used by GLOBAL-VIEW-CHANGE
Aru_Message = (site_id, gv, site_aru)
Global_Constraint = (site_id, gv, invocation_aru, a set of Proposals and/or Globally_Ordered_Updates with seq ≥
invocation_aru)
Collected_Global_Constraints(server_id, gv, lv, a set of majority Global_Constraint messages, sig)

//Messages used by GLOBAL-RECONCILIATION and LOCAL-RECONCILIATION
Global_Recon_Request = (server_id, global_session_seq, requested_aru, globally_ordered_update)
Local_Recon_Request = (server_id, local_session_seq, requested_aru)
Global_Recon = (site_id, server_id, global_session_seq, requested_aru)
```

Fig. A-1: Message types used in the global and local protocols.

```
int Server_id: unique id of this server within the site
int Site_id: unique id of this server's site

A. Global Context (Global Protocol) Data Structure
int Global_seq: next global sequence number to assign.
int Global_view: current global view of this server, initialized to 0.
int Global_preinstalled_view: last global view this server preinstalled, initialized to 0.
bool Installed_global_view: If it is 0, then Global_view is the new view to be installed.
Global_VC Latest_Global_VC[]: latest Global_VC message received from each site.
struct globally_proposed_item {
   Proposal_struct Proposal
   Accept_struct_List Accept_List
   Global_Ordered_Update_struct Globally_Ordered_Update
} Global_History[] // indexed by Global_seq
int Global_aru: global seq up to which this server has globally ordered all updates.
bool globally_constrained: set to true when constrained in global context.
int Last_Global_Session_Seq[]: latest session_seq from each server (local) or site (global)
int Last_Global_Requested_Aru[]: latest requested aru from each server (local) or site (global)
int Last_Global_Request_Time[]: time of last global reconciliation request from each local server
int Max_Global_Requested_Aru[]: maximum requested aru seen from each site

B. Local Context (Intra-site Protocols) Data Structure
int Local_view: local view number this server is in
int Local_preinstalled_vew: last local view this server preinstalled, initialized to 0.
bool Installed_local_view: If it is 0, then Global_view is the new one to be installed.
New_Rep Latest_New_Rep[]: latest New_Rep message received from each site.
struct pending_proposal_item {
   Pre-Prepare_struct Pre-Prepare
   Prepare_struct_List Prepare_List
   Prepare_Cert_struct Prepare_Certificate
   Proposal_struct Proposal
} Local_History[] //indexed by Global_seq
int Pending_proposal_aru: global seq up to which this server has constructed proposals
bool locally_constrained: set to true when constrained in the local context.
Partial_Sigs: associative container keyed by data. Each slot in the container holds an array, indexed by
server_id. To access data d from server s_id, we write Partial_Sigs{d}[s_id].
Update_Pool: pool of client updates, both unconstrained and constrained
int Last_Local_Session_Seq[]: latest session_seq from each local server
int Last_Local_Requested_Aru[]: latest requested aru from each local server
int Last_Local_Request_Time[]: time of last local reconciliation request from each local server
```

Fig. A-2: Global and Local data structures maintained by each server.

```
/* Notation: <== means append */
UPDATE-LOCAL-DATA-STRUCTURES:
  case message:
    Pre-Prepare(server_id, *, lv, seq, u):
      if Local_History[seq].Pre-Prepare is empty
        Local_History[seq].Pre-Prepare ← Pre-Prepare
      else
        ignore Pre-Prepare

    Prepare(server_id, *, lv, seq, digest):
      if Local_History[seq].Pre-Prepare is empty
        ignore Prepare
      if Local_History[seq].Prepare_List contains a Prepare with server_id
        ignore Prepare
      Local_History[seq].Prepare_List <== Prepare
      if Prepare_Certificate_Ready(seq)
        pre-prepare ← Local_History[seq].Pre-Prepare
        PC ← Construct_Prepare_Certificate(pre-prepare, Local_History[seq].Prepare_List)
        Local_History[seq].Prepare_Certificate ← PC

    Partial_Sig(server_id, data, partial_sig, verification_proof, sig):
      if Local_History.Partial_Sigs{ data }[Server_id] is empty
        ignore Partial_Sig
      Local_History.Partial_Sigs{ data }[server_id] ← Partial_Sig

    Local_Collected_Server_State(gv, lv, Local_Server_State[]):
      union ← Compute_Local_Union(Local_Collected_Server_State)
      invocation_aru ← Extract_Invocation_Aru(Local_Server_State[])
      max_local_entry ← Extract_Max_Local_Entry(Local_History[])
      for each seq from (invocation_aru+1) to max_local_entry
        if Local_History[seq].Prepare_Certificate(*, lv′, seq, *) exists and lv′ < lv
          clear Local_History[seq].Prepare_Certificate
        if Local_History[seq].Proposal(*, lv′, seq, *) exists and lv′ < lv
          clear Local_History[seq].Proposal
        if Local_History[seq].Pre-Prepare(*, lv′, seq, *) exists and lv′ < lv
          clear Local_History[seq].Pre-Prepare
      for each Prepare_Certificate(*, *, seq, *), PC, in union
        if Local_History[seq].Prepare_Certificate is empty
          Local_History[seq].Prepare_Certificate ← PC
      for each Proposal(*, *, seq, *), P, in union
        if Local_History[seq].Proposal is empty
          Local_History[seq].Proposal ← P

    New_Rep(site_id,lv):
      if (lv > Latest_New_Rep[site_id])
        Latest_New_Rep[site_id] ← New_Rep
      Local_preinstalled_view ← Latest_New_Rep[Site_id]

    Update($u$):
      SEND to all servers in site: Update($u$)
      if representative of non-leader site
        SEND to representative of leader site: Update($u$)
      Add Update($u$) to Update_Pool
```

Fig. A-3: Rules for applying a message to the Local_History data structure. The rules assume that there is no conflict, i.e., Conflict(message) == FALSE

```
/* Notation: <== means append */
UPDATE-GLOBAL-DATA-STRUCTURES:
  case message:
A1.   Proposal P(site_id, gv, *, seq, u):
A2.     if Global_History[seq].Proposal is empty
A3.       Global_History[seq].Proposal ← P
A4.       if server in leader site
A5.         Recompute Pending_proposal_aru
A6.       if Global_History[seq].Prepare_Certificate is not empty
A7.         remove Prepare_Certificate from Global_History[seq].Prepare_Certificate
A8.     if Global_History[seq].Proposal contains Proposal(site_id', gv', *, seq, u')
A9.       if gv > gv'
A10.        Global_History[seq].Proposal ← P
A11.        if server in leader site
A12.          Recompute Pending_proposal_aru
A13.        if Global_History[seq].Prepare_Certificate is not empty
A14.          remove Prepare_Certificate from Global_History[seq].Prepare_Certificate

B1.   Accept A(site_id, gv, *, seq, digest):
B2.     if Global_History[seq].Proposal is empty
B3.       ignore A
B4.     if Global_History[seq].Accept_List is empty
B5.       Global_History[seq].Accept_List <== A
B6.     if Global_History[seq].Accept_List has any Accept(site_id, gv', *, seq, digest')
B7.       if gv > gv'
B8.         discard all Accepts in Global_History[seq]
B9.         Global_History[seq].Accept_List <== A
B10.      if gv == gv' and Global_History[seq] does not have Accept from site_id
B11.        Global_History[seq].Accept_List <== A
B12.      if gv < gv'
B13.        ignore A
B14.    if Globally_Ordered_Ready(seq)
B15.      Construct globally_ordered_update from Proposal and list of Accepts
B16.      Apply globally_ordered_update to Global_History

C1.   Globally_Ordered_Update G(gv, seq, u):
C2.     if not Globally_Ordered(seq) and Is_Contiguous(seq)
C3.       Global_History[seq].Globally_Ordered_Update ← G
C4.       Recompute Global_aru
C5.       exec_set ← all unexecuted globally ordered updates with seq $\leq$ Global_aru
C6.       execute the updates in exec_set
C7.       if there exists at least one Globally_Ordered_Update(*, *, *) in exec_set
C8.         RESET-GLOBAL-TIMER()
C9.         RESET-LOCAL-TIMER()
C10.      if server in leader site
C11.        Recompute Pending_proposal_aru

D1.   Collected_Global_Constraints(gv, Global_Constraint[]):
D2.     union ← Compute_Constraint_Union(Collected_Global_Constraints)
D3.     invocation_aru ← Extract_Invocation_Aru(Global_Constraint[])
D4.     max_global_entry ← Extract_Max_Global_Entry(Global_History[])
D5.     for each seq from (invocation_aru+1) to max_global_entry
D6.       if Global_History[seq].Prepare_Certificate(gv', *, seq, *) exists and gv' < gv
D7.         clear Global_History[seq].Prepare_Certificate
D8.       if Global_History[seq].Proposal(gv', *, seq, *) exists and gv' < gv
D9.         clear Global_History[seq].Proposal
D10.    for each Globally_Ordered_Update(*, *, seq, *), G, in union
D11.      Global_History[seq].Globally_Ordered_Update ← G
D12.    for each Proposal(*, *, seq, *), P, in union
D13.      if Global_History[seq].Proposal is empty
D14.        Global_History[seq].Proposal ← P

E1.   Global_VC(site_id, gv):
E2.     if ( gv > Latest_Global_VC[site_id].gv )
E3.       Latest_Global_VC[site_id] ← Global_VC
E4.       sorted_vc_messages ← sort Latest_Global_VC by gv
E5.       Global_preinstalled_view ← sorted_vc_messages[ $\lfloor N/2 \rfloor + 1$ ].gv
E6.     if ( Global_preinstalled_view > Global_view )
E7.       Global_view ← Global_preinstalled_view
E8.       globally_constrained ← False

F1.   Global_Preinstall_Proof(global_vc_messages[]):
F2.     for each Global_VC(gv) in global_vc_messsages[]
F3.       Apply Global_VC
```

Fig. A-4: Rules for applying a message to the Global_History data structure. The rules assume that there is no conflict, i.e., Conflict(message) == FALSE

```
boolean Globally_Ordered(seq):
  if Global_History[seq].Globally_Ordered_Update is not empty
     return TRUE
  return FALSE

boolean Globally_Ordered_Ready(seq):
  if Global_History.Proposal[seq] contains a Proposal(site_id, gv, lv, seq, u)
     if Global_History[seq].Accept_List contains (majority-1) of distinct
     Accept(site_id(i), gv, lv, seq, Digest(u)) with site_id(i) ≠ site_id
        return TRUE
     if Global_History[seq].Accept_List contains a majority of distinct
        Accept(site_id(i), gv', lv, seq, Digest(u)) with gv >= gv'
        return TRUE
  return FALSE

boolean Prepare_Certificate_Ready(seq):
  if Local_History.Proposal[seq] contains a Pre-Prepare(server_id, gv, lv, seq, u)
     if Local_History[seq].Prepare_List contains 2f distinct
     Prepare(server_id(i), gv, lv, seq, d) with server_id ≠ server_id(i) and d == Digest(u)
        return TRUE
  return FALSE

boolean In_Window(seq):
  if Global_aru < seq ≤ Global_aru + W
     return TRUE
  else
     return FALSE

boolean Is_Contiguous(seq):
  for i from Global_aru+1 to seq-1
     if Global_History[seq].Prepare-Certificate == NULL and
           Global_History[seq].Proposal == NULL and
           Global_History[seq].Globally_Ordered_Update == NULL and
           Local_History[seq].Prepare-Certificate == NULL and
           Local_History[seq].Proposal == NULL
        return FALSE
  return TRUE
```

Fig. A-5: Predicate functions used by the global and local protocols to determine if and how a message should be applied to a server's data structures.

```
boolean Valid(message):
  if message has threshold RSA signature S
     if NOT VERIFY(S)
        return FALSE
  if message has RSA-signature S
     if NOT VERIFY(S)
        return FALSE
  if message contains update with client signature C
     if NOT VERIFY(C)
        return FALSE
  if message.sender is in Corrupted_Server_List
     return FALSE
  return TRUE
```

Fig. A-6: Validity checks run on each incoming message. Invalid messages are discarded.

```
boolean Conflict(message):
  case message
    Proposal((site_id, gv, lv, seq, u):
      if gv $\neq$ Global_view
        return TRUE
      if server in leader site
        return TRUE
      if Global_History[seq].Global_Ordered_Update(gv’, seq, u’) exists
        if (u’ $\neq$ u) or (gv’ > gv)
          return TRUE
      if not Is_Contiguous(seq)
        return TRUE
      if not In_Window(seq)
        return TRUE
    return FALSE

    Accept(site_id, gv, lv, seq, digest):
      if gv $\neq$ Global_view
        return TRUE
      if (Global_History[seq].Proposal(*, *, *, seq, u’) exists) and (Digest(u’) $\neq$ digest)
        return TRUE
      if Global_History[seq].Global_Ordered_Update(gv’, seq, u’) exists
        if (Digest(u’) $\neq$ digest) or (gv’ > gv)
          return TRUE
    return FALSE

    Aru_Message(site_id, gv, site_aru):
      if gv $\neq$ Global_view
        return TRUE
      if server in leader site
        return TRUE
    return FALSE

    Request_Global_State(server_id, gv, lv, aru):
      if (gv $\neq$ Global_view) or (lv $\neq$ Local_view)
        return TRUE
      if server_id $\neq$ lv mod num_servers_in_site
        return TRUE
    return FALSE

    Global_Server_State(server_id, gv, lv, seq, state_set):
      if (gv $\neq$ Global_view) or (lv $\neq$ Local_view)
        return TRUE
      if not representative
        return TRUE
      if entries in state_set are not contiguous above seq
        return TRUE
     return FALSE

    Global_Collected_Servers_State(server_id, gv, lv, gss_set):
      if (gv $\neq$ Global_view) or (lv $\neq$ Local_view)
        return TRUE
      if each message in gss_set is not contiguous above invocation_seq
        return TRUE

    Global_Constraint(site_id, gv, seq, state_set):
      if gv $\neq$ Global_view
        return TRUE
      if server not in leader site
        return TRUE
    return FALSE

    Collected_Global_Constraints(server_id, gv, lv, gc_set):
      if gv $\neq$ Global_view
        return TRUE
      aru $\leftarrow$ Extract_Aru(gc_set)
      if Global_aru $<$ aru
        return TRUE
    return FALSE
```

Fig. A-7: Conflict checks run on incoming messages used in the global context. Messages that conflict with a server’s current global state are discarded.

```
boolean Conflict(message):
  case message
    Pre-Prepare(server_id, gv, lv, seq, u):
      if not (globally_constrained && locally_constrained)
        return TRUE
      if server_id ≠ lv mod num_servers_in_site
        return TRUE
      if (gv ≠ Global_view) or (lv ≠ Local_view)
        return TRUE
      if Local_History[seq].Pre-Prepare(server_id, gv, lv, seq, u′) exists and u′ ≠ u
        return TRUE
      if Local_History[seq].Prepare_Certificate.Pre-Prepare(gv, lv′, seq, u′) exists and u′ ≠ u
        return TRUE
      if Local_History[seq].Proposal(site_id, gv, lv′, u′) exists
        if (u′ ≠ u) or (lv′ > lv)
          return TRUE
      if Global_History[seq].Proposal(site_id, gv′, lv′, seq, u′) exists
        if (u′ ≠ u) or (gv′ > gv)
          return TRUE
      if Global_History[seq].Globally_Ordered_Update(*, seq, u′) exists
        if (u′ ≠ u)
          return TRUE
      if not Is_Contiguous(seq)
        return TRUE
      if not In_Window(seq)
        return TRUE
      if u is bound to seq′ in Local_History or Global_History
        return TRUE
    return FALSE

    Prepare(server_id, gv, lv, seq, digest):
      if not (globally_constrained && locally_constrained)
        return TRUE
      if (gv ≠ Global_view) or (lv ≠ Local_view)
        return TRUE
      if Local_History[seq].Pre-Prepare(server_id′, gv, lv, seq, u) exists
        if digest ≠ Digest(u)
          return TRUE
      if Local_History[seq].Prepare_Certificate.Pre-Prepare(gv, lv′, seq, u) exists
        if (digest ≠ Digest(u)) or (lv′ > lv)
          return TRUE
      if Local_History[seq].Proposal(gv, lv′, seq, u) exists
        if (digest ≠ Digest(u)) or (lv′ > lv)
          return TRUE
    return FALSE

    Request_Local_State(server_id, gv, lv, aru):
      if (gv ≠ Global_view) or (lv ≠ Local_view)
        return TRUE
      if server_id ≠ lv mod num_servers_in_site
        return TRUE
    return FALSE

    Local_Server_State(server_id, gv, lv, seq, state_set):
      if (gv ≠ Global_view) or (lv ≠ Local_view)
        return TRUE
      if not representative
        return TRUE
      if entries in state_set are not contiguous above seq
        return TRUE
    return FALSE

    Local_Collected_Servers_State(server_id, gv, lv, lss_set):
      if (gv ≠ Global_view) or (lv ≠ Local_view)
        return TRUE
      if each message in lss_set is not contiguous above invocation_seq
        return TRUE
    return FALSE
```

Fig. A-8: Conflict checks run on incoming messages used in the local context. Messages that conflict with a server's current local state are discarded.

```
THRESHOLD-SIGN(Data_s data, int server_id):
Partial_Sig ← GENERATE_PARTIAL_SIG(data, server_id)
SEND to all local servers: Partial_Sig

Upon receiving a set, PSig_Set, of 2f+1 Partial_Sigs from distinct servers:
   signature ← COMBINE(PSig_Set)
   if VERIFY(signature)
      return signature
   else
      for each S in PSig_Set
         if NOT VERIFY(S)
            REMOVE(S, PSig_Set)
            ADD(S.server_id, Corrupted_Servers_List)
            Corrupted_Server ← CORRUPTED(S)
            SEND to all local servers: Corrupted_Server
            continue to wait for more Partial_Sig messages
```

Fig. A-9: 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):
Upon invoking:
  SEND to all local servers: Pre-Prepare(gv, lv, Global_seq, u)
  Global_seq++

Upon receiving Pre-Prepare(gv, lv, seq, u):
  Apply Pre-Prepare to Local_History
  SEND to all local servers: Prepare(gv, lv, seq, Digest(u))

Upon receiving Prepare(gv, lv, seq, digest):
  Apply Prepare to Local_History
  if Prepare_Certificate_Ready(seq)
     prepare_certificate ← Local_History[seq].Prepare_Certificate
     pre-prepare ← prepare_certificate.Pre-Prepare
     unsigned_proposal ← ConstructProposal(pre-prepare)
     invoke THRESHOLD-SIGN(unsigned_proposal, Server_id) //returns signed_proposal

Upon THRESHOLD-SIGN returning signed_proposal:
  Apply signed_proposal to Global_History
  Apply signed_proposal to Local_History
  return signed_proposal
```

Fig. A-10: ASSIGN-SEQUENCE Protocol, used to bind an update to a sequence number and produce a threshold-signed Proposal message.

```
ASSIGN-GLOBAL-ORDER():
Upon receiving or executing an update, or becoming globally or locally constrained:
  if representative of leader site
     if (globally_constrained and locally_constrained and In_Window(Global_seq))
        u ← Get_Next_To_Propose()
        if (u ≠ NULL)
           invoke ASSIGN-SEQUENCE(u) //returns Proposal

Upon ASSIGN-SEQUENCE returning Proposal:
  SEND to all sites: Proposal

Upon receiving Proposal(site_id, gv, lv, seq, u):
  Apply Proposal to Global_History
  if representative
     SEND to all local servers: Proposal
  unsigned_accept ← ConstructAccept(Proposal)
  invoke THRESHOLD-SIGN(unsigned_accept, Server_id) //returns signed_accept

Upon THRESHOLD-SIGN returning signed_accept:
  Apply signed_accept to Global_History
  if representative
     SEND to all sites: signed_accept

Upon receiving Accept(site_id, gv, lv, seq, Digest(u)):
  Apply Accept to Global_History
  if representative
     SEND to all local servers: Accept
  if Globally_Ordered_Ready(seq)
     globally_ordered_update ← ConstructOrderedUpdate(seq)
     Apply globally_ordered_update to Global_History
```

Fig. A-11: ASSIGN-GLOBAL-ORDER Protocol. The protocol runs among all sites and is similar to Paxos. It invokes the ASSIGN-SEQUENCE and THRESHOLD-SIGN intra-site protocols to allow a site to emulate the behavior of a Paxos participant.

```
Get_Next_To_Propose():
u ← NULL
if(Global_History[Global_seq].Proposal is not empty)
    u ← Global_History[Global_seq].Proposal.Update
else if(Local_History[Global_seq].Prepare_Certificate is not empty)
    u ← Local_History[Global_seq].Prepare_Certificate.Update
else if(Unconstrained_Updates is not empty)
    u ← Unconstrained_Updates.Pop_Front()
return u
```

Fig. A-12: Get_Next_To_Propose Procedure. For a given sequence number, the procedure returns (1) the update currently bound to that sequence number, (2) some update not currently bound to any sequence number, or (3) NULL if the server does not have any unbound updates.

```
Initial State:
Local_view = 0
my_preinstall_proof = a priori proof that view 0 was preinstalled
RESET-LOCAL-TIMER()

LOCAL-VIEW-CHANGE()
A1. Upon Local_T expiration:
A2.   Local_view++
A3.   locally_constrained ← False
A4.   unsigned_new_rep ← Construct_New_Rep(Local_view)
A5.   invoke THRESHOLD-SIGN(unsigned_new_rep, Server_id) //returns New_Rep

B1. Upon THRESHOLD-SIGN returning New_Rep(lv):
B2.   Apply New_Rep()
B3.   SEND to all servers in site: New_Rep

C1. Upon receiving New_Rep(lv):
C2.   Apply New_Rep()

D1. Upon increasing Local_preinstalled_view:
D2.   RELIABLE-SEND-TO-ALL-SITES(New_Rep)
D3.   SEND to all servers in site: New_Rep
D4.   RESET-LOCAL-TIMER(); Start Local_T
D5.   if representative of leader site
D6.      invoke CONSTRUCT-LOCAL-CONSTRAINT(Pending_proposal_aru)
D7.      if NOT globally_constrained
D8.         invoke GLOBAL_VIEW_CHANGE
D9.      else
D10.        my_global_constraints ← Construct_Collected_Global_Constraints()
D11.        SEND to all servers in site: My_global_constraints
```

Fig. A-13: LOCAL-VIEW-CHANGE Protocol, used to elect a new site representative when the current one is suspected to have failed. The protocol also ensures that the servers in the leader site have enough knowledge of pending decisions to preserve safety in the new local view.

```
GLOBAL-LEADER-ELECTION:
A1. Upon Global_T expiration:
A2.   Global_view++
A3.   globally_constrained ← False
A4.   unsigned_global_vc ← Construct_Global_VC()
A5.   invoke THRESHOLD-SIGN(unsigned_global_vc, Server_id)

B1. Upon THRESHOLD-SIGN returning Global_VC(gv):
B2.   Apply Global_VC to data structures
B3.   ReliableSendToAllSites(Global_VC)

C1. Upon receiving Global_VC(gv):
C2.   Apply Global_VC to data structures

D1. Upon receiving Global_Preinstall_Proof(gv):
D2.   Apply Global_Preinstall_Proof()

E1. Upon increasing Global_preinstalled_view:
E2.   sorted_vc_messages ← sort Latest_Global_VC by gv
E3.   proof ← last $\lfloor N/2 \rfloor + 1$ Global_VC messages in sorted_vc_messages
E4.   ReliableSendToAllSites( proof )
E5.   SEND to all local servers: proof
E6.   RESET-GLOBAL-TIMER(); Start Global_T
E7.   if representative of leader site
E8.      invoke GLOBAL-VIEW-CHANGE
```

Fig. A-14: GLOBAL-LEADER-ELECTION Protocol. When the Global_T timers of at least $2f + 1$ servers in a majority of sites expire, the sites run a distributed, global protocol to elect a new leader site by exchanging threshold-signed Global_VC messages.

```
RESET-GLOBAL-PROGRESS-TIMER():
A1.   Global_T ← GLOBAL-TIMEOUT()

RESET-LOCAL-TIMER():
B1.   if in leader site
B2.      Local_T ← GLOBAL-TIMEOUT()$/(f+3)$
B3.   else
B4.      Local_T ← GLOBAL-TIMEOUT()$/(f+3)(f+2)$

GLOBAL_TIMEOUT():
C1.   return $K * 2^{\lceil Global\_view/N \rceil}$
```

Fig. A-15: RESET-GLOBAL-TIMER and RESET-LOCAL-TIMER procedures. These procedures establish the relationships between Steward's timeout values at both the local and global levels of the hierarchy. Note that the local timeout at the leader site is longer than at the non-leader sites to ensure a correct representative of the leader site has enough time to communicate with correct representatives at the non-leader sites. The values increase as a function of the global view.

```
GLOBAL-VIEW-CHANGE:
A1. Upon invoking:
A2.   Invoke CONSTRUCT-ARU(Global_aru)// returns (Global_Constraint, Aru_Message)

B1. Upon CONSTRUCT-ARU returning (Global_Constraint, Aru_Message):
B2.   Store Global_Constraint
B3.   if representative of leader site
B4.      SEND to all sites: Aru_Message

C1. Upon receiving Aru_Message(site_id, gv, site_aru):
C2.   if representative site
C3.      SEND to all servers in site: Aru_Message
C4.   invoke CONSTRUCT-GLOBAL-CONSTRAINT(Aru_Message) //returns Global_Constraint

D1. Upon CONSTRUCT-GLOBAL-CONSTRAINT returning Global_Constraint:
D2.   if representative of non-leader site
D3.      SEND to representative of leader site: Global_Constraint

E1. Upon collecting GC_SET with majority distinct Global_Constraint messages:
E2.   if representative
E3.      Collected_Global_Constraints $\longleftarrow$ ConstructBundle(GC_SET)
E4.      SEND to all in site: Collected_Global_Constraints
E5.      Apply Collected_Global_Constraints to Global_History
E6.      globally_constrained $\longleftarrow$ True

F1. Upon receiving Collected_Global_Constraints:
F2.   Apply Collected_Global_Constraints to Global_History
F3.   globally_constrained $\longleftarrow$ True
F4.   Pending_proposal_aru $\longleftarrow$ Global_aru
```

Fig. A-16: GLOBAL-VIEW-CHANGE Protocol, used to globally constrain the servers in a new leader site. These servers obtain information from a majority of sites, ensuring that they will respect the bindings established by any updates that were globally ordered in a previous view.

```
CONSTRUCT-LOCAL-CONSTRAINT(int seq):
A1. if representative
A2.   Request_Local_State $\longleftarrow$ ConstructRequestState(Global_view, Local_view, seq)
A3.   SEND to all local servers: Request_Local_State

B1. Upon receiving Request_Local_State(gv, lv, s):
B2.   invocation_aru $\longleftarrow$ s
B3.   if (Pending_Proposal_Aru < s)
B4.      Request missing Proposals or Globally_Ordered_Update messages from representative
B5.   if (Pending_Proposal_Aru $\geq$ s)
B6.      Local_Server_State $\longleftarrow$ Construct_Local_Server_State(s)
B7.      SEND to the representative: Local_Server_State

C1. Upon collecting LSS_Set with 2f+1 distinct Local_Server_State(invocation_aru) messages:
C2.   Local_Collected_Servers_State $\longleftarrow$ Construct_Bundle(LSS_Set)
C3.   SEND to all local servers: Local_Collected_Servers_State

D1. Upon receiving Local_Collected_Servers_State:
D2.   if (all Local_Server_State messages in bundle contain invocation_aru)
D3.      if (Pending_Proposal_Aru $\geq$ invocation_aru)
D4.         Apply Local_Collected_Servers_State to Local_History
D5.         locally_constrained $\longleftarrow$ True
D6.         return Local_Collected_Servers_State
```

Fig. A-17: CONSTRUCT-LOCAL-CONSTRAINT Protocol. The protocol is invoked by a newly-elected leader site representative and involves the participation of all servers in the leader site. Upon completing the protocol, a server becomes locally constrained and will act in a way that enforces decisions made in previous local views.

```
CONSTRUCT-ARU(int seq):
A1. if representative
A2.   Request_Global_State ← ConstructRequestState(Global_view, Local_view, seq)
A3.   SEND to all local servers: Request_Global_State

B1. Upon receiving Request_Global_State(gv, lv, s):
B2.   invocation_aru ← s
B3.   if (Global_aru < s)
B4.      Request missing Globally_Ordered_Updates from representative
B5.   if (Global_aru ≥ s)
B6.      Global_Server_State ← Construct_Global_Server_State(s)
B7.      SEND to the representative: Global_Server_State

C1. Upon collecting GSS_Set with 2f+1 distinct Global_Server_State(invocation_aru) messages:
C2.   Global_Collected_Servers_State ← Construct_Bundle(GSS_Set)
C3.   SEND to all local servers: Global_Collected_Servers_State

D1. Upon receiving Global_Collected_Servers_State:
D2.   if (all Global_Server_State message in bundle contain invocation_aru)
D3.      if(Global_aru ≥ invocation_aru)
D4.         union ← Compute_Global_Union(Global_Collected_Servers_State)
D5.         for each Prepare Certificate, PC(gv, lv, seq, u), in union
D6.            Invoke THRESHOLD-SIGN(PC, Server_id) //Returns Proposal

E1. Upon THRESHOLD-SIGN returning Proposal P(gv, lv, seq, u):
E2.   Global_History[seq].Proposal ← P

F1. Upon completing THRESHOLD-SIGN on all Prepare Certificates in union:
F2.   Invoke THRESHOLD-SIGN(union, Server_id) //Returns Global_Constraint

G1. Upon THRESHOLD-SIGN returning Global_Constraint:
G2.   Apply each Globally_Ordered_Update in ConstraintMessage to Global_History
G3.   union_aru ← Extract_Aru(union)
G4.   Invoke THRESHOLD-SIGN(union_aru, Server_id) //Returns Aru_Message

H1. Upon THRESHOLD-SIGN returning Aru_Message:
H2.   return (Global_Constraint, Aru_Message)
```

Fig. A-18: CONSTRUCT-ARU Protocol, used by the leader site to generate an Aru_Message during a global view change. The Aru_Message contains a sequence number through which at least $f+1$ correct servers in the leader site have globally ordered all updates.

```
CONSTRUCT-GLOBAL-CONSTRAINT(Aru_Message A):
A1. invocation_aru ← A.seq
A2. Global_Server_State ← Construct_Global_Server_State(global_context, A.seq)
A3. SEND to the representative: Global_Server_State

B1. Upon collecting GSS_Set with 2f+1 distinct Global_Server_State(invocation_aru) messages:
B2.    Global_Collected_Servers_State ← Construct_Bundle(GSS_Set)
B3.    SEND to all local servers: Global_Collected_Servers_State

C1. Upon receiving Global_Collected_Servers_State:
C2.    if (all Global_Server_State messages in bundle contain invocation_aru)
C3.       union ← Compute_Global_Union(Global_Collected_Servers_State)
C4.       for each Prepare Certificate, PC(gv, lv, seq, u), in union
C5.          Invoke THRESHOLD-SIGN(PC, Server_id) //Returns Proposal

D1. Upon THRESHOLD-SIGN returning Proposal P(gv, lv, seq, u):
D2.    Global_History[seq].Proposal ← P

E1. Upon completing THRESHOLD-SIGN on all Prepare Certificates in union:
E2.    Invoke THRESHOLD-SIGN(union, Server_id) //Returns Global_Constraint

F1. Upon THRESHOLD-SIGN returning Global_Constraint:
F2.    return Global_Constraint
```

Fig. A-19: CONSTRUCT-GLOBAL-CONSTRAINT Protocol, used by the non-leader sites during a global view change to generate a Global_Constraint message. The Global_Constraint contains Proposals and Globally_Ordered_Updates for all sequence numbers greater than the sequence number contained in the Aru_Message, allowing the servers in the leader site to enforce decisions made in previous global views.

```
Construct_Local_Server_State(seq):
state_set ← $\emptyset$
For each sequence number i from (seq + 1) to (Global_Aru + W):
  if Local_History[i].Proposal, P, exists
     state_set ← state_set ∪ P
  else if Local_History[i].Prepare_Certificate, PC, exists:
     state_set ← state_set ∪ PC
return Local_Server_State(Server_id, gv, lv, seq, state_set)

Construct_Global_Server_State(seq):
state_set ← $\emptyset$
For each sequence number i from (seq + 1) to (Global_aru + W):
  if Global_History[i].Globally_Ordered_Update, G, exists
     state_set ← state_set ∪ G
  else if Global_History[i].Proposal, P, exists:
     state_set ← state_set ∪ P
  else if Global_History[i].Prepare_Certificate, PC, exists:
     state_set ← state_set ∪ PC
return Global_Server_State(Server_id, gv, lv, seq, state_set)
```

Fig. A-20: Construct Server State Procedures. During local and global view changes, individual servers use these procedures to generate Local_Server_State and Global_Server_State messages. These messages contain entries for each sequence number, above some invocation sequence number, to which a server currently has an update bound.

```
// Assumption: all entries in css are from Global_view
Compute_Local_Union(Local_Collected_Servers_State css):
union ← $\emptyset$
css_unique ← Remove duplicate entries from css
seq_list ← Sort entries in css_unique by increasing (seq, lv)

For each item in seq_list
   if any Proposal P
      $P^*$ ← Proposal from latest local view
      union ← union ∪ $P^*$
   else if any Prepare Certificate PC
      $PC^*$ ← PC from latest local view
      union ← union ∪ $PC^*$
return union

Compute_Global_Union(Global_Collected_Servers_State css):
union ← $\emptyset$
css_unique ← Remove duplicate entries from css
seq_list ← Sort entries in css_unique by increasing (seq, gv, lv)

For each item in seq_list
   if any Globally_Ordered_Update
      $G^*$ ← Globally_Ordered_Update with Proposal from latest view (gv, lv)
      union ← union ∪ $G^*$
   else
      MAX_GV ← global view of entry with latest global view
      if any Proposal from MAX_GV
         $P^*$ ← Proposal from MAX_GV and latest local view
         union ← union ∪ $P^*$
      else if any Prepare Certificate PC from MAX_GV
         $PC^*$ ← PC from MAX_GV and latest local view
         union ← union ∪ $PC^*$
return union

Compute_Constraint_Union(Collected_Global_Constraints cgc):
union ← $\emptyset$
css_unique ← Remove duplicate entries from cgc
seq_list ← Sort entries in css_unique by increasing (seq, gv)

For each item in seq_list
   if any Globally_Ordered_Update
      $G^*$ ← Globally_Ordered_Update with Proposal from latest view (gv, lv)
      union ← union ∪ $G^*$
   else
      MAX_GV ← global view of entry with latest global view
      if any Proposal from MAX_GV
         $P^*$ ← Proposal from MAX_GV and latest local view
         union ← union ∪ $P^*$
return union
```

Fig. A-21: Compute_Union Procedures. The procedures are used during local and global view changes. For each entry in the input set, the procedures remove duplicates (based on sequence number) and, for each sequence number, take the appropriate entry from the latest view.

```
LOCAL-RECONCILIATION:
A1. Upon expiration of LOCAL_RECON_TIMER:
A2.   local_session_seq++
A3.   requested_aru $\leftarrow$ Global_aru
A4.   Local_Recon_Request $\leftarrow$ ConstructRequest(server_id, local_session_seq, requested_aru)
A5.   SEND to all local servers: Local_Recon_Request
A6.   Set LOCAL_RECON_TIMER

B1. Upon receiving Local_Recon_Request(server_id, local_session_seq, requested_aru):
B2.   if local_session_seq $\leq$ last_session_seq[server_id]
B3.      ignore Local_Recon_Request
B4.   if (current_time - last_local_request_time[server_id]) < LOCAL_RECON_THROTTLE_PERIOD
B5.      ignore Local_Recon_Request
B6.   if requested_aru < last_local_requested_aru[server_id]
B7.      ignore Local_Recon_Request
B8.   last_local_session_seq[server_id] $\leftarrow$ local_session_seq
B9.   last_local_request_time[server_id] $\leftarrow$ current_time
B10.  last_local_requested_aru[server_id] $\leftarrow$ requested_aru
B11.  if Global_aru > requested_aru
B12.     THROTTLE-SEND(requested_aru, Global_aru, LOCAL_RATE, W) to server_id
```

Fig. A-22: LOCAL-RECONCIILIATION Protocol, used to recover missing Globally_Ordered_Updates within a site. Servers limit both the rate at which they will respond to requests and the rate at which they will send requested messages.

```
GLOBAL-RECONCILIATION:
A1. Upon expiration of GLOBAL_RECON_TIMER:
A2.   global_session_seq++
A3.   requested_aru $\leftarrow$ Global_aru
A4.   g $\leftarrow$ Global_History[requested_aru].Globally_Ordered_Update
A5.   Global_Recon_Request $\leftarrow$ ConstructRequest(server_id,global_session_seq,requested_aru,g)
A6.   SEND to all local servers: Global_Recon_Request
A7.   Set GLOBAL_RECON_TIMER

B1. Upon receiving Global_Recon_Request(server_id, global_session_seq, requested_aru, g):
B2.   if global_session_seq $\leq$ last_global_session_seq[server_id]
B3.      ignore Global_Recon_Request
B4.   if (current_time - last_global_request_time[server_id]) < GLOBAL_RECON_THROTTLE_PERIOD
B5.      ignore Global_Recon_Request
B6.   if requested_aru < last_global_requested_aru[server_id]
B7.      ignore Global_Recon_Request
B8.   if g is not a valid Globally_Ordered_Update for requested_aru
B9.      ignore Global_Recon_Request
B10.  last_global_session_seq[server_id] $\leftarrow$ global_session_seq
B11.  last_global_request_time[server_id] $\leftarrow$ current_time
B12.  last_global_requested_aru[server_id] $\leftarrow$ requested_aru
B13.  if Global_aru $\geq$ requested_aru
B14.     sig_share $\leftarrow$ GENERATE_SIGNATURE_SHARE()
B15.     SEND to server_id: sig_share
B16.  if Global_aru $<$ requested_aru
B17.     when Global_aru $\geq$ requested_aru:
B18.        sig_share $\leftarrow$ GENERATE_SIGNATURE_SHARE()
B19.        SEND sig_share to server_id

C1. Upon collecting $2f + 1$ Partial_sig messages for global_session_seq:
C2.   GLOBAL_RECON $\leftarrow$ COMBINE(partial_sigs)
C3.   SEND to peer server in each site: GLOBAL_RECON

D1. Upon receiving GLOBAL_RECON(site_id, server_id, global_session_seq, requested_aru):
D2.   if max_global_requested_aru[site_id] $\leq$ requested_aru
D3.      max_global_requested_aru[site_id] $\leftarrow$ requested_aru
D4.   else
D5.      ignore GLOBAL_RECON
D6.   if (site_id == Site_id) or (server_id $\neq$ Server_id)
D7.      ignore GLOBAL_RECON
D8.   if global_session_seq $\leq$ last_global_session_seq[site_id]
D9.      ignore GLOBAL_RECON
D10.  if (current_time - last_global_request_time[site_id]) < GLOBAL_RECON_THROTTLE_PERIOD
D11.     ignore GLOBAL_RECON
D12.  SEND to all local servers: GLOBAL_RECON
D13.  last_global_session_seq[site_id] $\leftarrow$ global_session_seq
D14.  last_global_request_time[site_id] $\leftarrow$ current_time
D15.  if Global_aru > requested_aru
D16.     THROTTLE-SEND(requested_aru, Global_aru, GLOBAL_RATE, W) to server_id
```

Fig. A-23: GLOBAL-RECONCIILIATION Protocol, used by a site to recover missing Globally_Ordered_Updates from other wide area sites. Each server generates threshold-signed reconciliation requests and communicates with a single server at each other site.

```
RELIABLE-SEND-TO-ALL-SITES( message $m$ ):
A1. Upon invoking:
A2.   rel_message $\longleftarrow$ ConstructReliableMessage($m$)
A3.   SEND to all servers in site: rel_message
A4.   SendToPeers($m$)

B1. Upon receiving message Reliable_Message($m$):
B2.   SendToPeers($m$)

C1. Upon receiving message $m$ from a server with my id:
C2.   SEND to all servers in site: $m$

SendToPeers($m$):
D1.   if $m$ is a threshold signed message from my site and my Server_id $\leq 2f+1$:
D2.      my_server_id $\longleftarrow$ Server_id
D3.      for each site $S$:
D4.         SEND to server in site $S$ with Server_id $=$ my_server_id: $m$
```

Fig. A-24: RELIABLE-SEND-TO-ALL-SITES Protocol. Each of $2f+1$ servers within a site sends a given message to a peer server in each other site. When sufficient connectivity exists, the protocol reliably sends a message from one site to all other servers in all other sites sites despite the behavior of faulty servers.

APPENDIX B
PROOFS OF CORRECTNESS

In this section we show that Steward provides the service properties specified in Section 5. We begin with a proof of safety and then consider liveness.

B.1 Proof of Safety

Our goal in this section is to prove that Steward meets the following safety property:

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

Proof Strategy: We prove Safety by showing that two servers cannot globally order conflicting updates for the same sequence number. We show this using 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. To prove this claim, 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. To prove this claim, 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. The value that may be contained in a Proposal for this sequence number is thus *anchored*. 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 will therefore execute the same update.

We now proceed to prove the first main claim:

Claim *A.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 *A.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 A.1 with a series of lemmas. We begin with two preliminary lemmas, proving that two servers cannot collect conflicting Prepare Certificates or construct conflicting Proposals in the same global and local view.

Lemma *A.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 ($u \neq u'$).

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 A-10 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 (Figure A-8, line A8). Thus, this server did not contribute to PC2, a contradiction. □

Lemma *A.3:* Let P1(gv, lv, seq, u) be a Proposal message constructed by some server in leader site S. Then no other Proposal message P2(gv, lv, seq, u') with ($u \neq u'$) can be constructed by any server in S.

Proof: By Lemma A.2, only one Prepare Certificate can be constructed in each view (gv, lv) for a given sequence number seq. For P2 to be constructed, 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 A-10, line C7). Since P1 was constructed, there must have been a Prepare Certificate PC1 reflecting the binding of seq to u. Thus, the $f + 1$ correct servers cannot have obtained PC2, since this would contradict Lemma A.2. □

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

INVARIANT A.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 A.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 $gv' \geq gv$.

We first show that the invariant holds in the first global and local view in which any Proposal might have been constructed for a given sequence number. We then show that the invariant holds throughout the remainder of the global view. Finally, we show 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 *A.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. Then when P is constructed, Invariant A.1 holds with respect to P, and it holds for the remainder of (gv, lv).

Proof: Since P is constructed, there exists a set of at least $f+1$ correct servers which sent a partial signature on P (Figure A-10, line C7). These servers do so after collecting a Prepare Certificate(gv, lv, seq, u) binding seq to u (Figure A-10, line C3). By Lemmas A.2 and A.3, any server that collects a Prepare Certificate or a Proposal in (gv, lv) collects the same one. Since this is the first Proposal that was constructed, and a Proposal is required to globally order an update, the only Globally_Ordered_Update that can exist binds seq to u. Thus, the invariant is met when the Proposal is constructed.

According to the rules for updating the Local_History data structure, a correct server with a Prepare Certificate from (gv, lv) will not replace it and may only add a Proposal message from the same view (Figure A-10, line D3). By Lemma A.3, this Proposal is unique, and since it contains the same update and sequence number as the unique Prepare Certificate, it will not conflict with the Prepare Certificate.

A correct server with a Proposal will not replace it with any other message while in global view gv. A correct server with a Globally_Ordered_Update will never replace it. Thus, Invariant A.1 holds with respect to P for the remainder of (gv, lv). □

We now proceed to show that Invariant A.1 holds across local view changes. Before proceeding, we introduce the following terminology:

DEFINITION A.1: We say that an execution of the CONSTRUCT-LOCAL-CONSTRAINT protocol **completes** at a server within the site in a view (gv, lv) if that server successfully generates and applies a Local_Collected_Servers_State message for (gv, lv).

We first prove the following property of CONSTRUCT-LOCAL-CONSTRAINT:

Lemma *A.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 A.1 holds with respect to P at the beginning of a run of CONSTRUCT-LOCAL-CONSTRAINT, then it is never violated during the run.

Proof: During the run of CONSTRUCT-LOCAL-CONSTRAINT, a server only alters its Local_History[seq] data structure during the reconciliation phase (which occurs before sending a Local_Server_State message, Figure A-17 line B7) or when processing the resultant Local_Collected_Servers_State message. During the reconciliation phase, a correct server will only replace a Prepare Certificate with a Proposal (either independently or in a Globally_Ordered_Update), since the server and the representative are only exchanging Proposals and Globally_Ordered_Updates. Since Invariant A.1 holds at the beginning of the run, any Proposal from a later local view than the Prepare Certificate held by some correct server will not conflict with the Prepare Certificate. A server with a Globally_Ordered_Update in its Global_History data structure does not remove it. Thus, the invariant is not violated by this reconciliation.

If one or more correct servers processes the resultant Local_Collected_Servers_State message, we must show that the invariant still holds.

When a correct server processes the Local_Collected_Servers_State message (Figure A-3, block D), there are two cases to consider. First, if the message contains an entry for seq (i.e., it contains either a Prepare Certificate or a Proposal binding seq to an update), then the correct server adopts the binding. In the second case, the Local_Collected_Servers_State message does not contain an entry for seq, and the correct server clears out its Prepare Certificate for seq, if it has one. We need to show that in both cases, Invariant

A.1 is not violated.

The Local_Server_State message from at least one correct server from the set of at least $f + 1$ correct servers maintained by the invariant appears in any Local_Collected_Servers_State message, since any two sets of $2f + 1$ servers intersect on at least one correct server. We consider the contents of this server's Local_Server_State message. If this server received a Request_Local_State message with an invocation sequence number lower than seq, then the server includes its entry binding seq to u in the Local_Server_State message (Figure A-20, Block A), after bringing its Pending_Proposal_Aru up to the invocation sequence number (if necessary). Invariant A.1 guarantees that the Prepare Certificate or Proposal from this server is the latest entry for sequence number seq. Thus, the entry binding seq to u in any Local_Collected_Servers_State bundle will not be removed by the Compute_Local_Union function (Figure A-21 line B3 or B6).

If this server received a Request_Local_State message with an invocation sequence number greater than or equal to seq, then the server will not report a binding for seq, since it will obtain either a Proposal or a Globally_Ordered_Update via reconciliation before sending its Local_Server_State message. In turn, the server only applies the Local_Collected_Servers_State if the $2f + 1$ Local_Server_State messages contained therein contain the same invocation sequence number, which was greater than or equal to seq (Figure A-17, line D2). Since a correct server only sends a Local_Server_State message if its Pending_Proposal_Aru is greater than or equal to the invocation sequence number it received (Figure A-17, line B5), this implies that at least $f + 1$ correct servers have a Pending_Proposal_Aru greater than or equal to seq. The invariant ensures that all such Proposals or Globally_Ordered_Updates bind seq to u. Since only Proposals with a sequence number greater than the invocation sequence number may be removed by applying the Local_Collected_Servers_State message, and since Globally_Ordered_Update messages are never removed, applying the message will not violate Invariant A.1. □

Our next goal is to show that if Invariant A.1 holds at the beginning of a view after the view in which a Proposal has been constructed, then it holds throughout the view.

Lemma *A.6:* 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 A.1 holds with respect to P at the beginning of a view (gv, lv'), with $lv' \geq lv$, then it holds throughout the view.

Proof: To show that the invariant will not be violated during the view, we show that no server can collect a Prepare Certificate(gv, lv', seq, u'), Proposal(gv, lv', seq, u'), or Globally_Ordered_Update(gv, seq,u'), for $u \neq u'$, that would cause the invariant to be violated.

Since Invariant A.1 holds at the beginning of the view, there exists a set of at least $f + 1$ correct servers with a Prepare Certificate or a Proposal in their Local_History[seq] data structure binding seq to u, or a Globally_Ordered_Update in their Global_History[seq] data structure binding seq to u. If a conflicting Prepare Certificate is constructed, then some server collected a Pre-Prepare(gv, lv', seq, u') message and $2f$ Prepare(gv, lv', seq, Digest(u')) messages. At least $f+1$ of these messages were from correct servers. This implies that at least one correct server from the set maintained by the invariant contributed to the conflicting Prepare Certificate (either by sending a Pre-Prepare or a Prepare). This cannot occur because the server would have seen a conflict in its Local_History[seq] data structure (Figure A-8, A8) or in its Global_History[seq] data structure (Figure A-8, A18). Thus, the conflicting Prepare Certificate cannot be constructed.

Since no server can collect a conflicting Prepare Certificate, no server can construct a conflicting Proposal. Thus, by the rules of updating the Local_History data structure, a correct server only replaces its Prepare Certificate (if any) with a Prepare Certificate or Proposal from (gv, lv'), which cannot conflict. Since a Proposal is needed to construct a Globally_Ordered_Update, no conflicting Globally_Ordered_Update can be constructed, and no Globally_Ordered_Update is ever removed from the Global_History data structure. Thus, Invariant A.1 holds throughout (gv, lv'). □

We can now prove Lemma A.1:

Proof: By Lemma A.4, Invariant A.1 holds with respect to P throughout (gv, lv). By Lemma A.5, the invariant holds with respect to P during and after CONSTRUCT-LOCAL-CONSTRAINT. By Lemma A.6, the invariant holds at the beginning and end of view (gv, $lv + 1$). Repeated applications of Lemma A.5 and Lemma A.6 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 A-10, 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 A.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 $\lfloor S/2 \rfloor$ Accept corresponding Accept messages. By Lemma A.1, all Proposal messages constructed in global view gv are for the same update, which implies that all servers which globally order an update in global view gv for sequence number seq globally order the same update. □

We now prove the second main claim:

Claim *A.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 A.2 using the following lemma, 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 *A.7:* 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 A.7 using a series of lemmas. We use a strategy similar to the one used in proving Lemma A.1 above, and we maintain the following invariant:

INVARIANT A.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 A.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 $u' \neq u$.

We first show that Invariant A.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.

Lemma *A.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. Then when u is globally ordered, Invariant A.2 holds with respect to P, and it holds for the remainder of global view gv.

Proof: Since u was globally ordered in gv, some server collected a Proposal(gv, *, seq, u) message and $\lfloor S/2 \rfloor$ Accept(gv, *, seq, Digest(u)) messages. Each of the $\lfloor S/2 \rfloor$ sites that generated a threshold-signed Accept message has at least $f+1$ correct servers that contributed to the Accept, since $2f + 1$ partial signatures are required to construct the Accept and at most f are faulty. These servers store P in Global_History[seq].Proposal when they apply it (Figure A-4, block A). Since the leader site constructed P and P is threshold-signed, at least $f + 1$ correct servers in the leader site have either a Prepare Certificate corresponding to P in Global_History[seq].Prepare_Certificate or the Proposal P in Global_History[seq].Proposal. Thus, Condition 1 is met.

By Lemma A.1, all Proposals generated by the leader site for sequence number seq in gv contain the same update. Thus, no server can have a conflicting Proposal or Globally_Ordered_Update, since gv is the first view in which an update has been globally ordered for sequence number seq. Since Invariant A.1 holds in gv, no server has a conflicting Prepare Certificate from (gv, lv'), with $lv' \geq lv$. Thus, Condition 2 is met.

We now show that Condition 1 is not violated throughout the rest of global view gv. By the rules of updating the Global_History data structure in gv, a correct server with an entry in Global_History[seq].Prepare_Certificate only removes it if it generates a Proposal message from the same global view (Figure A-4, lines A7 and A14), which does not conflict with the Prepare_Certificate because it contains u, and thus it does not violate Condition 1. Similarly, a correct server in gv only replaces an entry in Global_History[seq].Proposal with a Globally_Ordered_Update. Since a Globally_Ordered_Update contains a Proposal from gv, and all Proposals from gv for sequence number seq contain u, Condition 1 is still met. No correct server ever replaces an entry in Global_History[seq].Globally_Ordered_Update. □

We now show that Invariant A.2 holds across global view changes. We start by showing that the CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT protocols, used during a global view change in the leader site and non-leader sites, respectively, will not cause the invariant to be violated. We then show that if any correct server in the leader site becomes globally constrained by completing the global view change protocol, the invariant will still hold after applying the Collected_Global_Constraints message to its data structure.

Lemma *A.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. Assume Invariant A.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 begins at S, the invariant is never violated during the run.

Proof: During a run of CONSTRUCT-ARU, a correct server only modifies its Global_History[seq] data structure in three cases. We show that, in each case, Invariant A.2 will not be violated if it is already met.

The first case occurs during the reconciliation phase of the protocol. In this phase, a correct server with either a Prepare Certificate or Proposal in Global_History[seq] may replace it with a Globally_Ordered_Update, since the server and the representative only exchange Globally_Ordered_Update messages. Since Invariant A.2 holds at the beginning of the run, no server has a Globally_Ordered_Update from any view $gv' \geq gv$ that conflicts with the binding of seq to u. Since u could only have been globally ordered in a global view $gv' \geq gv$, no conflicting Globally_Ordered_Update exists from a previous global view. Thus, Invariant A.2 is not violated during the reconciliation phase.

In the second case, a correct server with a Prepare Certificate in Global_History[seq] tries to construct corresponding Proposals (replacing the Prepare Certificate) by invoking THRESHOLD-SIGN (Figure A-18, line D6). Since the Proposal is for the same binding as the Prepare Certificate, the invariant is not violated.

In the third case, a correct server applies any Globally_Ordered_Updates appearing in the Global_Constraint message to its Global_History data structure (Figure A-18, line G2). Since Invariant A.2 holds at the beginning of the run, no Globally_Ordered_Update exists from any view $gv' \geq gv$ that conflicts with the binding of seq to u. Since u could only have been globally ordered in a global view $gv' \geq gv$, no conflicting Globally_Ordered_Update exists from a previous global view.

Since these are the only cases in which Global_History[seq] is modified during the protocol, the invariant holds throughout the run. □

Lemma *A.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 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 A.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-GLOBAL-CONSTRAINT begins at S, the invariant is never violated during the run.

Proof: During a run of CONSTRUCT-GLOBAL-CONSTRAINT, a correct server only modifies its Global_History[seq] data structure when trying to construct Proposals corresponding to any Prepare Certificates appearing in the union (Figure A-19, line C5). Since the Proposal resulting from THRESHOLD-SIGN is for the same binding as the Prepare Certificate, the invariant is not violated. □

We now show that if Invariant A.2 holds at the beginning of a run of the GLOBAL-VIEW-CHANGE protocol after the global view in which an update was globally ordered, then the invariant is never violated during the run.

Lemma *A.11:* 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 A.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.

Proof: During a run of GLOBAL-VIEW-CHANGE, a correct server may only modify its Global_History[seq] data structure in three cases. The first occurs in the leader site, during a run of CONSTRUCT-ARU (Figure A-16, line A2). By Lemma A.9, Invariant A.2 is not violated during this protocol. The second case occurs at the non-leader sites, during a run of CONSTRUCT-GLOBAL-CONSTRAINT (Figure A-16, line C4). By Lemma A.10, Invariant A.2 is not violated during this protocol.

The final case occurs at the leader site when a correct server becomes globally constrained by applying a Collected_Global_Constraints message to its Global_History data structure (Figure A-16, lines E5 and F2). We must now show that Invariant A.2 is not violated in this case.

Any Collected_Global_Constraints message received by a correct server contains a Global_Constraint message from at least one site maintained by Invariant A.2, since any two majorities intersect on at least one site. We consider the Global_Constraint message sent by this site, S. The same logic will apply when Global_Constraint messages from more than one site in the set maintained by the invariant appear in the Collected_Global_Constraints message.

We first consider the case where S is a non-leader site. There are two sub-cases to consider.

Case 1a: In the first sub-case, the Aru_Message generated by the leader site in CONSTRUCT-ARU contains a sequence number less than seq. In this case, each of the $f + 1$ correct servers in S maintained by Invariant A.2 reports a Proposal message binding seq to u in its Global_Server_State message (Figure A-20, Block B). At least one such message will appear in the Global_Collected_Servers_State bundle, since any two sets of $2f+1$ intersect on at least one correct server. Invariant A.2 maintains that the entry binding seq to u is the latest, and thus it will not be removed by the Compute_Global_Union procedure (Figure A-21, Blocks C and D). The resultant Global_Constraint message therefore binds seq to u. Invariant A.2 also guarantees that this entry or one with the same binding will be the latest among those contained in the Collected_Global_Constraints message, and thus it will not be removed by the Compute_Constraint_Union function run when applying the message to Global_History (Figure A-21, Blocks E and F) By the rules of applying the Collected_Global_Constraints message (Figure A-4, Block D), the binding of seq to u will be adopted by the correct servers in the leader site that become globally constrained, and thus Invariant A.2 is not violated.

Case 1b: In the second sub-case, the Aru_Message generated by the leader site in CONSTRUCT-ARU contains a sequence number greater than or equal to seq. In this case, no entry binding seq to u will be reported in the Global_Constraint message. In this case, we show that at least $f+1$ correct servers in the leader site have already globally ordered seq. The invariant guarantees that those servers which have already globally ordered an update for seq have globally ordered u. To construct the Aru_Message, at least $f+1$ correct servers contributed partial signatures to the result of calling Extract_Aru (Figure A-18, line G3) on the union derived from the Global_Collected_Servers_State bundle. Thus, at least $f+1$ correct servers accepted the Global_Collected_Servers_State message as valid, and, at Figure A-18, line D3, enforced that their Global_aru was at least as high as the invocation sequence number (which was greater than or equal to seq). Thus, these servers have Globally_Ordered_Update messages for seq, and the invariant holds in this case.

We must now consider the case where S is the leader site. As before, there are two sub-cases to consider. We must show that Invariant A.2 is not violated in each case. During CONSTRUCT-ARU, the Global_Server_State message from at least one correct server from the set of at least $f+1$ correct servers maintained by the invariant appears in any Collected_Global_Servers_State message, since any two sets of $2f+1$ servers intersect on at least one correct server. We consider the contents of this server's Global_Server_State message.

Case 2a: In the first sub-case, if this server received a Request_Global_State message with an invocation sequence number lower than seq, then the server includes its entry binding seq to u in the Global_Server_State message, after bringing its Global_Aru up to the invocation sequence number (if necessary) (Figure A-18, lines B5 and B7). Invariant A.2 guarantees that the Prepare Certificate, Proposal, or Globally_Ordered_Update binding seq to u is the latest entry for sequence number seq. Thus, the entry binding seq to u in any Global_Collected_Servers_State bundle will not be removed by the Compute_Global_Union function (Figure A-21, Blocks C and D) and will appear in the resultant Global_Constraint message. Thus, the Collected_Global_Constraints message will bind seq to u, and by the rules of applying this message to the Global_History[seq] data structure, Invariant A.2 is not violated when the correct servers in the leader site become globally constrained by applying the mesasge (Figure A-4, block D).

Case 2b: If this server received a Request_Global_State message with an invocation sequence number greater than or equal to seq, then the server will not report a binding for seq, since it will obtain a Globally_Ordered_Update via reconciliation before sending its Global_Server_State message (Figure A-18, lines B4). In turn, the server only contributes a partial signature on the Aru_Message if it received a valid Global_Collected_Servers_State message, which implies that the $2f+1$ Global_Server_State messages in the Global_Collected_Servers_State bundle contained the same invocation sequence number, which was greater than or equal to seq (Figure A-18, line D2). Since a correct server only sends a Global_Server_State message if its Global_Aru is greater than or equal to the invocation sequence number it received (Figure A-18, line D3), this implies that at least $f+1$ correct servers have a Global_Aru greater than or equal to seq. The invariant ensures that all such Globally_Ordered_Updates bind seq to u. Thus, even if the Collected_Global_Constraints message does not contain an entry binding seq to u, the leader site and $\lfloor S/2 \rfloor$ non-leader sites will maintain Invariant A.2. □

Corollary *A.12:* 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 A.2 holds with respect to P at the beginning of a run of the GLOBAL-VIEW-CHANGE protocol, then 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 A.2.

Proof: We consider each of the four sub-cases described in Lemma A.11. In Cases 1a and 2a, any correct server that becomes globally constrained binds seq to u. In Cases 1b and 2b, there exists a set of at least $f+1$ correct servers that have globally ordered u for sequence number seq. Thus, in all four cases, if at least $f+1$ correct servers become globally constrained, the leader site meets the data structure condition of of Condition 1 of Invariant A.2. □

Our next goal is to show that if Invariant A.2 holds at the beginning of a global view after which an update has been globally ordered, then it holds throughout the view.

Lemma *A.13:* 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 A.2 holds with respect to P at the beginning of a global view (gv', *), with $gv' > gv$, then it holds throughout the view.

Proof: To show that the invariant will not be violated during global view gv', we show that no conflicting Prepare Certificate, Proposal, or Globally_Ordered_Update can be constructed during the view 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 Corollary A.12, since at least $f + 1$ correct servers became globally constrained in gv', the leader site meets Condition 1 of Invariant A.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 server from the set of at least $f+1$ correct servers binding seq to u 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 u (Figure A-12, line A3 or A5). Similarly, a correct server would see a conflict (Figure A-8, line A10 or A13).

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.

Thus, Invariant A.2 holds throughout global view gv'. □

We can now prove Lemma A.7:

Proof: By Lemma A.8, Invariant A.2 holds with respect to P1 throughout global view gv. By Lemma A.11, the invariant holds with respect to P1 during and after the GLOBAL-VIEW-CHANGE protocol. By Lemma A.13, the invariant holds at the beginning and end of global view $gv+1$. Repeated application of Lemma A.11 and Lemma A.13 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 A-10, 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 A.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 A.7, 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. □

We can now prove SAFETY - S1.

Proof: By Claims A.1 and A.2, if two servers globally order an update for the same sequence number in any two global views, then they globally order the same update. Thus, if two servers execute an update for any sequence number, they execute the same update, completing the proof. □

We now prove that Steward meets the following validity property:

S2 - VALIDITY Only an update that was proposed by a client may be executed.

Proof: A server executes an update when it has been globally ordered. To globally order an update, a server obtains a Proposal and $\lfloor S/2 \rfloor$ corresponding Accept messages. To construct a Proposal, at least $f + 1$ correct servers collect a Prepare Certificate and invoke THRESHOLD-SIGN. To collect a Prepare Certificate, at least $f + 1$ correct servers must have sent either a Pre-Prepare or a Prepare in response to a Pre-Prepare. From the validity check run on each incoming message (Figure A-6, lines A7 - A9), a Pre-Prepare message is only processed if the update contained within has a valid client signature. Since we assume that client signatures cannot be forged, only a valid update, proposed by a client, may be globally ordered. □

B.2 Liveness Proof

We now prove that Steward meets the following liveness property:

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.

Proof Strategy: 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, a contradiction. We prove Global Liveness using three main claims. In the first claim, we show that if no global progress occurs, then all stable servers eventually reconcile their Global_History data structures to a common point. Specifically, the stable servers set their Global_aru variables 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. In the second claim, we show that, once this reconciliation has occurred, 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, which is a precondition for globally ordering an update that would cause progress to occur. To prove the second claim, we first prove three subclaims. The first two subclaims 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. The third subclaim establishes relationships between the global and local timeouts, which we use to show that the stable servers will eventually remain in their views long enough for global progress to be made. Finally, in the third claim, 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, which contradicts our assumption.

In the claims and proofs that follow, we assume that the system has already reached a stabilization time, T, at which the system became stable. Since we assume that no global progress occurs, we use the following definition:

DEFINITION B.1: We say that a sequence number is the **max_stable_seq** if, assuming no further global progress is made, it is the last sequence number for which any stable server has executed an update.

We now proceed to prove the first main claim:

Claim *B.1:* If no global progress occurs, then all stable servers in all stable sites eventually set their Global_aru variables to max_stable_seq.

To prove Claim B.1, we first prove two lemmas relating to LOCAL-RECONCILIATION and GLOBAL-RECONCILIATION.

Lemma *B.1:* Let aru be the Global_aru of some stable server, s, in stable Site S at time T. Then all stable servers in S eventually have a Global_aru of at least aru.

Proof: The stable servers in S run LOCAL-RECONCILIATION by sending a Local_Recon_Request message every LOCAL-RECON-THROTTLE-PERIOD time units (Figure A-22, line A1). Since S is stable, s will receive a Local_Recon_Request message from each stable server within one local message delay. If the requesting server, r, has a Global_aru less than aru, s will send to r Globally_Ordered_Update messages for each sequence number in the difference. These messages will arrive in bounded time. Thus, each stable server in S sets it Global_aru to at least aru. □

Lemma *B.2:* Let S be a stable site in which all stable servers have a Global_aru of at least aru at time T. Then if no global progress occurs, at least one stable server in all stable sites eventually has a Global_aru of at least aru.

Proof: Since no global progress occurs, there exists some sequence number aru', for each stable site, R, that is the last sequence number for which a stable server in R globally ordered an update. By Lemma B.1, all stable servers in R eventually reach aru' via the LOCAL-RECONCILIATION protocol.

The stable servers in R run GLOBAL-RECONCILIATION by sending a Global_Recon_Request message every GLOBAL-RECON-THROTTLE-PERIOD time units (Figure A-23, line A1). Since R is stable, each stable server in R receives the request of all other stable servers in R within a local message delay. Upon receiving a request, a stable server will send a Partial_Sig message to the requester, since they have the same Global_aru, aru'. Each stable server can thus construct a threshold-signed GLOBAL-RECON message containing aru'. Since there are $2f+1$ stable servers, the pigeonhole principle guarantees that at least one of them sends a GLOBAL-RECON message to a stable peer in each other stable site. The message arrives in one wide area message delay.

If all stable sites send a GLOBAL-RECON message containing a requested_aru value of at least aru, then the lemma holds, since at least $f+1$ correct servers contributed a Partial_sig on such a message, and at least one of them is stable. If there exists any stable site R that sends a GLOBAL-RECON message with a requested_aru value lower than aru, we must show that R will eventually have at least one stable server with a Global_aru of at least aru.

Each stable server in S has a Global_aru of aru', with $aru' \geq aru$. Upon receiving the GLOBAL-RECON message from R, a stable server uses the THROTTLE-SEND procedure to send all Globally_Ordered_Update messages in the difference to the requester (Figure A-

23, line D16). Since the system is stable, each Globally_Ordered_Update will arrive at the requester in bounded time, and the requester will increase its Global_aru to at least aru. □

We now prove Claim B.1:

Proof: Assume, without loss of generality, that stable site S has a stable server with a Global_aru of max_stable_seq. By Lemma B.1, all stable servers in S eventually set their Global_aru to at least max_stable_seq. Since no stable server sets its Global_aru beyond this sequence number (by the definition of max_stable_seq), the stable servers in S set their Global_aru to exactly max_stable_seq. By Lemma B.2, at least one stable server in each stable site eventually sets its Global_aru to at least max_stable_seq. Using similar logic as above, these stable servers set their Global_aru variables to exactly max_stable_seq. By applying Lemma B.1 in each stable site and using the same logic as above, all stable servers in all stable sites eventually set their Global_aru to max_stable_seq. □

We now proceed to prove the second main claim, which shows that, once the above reconciliation has taken place, the system will reach a state in which a stable representative of a stable leader site can complete the GLOBAL-VIEW-CHANGE protocol, which is a precondition for globally ordering a new update. This notion is encapsulated in the following claim:

Claim *B.2:* If no global progress occurs, and the system is stable with respect to time T, then there exists an infinite set of global views gv_i, each with stable leader site S_i, in which the first stable representative in S_i serving for at least a local timeout period can complete GLOBAL-VIEW-CHANGE.

Since completing GLOBAL-VIEW-CHANGE requires all stable servers to be in the same global view for some amount of time, we begin by proving several claims about the GLOBAL-LEADER-ELECTION protocol. Before proceeding, we prove the following claim relating to the THRESHOLD-SIGN protocol, which is used by GLOBAL-LEADER-ELECTION:

Claim *B.3:* If all stable servers in a stable site invoke THRESHOLD-SIGN on the same message, m, then THRESHOLD-SIGN returns a correctly threshold-signed message m at all stable servers in the site within some finite time, Δ_{sign}.

To prove Claim B.3, we use the following lemma:

Lemma *B.3:* If all stable servers in a stable site invoke THRESHOLD-SIGN on the same message, m, then all stable servers will receive at least $2f + 1$ correct partial signature shares for m within a bounded time.

Proof: When a correct server invokes THRESHOLD-SIGN on a message, m, it generates a partial signature for m and sends this to all servers in its site (Figure A-9, Block A). A correct server uses only its threshold key share and a deterministic algorithm to generate a partial signature on m. The algorithm is guaranteed to complete in a bounded time. Since the site is stable, there are at least $2f + 1$ correct servers that are connected to each other in the site. Therefore, if the stable servers invoke THRESHOLD-SIGN on m, then each stable server will receive at least $2f + 1$ partial signatures on m from correct servers. □

We can now prove Claim B.3.

Proof: A correct server combines $2f+1$ correct partial signatures to generate a threshold signature on m. From Lemma B.3, a correct server will receive $2f + 1$ correct partial signatures on m.

We now need to show that a correct server will eventually combine the correct signature shares. Malicious servers can contribute an incorrect signature share. If the correct server combines a set of $2f + 1$ signature shares, and one or more of the signature shares are incorrect, the resulting threshold signature is also incorrect.

When a correct server receives a set of $2f+1$ signature shares, it will combine this set and test to see if the resulting signature verifies (Figure A-9, Block B). If the signature verifies, the server will return message m with a correct threshold signature (line B4). If the signature does not verify, then THRESHOLD-SIGN does not return message m with a threshold signature. On lines B6-B11, the correct server checks each partial signature that it has received from other servers. If any partial signature does not verify, it removes the incorrect partial signature from its data structure and adds the server that sent the partial signature to a list of corrupted servers. A correct server will drop any message sent by a server in the corrupted server list (Figure A-6, lines A10-A11). Since there are at most f malicious servers in the site, these servers can prevent a correct server from correctly combining the $2f + 1$ correct partial signatures on m at most f times. Therefore, after a maximum of f verification failures on line B3, there will be a verification success and THRESHOLD-SIGN will return a correctly threshold signed message m at all correct servers, proving the claim. □

We now can prove claims about GLOBAL-LEADER-ELECTION. We first introduce the following terminology used in the proof:

DEFINITION B.2: We say that a server **preinstalls** global view gv when it collects a set of Global_VC(gv_i) messages from a majority of sites, where $gv_i \geq gv$.

DEFINITION B.3: A **global preinstall proof** for global view gv is a set of Global_VC(gv_i) messages from a majority of sites where $gv_i \geq gv$. The set of messages is proof that gv preinstalled.

Our goal is to prove the following claim:

Claim *B.4:* If global progress does not occur, and the system is stable with respect to time T, then all stable servers will preinstall the same global view, gv, in a finite time. Subsequently, all stable servers will: (1) preinstall all consecutive global views above gv within one wide area message delay of each other and (2) remain in each global view for at least one global timeout period.

To prove Claim B.4, we maintain the following invariant and show that it always holds:

INVARIANT B.1: If a correct server, s, has Global_view gv, then it is in one of the two following states:

1) Global_T is running and s has global preinstall proof for gv.
2) Global_T is not running and s has global preinstall proof for $gv - 1$.

Lemma *B.4:* Invariant B.1 always holds.

Proof: We show that Invariant B.1 holds using an argument based on a state machine, SM. SM has the two states listed in Invariant B.1.

We first show that a correct server starts in state (1). When a correct server starts, its Global_view is initialized to 0, it has an *a priori* global preinstall proof for 0, and its Global_T timer is running. Therefore, Invariant B.1 holds immediately after the system is initialized, and the server is in state (1).

We now show that a correct server can only transition between these two states. SM has the following two types of state transitions. These transitions are the only events where (1) the state of Global_T can change (from running to stopped or from stopped to running), (2) the value of Global_T changes, or (3) the value of global preinstall proof changes. In our pseudocode, the state transitions occur across multiple lines and functions. However, they are atomic events that always occur together, and we treat them as such.

- Transition (1): A server can transition from state (1) to state (2) only when Global_T expires and it increments its global view by one.
- Transition (2): A server can transition from state (2) to state (1) or from state (1) to state (1) when it increases its global preinstall proof and starts Global_T.

We now show that if Invariant B.1 holds before a state transition, it will hold after a state transition.

We first consider transition (1). We assume that Invariant B.1 holds immediately before the transition. Before transition (1), SM is in state (1) and Global_view is equal to Global_preinstalled_view, and Global_T is running. After transition (1), SM is in state (2) and Global_view is equal to Global_preinstalled_view + 1, and Global_T is stopped. Therefore, after the state transition, Invariant B.1 holds. This transition corresponds to Figure A-14, lines A1 and A2. On line A1, Global_T expires and stops. On line A2, Global_view is incremented by one. SM cannot transition back to state (1) until a transition (2) occurs.

We next consider transition (2). We assume that Invariant B.1 holds immediately before the transition. Before transition (2) SM can be in either state (1) or state (2). We now prove that the invariant holds immediately after transition (2) if it occurs from either state (1) or state (2).

Let gv be the value of Global_view before the transition. If SM is in state (1) before transition (2), then global preinstall proof is gv, and Global_T is running. If SM is in state (2) before transition (2), then global preinstall proof is $gv - 1$, and Global_T is stopped. In either case, the following is true before the transition: global preinstalled proof $\geq gv - 1$. Transition (2) occurs only when global preinstall proof increases (Figure A-14, block E). Line E6 of Figure A-14 is the only line in the pseudocode where Global_T is started after initialization, and this line is triggered upon increasing global preinstall proof. Let global preinstall proof equal gp after transition (2) and Global_view be gv'. Since the global preinstall proof must be greater than what it was before the transition, $gp \geq gv$. On lines E5 - E7 of Figure A-4, when global preinstall proof is increased, Global_view is increased to global preinstall proof if Global_view < global preinstall proof. Thus, $gv' \geq gp$. Finally, $gv' \geq gv$, because Global_view either remained the same or increase.

We now must examine two different cases. First, when $gv' > gv$, the Global_view was increased to gp, and, therefore, $gv' = gp$. Second, when $gv' = gv$ (i.e., Global_view was not increased), then, from $gp \geq gv$ and $gv' \geq gp$, $gv' = gp$. In either case, therefore, Invariant B.1 holds after transition (2).

We have shown that Invariant B.1 holds when a server starts and that it holds after each state transition. □

We now prove a claim about RELIABLE-SEND-TO-ALL-SITES that we use to prove Claim B.4:

Claim *B.5:* If the system is stable with respect to time T, then if a stable server invokes RELIABLE-SEND-TO-ALL-SITES on message m, then all stable servers will receive m.

Proof: When a stable server invokes RELIABLE-SEND-TO-ALL-SITES on message m, it first creates a Reliable_Message(m) message and sends it to all of the servers in its site, S, (Figure A-24, lines A2 and A3). Therefore, all stable servers in S will receive message m embedded within the Reliable_Message.

The server that invoked RELIABLE-SEND-TO-ALL-SITES calls SendToPeers on m (line A4). All other servers call SendToPeers(m) when they receive Reliable_Message(m) (line B2). Therefore, all stable servers in S will call SendToPeers(m). This function first checks to see if the server that called it has a Server_id between 1 and $2f+1$ (line D1). Recall that servers in each site are uniquely numbered with integers from 1 to $3f+1$. If a server is one of the $2f+1$ servers with the lowest values, it will send its message to all servers in all other sites that have a Server_id equal to its server id (lines D2-D4).

Therefore, if we consider S and any other stable site S', then message m is sent across $2f+1$ links, where the $4f+2$ servers serving as endpoints on these links are unique. A link passes m from site S to S' if both endpoints are stable servers. There are at most $2f$ servers that are not stable in the two sites. Therefore, if each of these non-stable servers blocks one link, there is still one link with stable servers at both endpoints. Thus, message m will pass from S to at least one stable server in all other sites. When a server on the receiving endpoint receives m (lines C1-C2), it sends m to all servers in its site. Therefore, we have proved that if any stable server in a stable system invokes RELIABLE-SEND-TO-ALL-SITES on m, all stable servers in all stable sites will receive m. □

We now show that if all stable servers increase their Global_view to gv, then all stable servers will preinstall global view gv.

Lemma *B.5:* If the system is stable with respect to time T, then if, at a time after T, all stable servers increase their Global_view variables to gv, all stable servers will preinstall global view gv.

Proof: We first show that if any stable server increases its global view to gv because it receives global preinstall proof for gv, then all stable servers will preinstall gv. When a stable server increases its global preinstall proof to gv, it reliably sends this proof to all servers (Figure A-14, lines E4 and E5) By Claim B.5, all stable servers receive this proof, apply it, and preinstall global view gv.

We now show that if all stable servers increase their global views to gv without first receiving global preinstall proof for gv, all stable servers will preinstall gv. A correct server can increase its Global_view to gv without having preinstall proof for gv in only one place in the pseudocode (Figure A-14, line A2). If a stable server executes this line, then it also constructs an unsigned Global_VC(gv) message and invokes THRESHOLD-SIGN on this message (lines A4-A5).

From Claim B.3, if all stable servers in a stable site invoke THRESHOLD-SIGN on Global_VC(gv), then a correctly threshold signed Global_VC(gv) message will be returned to all stable servers in this site. When THRESHOLD-SIGN returns a Global_VC message to a stable server, this server reliably sends it to all other sites. By Claim B.5, all stable servers will receive the Global_VC(gv) message. Since we assume all stable servers in all sites increase their global views to gv, all stable servers will receive a Global_VC(gv) message from a majority of sites. □

We next prove that soon after the system becomes stable, all stable servers preinstall the same global view gv. We also show that there can be no global preinstall proof for a global view above gv:

Lemma *B.6:* If global progress does not occur, and the system is stable with respect to time T, then all stable servers will preinstall the same global view gv before time $T + \Delta$, where gv is equal to the the maximum global preinstall proof in the system when the stable servers first preinstall gv.

Proof: Let s_{max} be the stable server with the highest preinstalled global view, gp_{max}, at time T, and let $gpsys_{max}$ be the highest preinstalled view in the system at time T. We first show that $gp_{max} + 1 \geq gpsys_{max}$. Second, we show that all stable servers will preinstall gp_{max}. Then we show that the Global_T timers will expire at all stable servers, and they will increase their global view to $gp_{max} + 1$. Next, we show that when all stable servers move to global view $gp_{max} + 1$, each site will create a threshold signed Global_VC($gp_{max}+1$) message, and all stables servers will receive enough Global_VC messages to preinstall $gp_{max} + 1$.

In order for $gpsys_{max}$ to have been preinstalled, some server in the system must have collected Global_VC($gpsys_{max}$) messages from a majority of sites. Therefore, at least $f + 1$ stable servers must have had global views for $gpsys_{max}$, because they must have invoked THRESHOLD-SIGN on Global_VC($gpsys_{max}$). From Invariant B.1, if a correct server is in $gpsys_{max}$, it must have global preinstall proof for at least $gpsys_{max} - 1$. Therefore, $gp_{max} + 1 \geq gpsys_{max}$.

When s_{max} preinstalls gp_{max}, it reliably sends global preinstall proof for gp_{max} to all stable sites (via the RELIABLE-SEND-TO-ALL-SITES protocol). By Claim B.5, all stable servers will receive and apply Global_Preinstall_Proof(gp_{max}) and increase their Global_view variables to gp_{max}. Therefore, within approximately one widea-area message delay of T, all stable servers will preinstall gp_{max}. By Invariant B.1, all stable servers must have global view gp_{max} or $gp_{max}+1$. Any stable server with Global_view $gp_{max}+1$ did not yet preinstall this global view. Therefore, its timer is stopped as described in the proof of Lemma B.4, and it will not increase its view again until it receives proof for a view higher than gp_{max}.

We now need to show that all stable servers with Global_view gp_{max} will move to Global_view $gp_{max}+1$. All of the servers in gp_{max} have running timers because their global preinstall proof = Global_view. The

Global_T timer is reset in only two places in the pseudocode. The first is on line E6 of Figure A-14. This code is not called unless a server increases its global preinstall proof, in which case it would also increase its Global_view to $gp_{max}+1$. The second case occurs when a server executes a Globally_Ordered_Update (Figure A-4, line C8), which cannot happen because we assume that global progress does not occur. Therefore, if a stable server that has view gp_{max} does not increase its view because it receives preinstall proof for $gp_{max}+1$, its Global_T timer will expire and it will increment its global view to $gp_{max}+1$.

We have shown that if global progress does not occur, and the system is stable with respect to time T, then all stable servers will move to the same global view, $gp_{max}+1$. A server either moves to this view because it has preinstall proof for $gp_{max}+1$ or it increments its global view to $gp_{max}+1$. If any server has preinstall proof for gp_{max}, it sends this proof to all stable servers using RELIABLE-SEND-TO-ALL-SITES and all stable servers will preinstall $gp_{max}+1$. By Lemma B.5, if none of the stable servers have preinstall proof for $gp_{max}+1$ and they have incremented their global view to $gp_{max}+1$, then all stable servers will preinstall $gp_{max}+1$.

We conclude by showing that time Δ is finite. As soon as the system becomes stable, the server with the highest global preinstall proof, gp_{max}, sends this proof to all stable servers as described above. It reaches them in one wide area message delay. After at most one global timeout, the stable servers will increment their global views because their Global_T timeout will expire. At this point, the stable servers will invoke THRESHOLD-SIGN, Global_VC messages will be returned at each stable site, and the stable servers in each site will reliably send their Global_VC messages to all stable servers. These messages will arrive in approximately one wide area delay, and all servers will install the same view, $gp_{max}+1$. □

We now prove the last lemma necessary to prove Claim B.4:

Lemma *B.7:* If the system is stable with respect to time T, then if all stable servers are in global view gv, the Global_T timers of at least $f+1$ stable servers must timeout before the global preinstall proof for $gv+1$ can be generated.

Proof: A stable system has a majority of sites each with at least $2f+1$ stable servers. If all of the servers in all non-stable sites generate Global_VC($gv+1$) messages, the set of existing messages does not constitute global preinstall proof for $gv+1$. One of the stable sites must contribute a Global_VC($gv+1$) message. In order for this to occur, $2f+1$ servers at one of the stable sites must invoke THRESHOLD-SIGN on Global_VC($gv+1$), which implies $f+1$ stable servers had global view $gv+1$. Since global preinstall proof could not have been generated without the Global_VC message from their site, Global_T at these servers must have expired. □

We now use Lemmas B.5, B.6, and B.7 to prove Claim B.4:

Proof: By Lemma B.6, all servers will preinstall the same view, gv, and the highest global preinstall proof in the system is gv. If global progress does not occur, then the Global_T timer at all stable servers will eventually expire. When this occurs, all stable servers will increase their global view to $gv+1$. By Lemma B.5, all stable servers will preinstall $gv+1$. By Lemma B.5, Global_T must have expired at at least $f+1$ stable servers. We have shown that if all stable servers are in the same global view, they will remain in this view until at least $f+1$ stable servers Global_T timer expires, and they will definitely preinstall the next view when all stable servers' Global_T timer expires.

When the first stable server preinstalls global view $gv+1$, it reliably sends global preinstall proof $gv+1$ to all stable servers (Figure A-14, line E4). Therefore, all stable servers will receive global preinstall proof for $gv+1$ at approximately the same time (within approximately one wide area message delay). The stable servers will reset their Global_T timers and start them when they preinstall. At this point, no server can preinstall the next global view until there is a global timeout at at least $f+1$ stable servers. If the servers don't preinstall the next global view before, they will do so when there is a global timeout at all stable servers. Then the process repeats. The stable servers preinstall all consecutive global views and remain in them for a global timeout period. □

We now prove a similar claim about the local representative election protocol. The protocol is embedded within the LOCAL-VIEW-CHANGE protocol, and it is responsible for the way in which stable servers within a site synchronize their Local_view variable.

Claim *B.6:* If global progress does not occur, and the system is stable with respect to time T, then all stable servers in a stable site will preinstall the same local view, lv, in a finite time. Subsequently, all stable servers in the site will: (1) preinstall all consecutive local views above lv within one local area message delay of each other and (2) remain in each local view for at least one local timeout period.

To prove Claim B.6, we use a state machine based argument to show that the following invariant holds:

INVARIANT B.2: If a correct server, s, has Local_view lv, then it is in one of the following two states:

1) Local_T is running and s has local preinstall proof lv
2) Local_T is not running and s has local preinstall proof $lv-1$.

Lemma *B.8:* Invariant B.2 always holds.

Proof: When a correct server starts, Local_T is started, Local_view is set to 0, and the server has an *a priori* proof (New_Rep message) for local view 0. Therefore, it is in state (1).

A server can transition from one state to another only in the following two cases. These transitions are the only times where a server (1) increases its local preinstall proof, (2) increases its Local_view, or (3) starts or stops Local_T.

- Transition (1): A server can transition from state (1) to state (2) only when Local_T expires and it increments its local view by one.
- Transition (2): A server can transition from state (2) to state (1) or from state (1) to state (1) when it increases its local preinstall proof and starts Local_T.

We now show that if Invariant B.2 holds before a state transition, it will hold after a state transition.

We first consider transition (1). We assume that Invariant B.2 holds immediately before the transition. Before transition (1), the server is in state (1) and Local_view is equal to local preinstalled view, and Local_T is running. After transition (1), the server is in state (2) and Local_view is equal to local preinstalled view + 1, and Local_T is stopped. Therefore, after the state transition, Invariant B.2 holds. This transition corresponds to lines A1 and A2 in Figure A-13. On line A1, Local_T expires and stops. On line A2, Local_view is incremented by one. The server cannot transition back to state (1) until there is a transition (2).

We next consider transition (2). We assume that Invariant B.2 holds immediately before the transition. Before transition (2) the server can be in either state (1) or state (2). We now prove that the invariant holds immediately after transition (2) if it occurs from either state (1) or state (2).

Let lv be the value of Local_view before transition. If the server is in state (1) before transition (2), then local preinstall proof is lv, and Local_T is running. If the server is in state (2) before transition (2), then local preinstall proof is $lv-1$, and Local_T is stopped. In either case, the following is true before the transition: local preinstall proof $\geq gv-1$. Transition (2) occurs only when local preinstall proof increases (Figure A-13, block D). Line D4 of the LOCAL-VIEW-CHANGE protocol is the only line in the pseudocode where Local_T is started after initialization, and this line is triggered only upon increasing local preinstall proof. Let local preinstall proof equal lp after transition (2) and Local_view be lv'. Since the local preinstall proof must be greater than what it was before the transition, $lp \geq lv$. On lines E2-E4 of Figure A-3, when local preinstall proof is increased, Local_view is increased to local preinstall proof if Local_view < local preinstall proof. Thus, $lv' \geq lp$. Finally, $lv' \geq lv$, because Local_view either remained the same or increased.

We now must examine two different cases. First, when $lv' > lv$, Local_view was increased to lp, and, therefore, $lv' = lp$. Second, when $lv' = lv$ (i.e., Local_view was not increased), then, from $lp \geq lv$ and $lv' \geq lp$ and simple substituition, $lv' = lp'$. In either case, therefore, Invariant B.2 holds after transition (2).

We have shown that Invariant B.2 holds when a server starts and that it holds after each state transition, completing the proof. □

We can now prove Claim B.6.

Proof: Let s_{max} be the stable server with the highest local preinstalled view, lp_{max}, in stable site S. Let lv_{max} be server s_{max}'s local view. The local preinstall proof is a New_Rep(lp_{max}) message threshold signed by site S. Server s_{max} sends its local preinstall proof to all other servers in site S when it increases its local preinstall proof (Figure A-13, line D3). Therefore, all stable servers in site S will receive the New_Rep message and preinstall lp_{max}.

From Invariant B.2, $lp_{max} = lv_{max}-1$ or $lp_{max} = lv_{max}$. Therefore, all stable servers are within one local view of each other. If $lp_{max} = lv_{max}$, then all servers have the same local view and their Local_T timers are running. If not, then there are two cases we must consider.

1) Local_T will expire at the servers with local view lp_{max} and they will increment their local view to lv_{max} (Figure A-13, line D3). Therefore, all stable servers will increment their local views to lv_{max}, and invoke THRESHOLD-SIGN on New_Rep(lv_{max}) (Figure A-13, line A5). By Claim B.3, a correctly threshold signed New_Rep(lv_{max}) message will be returned to all stable servers. They will increase their local preinstall proof to lv_{max}, send the New_Rep message to all other servers, and start their Local_T timers.
2) The servers with local view lp_{max} will receive a local preinstall proof higher than lp_{max}. In this case, the servers increase their local view to the value of the preinstall proof they received, send the preinstall proof, and start their Local_T timers.

We have shown that, in all cases, all stable servers will preinstall the same local view and that their local timers will be running. Now, we need to show that these stable servers will remain in the same local view for one local timeout, and then all preinstall the next local view.

At least $2f + 1$ servers must first be in a local view before a New_Rep message will be created for that view. Therefore, the f malicious servers cannot create a preinstall proof by themselves. When any stable server increases its local preinstall proof to the highest in the system, it will send this proof to all other stable servers. These servers will adopt this preinstall proof and start their timers. Thus, all of their Local_T timers will start at approximately the same time. At least $f + 1$ stable servers must timeout before a higher preinstall proof can be created. Therefore, the stable servers will stay in the same local view for a local timeout period. Since all stable servers start Local_T at about the same time (within a local area message delay), they will all timeout at about the same time. At that time, they all invoke THRESHOLD-SIGN and a New_Rep message will be created for the next view. At this point, the first server to increase its preinstall proof sends this proof to all stable servers. They start their Local_T timers, and the process repeats. Each consecutive local view is guaranteed to preinstall, and the stable servers will remain in the same view for a local timeout. □

We now establish relationships between our timeouts. Each server has two timers, Global_T and Local_T, and a corresponding global and local timeout period for each timer. The servers in the leader site have a longer local timeout than the servers in the non-leader site so that a correct representative in the leader site can communicate with at least one correct representative in all stable non-leader sites. The following claim specifies the values of the timeouts relative to each other.

Claim *B.7:* All correct servers with the same global view, gv, have the following timeouts:

1) The local timeout at servers in the non-leader sites is $local\_to\_nls$
2) The local timeout at the servers in the leader site is $local\_to\_ls = (f+2)local\_to\_nls$
3) The global timeout is $global\_to = (f+3)local\_to\_ls = K * 2^{\lceil Global\_view/N \rceil}$

Proof: The timeouts are set by functions specified in Figure A-15. The global timeout $global\_to$ is a deterministic function of the global view, $global\_to = K * 2^{\lceil Global\_view/N \rceil}$, where K is the minimum global timeout and N is the number of sites. Therefore, all servers in the same global view will compute the same global timeout (line C1). The RESET-GLOBAL-TIMER function sets the value of Global_T to $global\_to$. The RESET-LOCAL-TIMER function sets the value of Local_T depending on whether the server is in the leader site. If the server is in the leader site, the Local_T timer is set to $local\_to\_ls = (global\_to/(f+3))$ (line B2). If the server is not in the leader site, the Local_T timer is set $local\_to\_nls = local\_to\_ls/(f+2)$ (line B4). Therefore, the above ratios hold for all servers in the same global view. □

We now prove that each time a site becomes the leader site in a new global view, correct representatives in this site will be able to communicate with at least one correct representative in all other sites. This follows from the timeout relationships in Claim B.7. Moreover, we show that each time a site becomes the leader, it will have more time to communicate with each correct representative. Intuitively, this claim follows from the relative rates at which the coordinators rotate at the leader and non-leader sites.

Claim *B.8:* If LS is the leader site in global views gv and gv' with $gv > gv'$, then any stable representative elected in gv can communicate with a stable representative at all stable non-leader sites for time Δ_{gv}, and any stable representative elected in gv' can communicate with a stable representative at all stable non-leader sites for time $\Delta_{gv'}$ and $\Delta_{gv} \geq 2 * \Delta_{gv'}$.

Proof: From Claim B.6, if no global progress occurs, (1) local views will be installed consecutively, and (2) the servers will remain in the same local view for one local timeout. Therefore, any correct representative at the leader site will reign for one local timeout at the leader site, $local\_to\_ls$. Similarly, any correct representative at a non-leader site will reign for approximately one local timeout at a non-leader site, $local\_to\_nls$.

From Claim B.7, the local timeout at the leader site is $f+2$ times the local timeout at the non-leader site ($local\_to\_ls = (f+2)local\_to\_nls$). If stable server r is representative for $local\_to\_ls$, then, at each leader site, there will be at least $f+1$ servers that are representative for time $local\_to\_nls$ during the time that r is representative. Since the representative has a Server_id equal to Local_view $\mathrm{mod}(3f+1)$, a server can never be elected representative twice during $f+1$ consecutive local views. It follows that a stable representative in the leader site can communicate with $f+1$ different servers for time period $local\_to\_ls$. Since there are at most f servers that are not stable, at least one of the $f+1$ servers must be stable.

From Claim B.7, the global timeout doubles every N consecutive global views, where N is the number of sites. The local timeouts are a constant fraction of a global timeout, and, therefore, they grow at the same rate as the global timeout. Since the leader site has Site_id = Global_view $\bmod N$, a leader site is elected exactly once every N consecutive global views. Therefore, each time a site becomes the leader, the local and global timeouts double. □

Claim *B.9:* If global progress does not occur and the system is stable with respect to time T, then in any global view gv that begins after time T, there will be at least two stable representatives in the leader site that are each leaders for a local timeout at the leader site, $local\_to\_ls$.

Proof: From Claim B.6, if no global progress occurs, (1) local views will be installed consecutively, and (2) the servers will remain in the same local view for one local timeout. From Claim B.4, if no global progress occurs, the servers in the same global view will remain in this global view for one global timeout, $global\_to$. From Claim B.7, $global\_to = (f+3)local\_to\_ls$. Therefore, during the time when all stable servers are in global view gv, there will be $f+2$ representatives in the leader site that each serve for $local\_to\_ls$. We say that these servers have complete reigns in gv. Since the representative has a Server_id equal to Local_view $\bmod(3f + 1)$, a server can never be elected representative twice during $f + 2$ consecutive local views. There are at most f servers in a stable site that are not stable, therefore at least two of the $f + 2$ servers that have complete reigns in gv will be stable. □

We now proceed with our main argument for proving Claim B.2, which will show that a stable server will be able to complete the GLOBAL-VIEW-CHANGE protocol. To complete GLOBAL-VIEW-CHANGE in a global view gv, a stable representative must coordinate the construction of an Aru_Message, send the Aru_Message to the other sites, and collect Global_Constraint messages from a majority of sites. We leverage the properties of the global and local timeouts to show that, as the stable sites move through global views together, a stable representative of the leader site will eventually remain in power long enough to complete the protocol, provided each component of the protocol completes in finite time. This intuition is encapsulated in the following lemma:

Lemma *B.9:* If global progress does not occur and the system is stable with respect to time T, then there exists an infinite set of global views gv_i, each with an associated local view lv_i and a stable leader site S_i, in which, if CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT complete in bounded finite times, then if the first stable representative of S_i serving for at least a local timeout period invokes GLOBAL-VIEW-CHANGE, it will complete the protocol in $(gv_i,\ lv_i)$.

Proof: By Claim B.4, if the system is stable and no global progress is made, all stable servers move together through all (consecutive) global views gv above some initial synchronization view, and they remain in gv for at least one global timeout period, which increases by at least a factor of two every N global view changes. Since the stable sites preinstall consecutive global views, an infinite number of stable leader sites will be elected. By Claim B.9, each such stable leader site elects three stable representatives before the Global_T timer of any stable server expires, two of which remain in power for at least a local timeout period before any stable server in S expires its Local_T timeout. We now show that we can continue to increase this timeout period (by increasing the value of gv) until, if CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT complete in bounded finite times Δ_{aru} and Δ_{gc}, respectively, the representative will complete GLOBAL-VIEW-CHANGE.

A stable representative invokes CONSTRUCT-ARU after invoking the GLOBAL-VIEW-CHANGE protocol (Figure A-16, line A2), which occurs either after preinstalling the global view (Figure A-14, line E8) or after completing a local view change when not globally constrained (Figure A-13, line D8). Since the duration of the local timeout period $local\_to\_ls$ increases by at least a factor of two every N global view changes, there will be a global view gv in which the local timeout period is greater than Δ_{aru}, at which point the stable representative has enough time to construct the Aru_Message.

By Claim B.8, if no global progress occurs, then a stable representative of the leader site can communicate with a stable representative at each stable non-leader site in a global view gv for some amount of time, Δ_{gv}, that increases by at least a factor of two every N global view changes. The stable representative of the leader site receives a New_Rep message containing the identity of the new site representative from each stable site roughly one wide area message delay after the non-leader site representative is elected. Since Δ_{gc} is finite, there is a global view sufficiently large such that (1) the leader site representative can send the Aru_Message it constructed to each non-leader site representative, the identity of which it learns from the New_Rep message, (2) each non-leader site representative can complete CONSTRUCT-GLOBAL-CONSTRAINT, and (3) the leader site representative can collect Global_Constraint messages from a majority of sites. We can apply the same logic to each subsequent global view gv' with a stable leader site. □

We call the set of views for which Lemma B.9 holds the *completion views*. Intuitively, a completion view is a view $(gv,\ lv)$ in which the timeouts are large enough such that, if CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT complete in some bounded finite amounts of time, the stable representative of the leader site S of gv (which is the first stable representative of S serving for at least a local timeout period) will complete the GLOBAL-VIEW-CHANGE protocol.

Given Lemma B.9, it just remains to show that there exists a completion view in which CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT terminate in bounded finite time. We use Claim B.1 to leverage the fact that all stable servers eventually reconcile their Global_History data structures to max_stable_seq to bound the amount of work required by each protocol. Since there are an infinite number of completion views, we consider those completion views in which this reconciliation has already completed.

We first show that there is a bound on the size of the Global_Server_State messages used in CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT.

Lemma *B.10:* If all stable servers have a Global_aru of max_stable_seq, then no server can have a Prepare Certificate, Proposal, or Globally_Ordered_Update for any sequence number greater than $(max\_stable\_seq + 2 * W)$.

Proof: Since obtaining a Globally_Ordered_Update requires a Proposal, and generating a Proposal requires collecting a Prepare Certificate, we assume that a Prepare Certificate with a sequence number greater than $(max\_stable\_seq + 2 * W)$ was generated and show that this leads to a contradiction.

If any server collects a Prepare Certificate for a sequence number seq greater than $(max\_stable\_seq+2*W)$, then it collects a Pre-Prepare message and $2f$ Prepare messages for $(max\_stable\_seq+2*W)$. This implies that at least $f+1$ correct servers sent either a Pre-Prepare or a Prepare. A correct representative only sends a Pre-Prepare message for seq if its Global_aru is at least $(seq-W)$ (Figure A-11, line A3), and a correct server only sends a Prepare message if its Global_aru is at least $(seq-W)$ (Figure A-8, A23). Thus, at least $f+1$ correct servers had a Global_aru of at least $(seq-W)$.

For this to occur, these $f+1$ correct servers obtained Globally_Ordered_Updates for those sequence numbers up to and including $(seq-W)$. To obtain a Globally_Ordered_Update, a server collects a Proposal message and $\lfloor S/2 \rfloor$ corresponding Accept messages. To construct a Proposal for $(seq-W)$, at least $f+1$ correct servers in the leader site had a Global_aru of at least $(seq-2W) > max\_stable\_seq$. Similarly, to construct an Accept message, at least $f+1$ correct servers in a non-leader site contributed a Partial_sig message. Thus, there exists a majority of sites, each with at least $f+1$ correct servers with a Global_aru greater than max_stable_seq.

Since any two majorities intersect, one of these sites is a stable site. Thus, there exists a stable site with some stable server with a Global_aru greater than max_stable_seq, which contradicts the definition of max_stable_seq. □

Lemma *B.11:* If all stable servers have a Global_aru of max_stable_seq, then if a stable representative of the leader site invokes CONSTRUCT-ARU, or if a stable server in a non-leader site invokes CONSTRUCT-GLOBAL-CONSTRAINT with an Aru_Message containing a sequence number at least max_stable_seq, then any valid Global_Server_State message will contain at most $2*W$ entries.

Proof: A stable server invokes CONSTRUCT-ARU with an invocation sequence number of max_stable_seq. By Lemma B.10, no server can have a Prepare Certificate, Proposal, or Globally_Ordered_Update for any sequence number greater than $(max\_stable\_seq+2*W)$. Since these are the only entries reported in a valid Global_Server_State message (Figure A-20, Block B), the lemma holds. We use the same logic as above in the case of CONSTRUCT-GLOBAL-CONSTRAINT. □

The next two lemmas show that CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT will complete in bounded finite time.

Lemma *B.12:* If the system is stable with respect to time T and no global progress is made, then there exists an infinite set of views (gv_i, lv_i) in which a run of CONSTRUCT-ARU invoked by the stable representative of the leader site will complete in some bounded finite time, Δ_{aru}.

Proof: By Claim B.1, if no global progress is made, then all stable servers eventually reconcile their Global_aru to max_stable_seq. We consider those completion views in which this reconciliation has already completed.

The representative of the completion view invokes CONSTRUCT-ARU upon completing GLOBAL-LEADER-ELECTION (Figure A-16, line A2). It sends a Request_Global_State message to all local servers containing a sequence number reflecting its current Global_aru value. Since all stable servers are reconciled up to max_stable_seq, this sequence number is equal to max_stable_seq. Since the leader site is stable, all stable servers receive the Request_Global_State message within one local message delay.

When a stable server receives the Request_Global_State message, it immediately sends a Global_Server_State message (Figure A-18, lines B5-B7), because it has a Global_aru of max_stable_seq. By Lemma B.11, any valid Global_Server_State message can contain entries for at most $2*W$ sequence numbers. We show below in Claim B.11 that all correct servers have contiguous entries above the invocation sequence number in their Global_History data structures. From Figure A-20 Block B, the Global_Server_State message from a correct server will contain contiguous entries. Since the site is stable, the representative collects valid Global_Server_State messages from at least $2f+1$ servers, bundles them together, and sends the Global_Collected_Servers_State message to all local servers (Figure A-18, line C3).

Since the representative is stable, and all stable servers have a Global_aru of max_stable_seq (which is equal to the invocation sequence number), all stable servers meet the conditionals at Figure A-18, lines D2 and D3. They do not see a conflict at Figure A-7, line F4, because the representative only collects Global_Server_State messages that are contiguous. They construct the union message by completing Compute_Global_Union (line D4), and invoke THRESHOLD-SIGN on each Prepare Certificate in the union. Since there are a finite number of entries in the union, there are a finite number of Prepare Certificates. By Lemma B.3, all stable servers convert the Prepare Certificates into Proposals and invoke THRESHOLD-SIGN on the union (line F2). By Lemma B.3, all stable servers generate the Global_Constraint message (line G1) and invoke THRESHOLD-SIGN on the extracted union_aru (line G4). By Lemma B.3, all stable servers generate the Aru_Message and complete the protocol.

Since gv_i can be arbitrarily high, with the timeout period increasing by at least a factor of two every N global view changes, there will eventually be enough time to complete the bounded amount of computation and communication in the protocol. We apply the same logic to all subsequent global views with a stable leader site to obtain the infinite set. □

Lemma *B.13:* Let A be an Aru_Message containing a sequence number of max_stable_seq. If the system is stable with respect to time T and no global progress is made, then there exists an infinite set of views (gv_i, lv_i) in which a run of CONSTRUCT-GLOBAL-CONSTRAINT invoked by a stable server in local view lv_i, where the representative of lv_i is stable, in a non-leader site with argument A, will complete in some bounded finite time, Δ_{gc}.

Proof: By Claim B.1, if no global progress is made, then all stable servers eventually reconcile their Global_aru to max_stable_seq. We consider those completion views in which this reconciliation has already occurred.

The Aru_Message A has a value of at max_stable_seq. Since the representative of lv' is stable, it sends A to all servers in its site. All stable servers receive A within one local message delay.

All stable servers invoke CONSTRUCT-GLOBAL-CONSTRAINT upon receiving A and send Global_Server_State messages to the representative. By Lemma B.11, the Global_Server_State messages contain entries for at most $2 * W$ sequence numbers. We show below in Claim B.11 that all correct servers have contiguous entries above the invocation sequence number in their Global_History data structures. From Figure A-20 Block B, the Global_Server_State message from a correct server will contain contiguous entries. The representative will receive at least $2f + 1$ valid Global_Server_State messages, since all messages sent by stable servers will be valid. The representative bundles up the messages and sends a Global_Collected_Servers_State message (Figure A-19, line B3).

All stable servers receive the Global_Collected_Servers_State message within one local message delay. The message will meet the conditional at line C2, because it was sent by a stable representative. They do not see a conflict at Figure A-7, line F4, because the representative only collects Global_Server_State messages that are contiguous. All stable servers construct the union message by completing Compute_Global_Union (line C3), and invoke THRESHOLD-SIGN on each Prepare Certificate in the union. Since all valid Global_Server_State messages contained at most $2 * W$ entries, there are at most $2 * W$ entries in the union and $2 * W$ Prepare Certificates in the union. By Lemma B.3, all stable servers convert the Prepare Certificates into Proposals and invoke THRESHOLD-SIGN on the union (line E2). By Lemma B.3, all stable servers generate the Global_Constraint message (line F2).

Since gv_i can be arbitrarily high, with the timeout period increasing by at least a factor of two every N global view changes, there will eventually be enough time to complete the bounded amount of computation and communication in the protocol. We apply the same logic to all subsequent global views with a stable leader site to obtain the infinite set. □

Finally, we can prove Claim B.2:

Proof: By Lemma B.9, the first stable representative of some leader site S can complete GLOBAL-VIEW-CHANGE in a completion view (gv, lv) if CONSTRUCT-ARU and CONSTRUCT-GLOBAL-CONSTRAINT complete in bounded finite time. By Lemmas B.12, S can complete CONSTRUCT-ARU in bounded finite time. This message is sent to a stable representative in each non-leader site, and by Lemma B.13, CONSTRUCT-GLOBAL-CONSTRAINT completes in bounded finite time. We apply this logic to all global views with stable leader site above gv, completing the proof. □

We now show that either the first or the second stable representative of the leader site serving for at least a local timeout period will make global progress, provided at least one stable server receives an update that it has not previously executed. This then implies our liveness condition.

We begin by showing that a stable representative of the leader site that completes GLOBAL-VIEW-CHANGE and serves for at least a local timeout period will be able to pass the Global_Constraint messages it collected to the other stable servers. This implies that subsequent stable representatives will not need to run the GLOBAL-VIEW-CHANGE protocol (because they will already have the necessary Global_Constraint messages and can become globally constrained) and can, after becoming locally constrained, attempt to make progress.

Lemma *B.14:* If the system is stable with respect to time T, then there exists an infinite set of global views gv_i in which either global progress occurs during the reign of the first stable representative at a stable leader site to serve for at least a local timeout period, or any subsequent stable representative elected at the leader site during gv_i will already have a set consisting of a majority of Global_Constraint messages from gv_i.

Proof: By Claim B.2, there exists an infinite set of global views in which the first stable representative serving for at least a local timeout period will complete GLOBAL-VIEW-CHANGE. To complete GLOBAL-VIEW-CHANGE, this representative collects Global_Constraint_Messages from a majority of sites. The representative sends a signed Collected_Global_Constraints message to all local servers (Figure A-13, line D11). Since the site is stable, all stable servers receive this message within one local message delay. If we extend the reign of the stable representative that completed GLOBAL-VIEW-CHANGE by one local message delay (by increasing the value of gv), then in all subsequent local views in this global view, a stable representative will already have

Global_Constraint_Messages from a majority of servers. We apply the same logic to all subsequent global views with a stable leader site to obtain the infinite set. □

We now show that if no global progress is made during the reign of the stable representative that completed GLOBAL-VIEW-CHANGE, then a second stable representative that is already globally constrained will serve for at least a local timeout period.

Lemma *B.15:* If the system is stable with respect to time T, then there exists an infinite set of global views gv_i in which either global progress occurs during the reign of the first stable representative at a stable leader site to serve for at least a local timeout period, or a second stable representative is elected that serves for at least a local timeout period and which already has a set consisting of a majority of Global_Constraint(gv_i) messages upon being elected.

Proof: By Lemma B.14, there exists an infinite set of global views in which, if no global progress occurs during the reign of the first stable representative to serve at least a local timeout period, all subsequent stable representatives already have a set consisting of a majority of Global_Constraint messages upon being elected. We now show that a second stable representative will be elected.

By Claim B.8, if no global progress is made, then the stable leader site of some such gv will elect $f+3$ representatives before any stable server expires its Global_T timer, and at least $f+2$ of these representatives serve for at least a local timeout period. Since there are at most f faulty servers in the site, at least two of these representatives will be stable. □

Since globally ordering an update requires the servers in the leader site to be locally constrained, we prove the following lemma relating to the CONSTRUCT-LOCAL-CONSTRAINT protocol:

Lemma *B.16:* If the system is stable with respect to time T and no global progress occrs, then there exists an infinite set of views (gv_i, lv_i) in which a run of CONSTRUCT-LOCAL-CONSTRAINT invoked by a stable representative of the leader site will complete at all stable servers in some bounded finite time, Δ_{lc}.

To prove Lemma B.16, we use the following two lemmas to bound the size of the messages sent in CONSTRUCT-LOCAL-CONSTRAINT:

Lemma *B.17:* If the system is stable with respect to time T, no global progress is made, and all stable servers have a Global_aru of max_stable_seq, then no server in any stable leader site S has a Prepare Certificate or Proposal message in its Local_History data structure for any sequence number greater than ($max\_stable\_seq + W$).

Proof: We show that no server in S can have a Prepare Certificate for any sequence number s', where $s' > (max\_stable\_seq + W)$. This implies that no server has a Proposal message for any such sequence number s', since a Prepare Certificate is needed to construct a Proposal message.

If any server has a Prepare Certificate for a sequence number $s' > (max\_stable\_seq + W)$, it collects a Pre-Prepare and a Prepare from $2f+1$ servers. Since at most f servers in S are faulty, some stable server sent a Pre-Prepare or a Prepare for sequence number s'. A correct representative only sends a Pre-Prepare message for those sequence numbers in its window (Figure A-11, line A3). A non-representative server only sends a Prepare message for those sequence numbers in its window, since otherwise it would have a conflict (Figure A-8, line A23). This implies that some stable server has a window that starts after max_stable_seq, which contradicts the definition of max_stable_seq. □

Lemma *B.18:* If no global progress occurs, and all stable servers have a Global_aru of max_stable_seq when installing a global view gv, then if a stable representative of a leader site S invokes CONSTRUCT-LOCAL-CONSTRAINT in some local view (gv, lv), any valid Local_Server_State message will contain at most W entries.

Proof: When the stable representative installed global view gv, it set Pending_Proposal_Aru to its Global_aru (Figure A-16, line F4), which is max_stable_seq. Since Pending_Proposal_Aru only increases, the stable representative invokes CONSTRUCT-LOCAL-CONSTRAINT with a sequence number of at least max_stable_seq. A valid Local_Server_State message contains Prepare Certificates or Proposals for those sequence numbers greater than the invocation sequence number (Figure A-8, line D6). By Lemma B.17, no server in S has a Prepare Certificate or Proposal for a sequence number greater than ($max\_stable\_seq + W$), and thus, a valid message has at most W entries. □

We now prove Lemma B.16:

Proof: By Claim B.1, if no global progress is made, then all stable servers eventually reconcile their Global_Aru to max_stable_seq. We consider the global views in which this has already occurred.

When a stable server becomes globally constrained in some such view gv, it sets its Pending_Proposal_Aru variable to its Global_aru (Figure A-16, line F4), which is equal to max_stable_seq, since reconciliation has already occurred. A stable representative only increases its Pending_Proposal_Aru when it globally orders an update or constructs a Proposal for the sequence number one higher than its current Pending_Proposal_Aru (Figure

A-4, lines A5, A12, and C11). The stable representative does not globally order an update for $(max\_stable\_seq + 1)$, since when the server globally ordered an update for $(max\_stable\_seq + 1)$, it would have increased its Global_Aru and executed the update, which violates the definition of max_stable_seq. By Lemma B.17, no server in S has a Prepare Certificate or a Proposal message for any sequence number $s > (max\_stable\_seq + W)$. Thus, the stable representative's Pending_Proposal_Aru can be at most $max\_stable\_seq + W$ when invoking CONSTRUCT-LOCAL-CONSTRAINT

Since the representative of lv is stable, it sends a Request_Local_State message to all local servers, which arrives within one local message delay. All stable servers have a Pending_Proposal_Aru of at least max_stable_seq and no more than $(max\_stable\_seq + W)$. Thus, if a stable server's Pending_Proposal_Aru is at least as high as the invocation sequence number, it sends a Local_Server_State message immediately (Figure A-17, lines B5 - B7). Otherwise, the server requests Proposals for those messages in the difference, of which there are at most W. Since the site is stable, these messages will arrive in some bounded time that is a function of the window size and the local message delay.

By Lemma B.18, any valid Local_Server_State message contains at most W entries. We show below in Claim B.11 that all correct servers have contiguous entries above the invocation sequence number in their Local_History data structures. From Figure A-20 Block A, the Local_Server_State message from a correct server will contain contiguous entries. The representative will receive at least $2f + 1$ valid Local_Server_State messages, since all messages sent by stable servers will be valid. The representative bundles up the messages and sends a Local_Collected_Servers_State message. All stable servers receive the Local_Collected_Servers_State message within one local message delay. The message will meet the conditionals in Figure A-17, lines D2 and D3, at any stable server that sent a Local_Server_State message. They do not see a conflict at Figure A-8, line E4, because the representative only collects Local_Server_State messages that are contiguous. All stable servers apply the Local_Collected_Servers_State message to their Local_History data structures.

Since gv can be arbitrarily high, with the timeout period increasing by at least a factor of two every N global view changes, there will eventually be enough time for all stable servers to receive the Request_Local_Server state message, reconcile their Local_History data structures (if necessary) and send a Local_Server_State message, and process a Local_Collected_Servers_State message from the representative. Thus, there will eventually be enough time to complete the bounded amount of computation and communication in the protocol, and we can apply this argument to all subsequent global views with stable leader sites to obtain the infinite set. □

The following lemma encapsulates the notion that all stable servers will become globally and locally constrained shortly after the second stable representative to serve for at least a local timeout period is elected:

Lemma *B.19:* If the system is stable with respect to time T and no global progress occurs, then there exists an infinite set of views in which all stable servers become globally and locally constrained within Δ_{lc} time of the election of the second stable representative serving for at least a local timeout period.

Proof: By Lemma B.14, the second stable representative serving for at least a local timeout period will have a set of a majority of Global_Constraint messages from its current global view upon being elected. This server bundles up the messages, signs the bundle, and send it to all local servers as a Collected_Global_Constraints message (Figure A-13, line D11). Since the site is stable, all stable servers receive the message within one local message delay and become globally constrained. The stable representative also invokes CONSTRUCT-LOCAL-CONSTRAINT upon being elected (line D6). Since we consider those global views in which reconciliation has already occurred, Lemma B.16 implies that all stable servers become locally constrained within some bounded finite time. □

Since all stable servers are globally and locally constrained, the preconditions for attempting to make global progress are met. We use the following term in the remainder of the proof:

DEFINITION B.4: We say that a server is a **Progress_Rep** if (1) it is a stable representative of a leader site, (2) it serves for at least a local timeout period if no global progress is made, and (3) it can cause all stable servers to be globally and locally constrained within Δ_{lc} time of its election.

The remainder of the proof shows that, in some view, the Progress_Rep can globally order and execute an update that it has not previously executed (i.e., it can make global progress) if no global progress has otherwise occurred.

We first show that there exists a view in which the Progress_Rep has enough time to complete the ASSIGN-GLOBAL-ORDER protocol (i.e., to globally order an update), assuming it invokes ASSIGN-SEQUENCE. To complete ASSIGN-GLOBAL-ORDER, the Progress_Rep must coordinate the construction of a Proposal message, send the Proposal message to the other sites, and collect Accept messages from $\lfloor S/2 \rfloor$ sites. As in the case of the GLOBAL-VIEW-CHANGE protocol, we leverage the properties of the global and local timeouts to show that, as the stable sites move through global views together, the Progress_Rep will eventually remain in power long enough to complete the protocol, provided each component of the protocol completes in some bounded, finite time. This intuition is encapsulated in the following lemma:

Lemma *B.20:* If the system is stable with respect to time T and no global progress occurs, then there exists a view (gv, lv) in which, if ASSIGN-SEQUENCE and THRESHOLD-SIGN complete in bounded finite times, and all stable servers at all non-leader sites invoke THRESHOLD-SIGN on the same Proposal from gv, then if the Progress_Rep invokes ASSIGN-SEQUENCE at least once and u is the update on which it is first invoked, it will globally order u in (gv, lv).

Proof: By Claim B.1, if no global progress occurs, then all stable servers eventually reconcile their Global_aru to max_stable_seq. We consider the global views in which this has already occurred.

Since the Progress_Rep has a Global_aru of max_stable_seq, it assigns u a sequence number of $max\_stable\_seq + 1$. Since ASSIGN-SEQUENCE completes in some bounded, finite time Δ_{seq}, the Progress_Rep constructs P(gv, lv, $max\_stable\_seq + 1$, u), a Proposal for sequence number $max\_stable\_seq + 1$.

By Claim B.8, if no global progress occurs, then a stable representative of the leader site can communicate with a stable representative at each stable non-leader site in a global view gv for some amount of time, Δ_{gv}, that increases by at least a factor of two every N global view changes. Since we assume that THRESHOLD-SIGN is invoked by all stable servers at the stable non-leader sites and completes in some bounded, finite time, Δ_{sign}, there is a global view sufficiently large that (1) the leader site representative can send the Proposal P to each non-leader site representative, (2) each non-leader site representative can complete THRESHOLD-SIGN to generate an Accept message, and (3) the leader site representative can collect the Accept messages from a majority of sites. □

We now show that, if no global progress occurs and some stable server received an update that it had not previously executed, then some Progress_Rep *will* invoke ASSIGN-SEQUENCE. We assume that the reconciliation guaranteed by Claim B.1 has already completed (i.e., all stable servers have a Global_aru equal to max_stable_seq). From the pseudocode (Figure A-11, line A1), the Progress_Rep invokes ASSIGN-GLOBAL-ORDER after becoming globally and locally constrained. The Progress_Rep calls Get_Next_To_Propose to get the next update, u, to attempt to order (line A4). The only case in which the Progress_Rep will *not* invoke ASSIGN-SEQUENCE is when u is NULL. Thus, we must first show that Get_Next_To_Propose will not return NULL.

Within Get_Next_To_Propose, there are two possible cases:

1) Sequence number $max\_stable\_seq + 1$ is constrained: The Progress_Rep has a Prepare-Certificate or Proposal in Local_History and/or a Proposal in Global_History for sequence number $max\_stable\_seq + 1$.
2) Sequence number $max\_stable\_seq + 1$ is unconstrained.

We show that, if $max\_stable\_seq + 1$ is constrained, then u is an update that has not been executed by any stable server. If $max\_stable\_seq{+}1$ is unconstrained, then we show that if any stable server in site S received an update that it had not executed after the stabilization time, then u is an update that has not been executed by any stable server.

To show that the update returned by Get_Next_To_Propose is an update that has not yet been executed by any stable server, we must first show that the same update cannot be globally ordered for two different sequence numbers. Claim B.10 states that if a Globally_Ordered_Update exists that binds update u to sequence number seq, then no other Globally_Ordered_Update exists that binds u to seq', where $seq \neq seq'$. We use this claim to argue that if a server globally orders an update with a sequence number above its Global_aru, then this update could not have been previously executed. It follows immediately that if a server globally orders any update with a sequence number one greater than its Global_aru, then it will update execute this update and make global progress. We now formally state and prove Claim B.10.

Claim *B.10:* If a Globally_Ordered_Update(seq, u) exists, then there does not exist a Globally_Ordered_Update(seq', u), where $seq \neq seq'$.

We begin by showing that, if an update is bound to a sequence number in either a Pre-Prepare, Prepare-Certificate, Proposal, or Globally_Ordered_Update, then, within a local view at the leader site, it cannot be bound to a different sequence number.

Lemma *B.21:* If in some global and local views (gv, lv) at least one of the following constraining entries exist in the Global_History or Local_History of $f + 1$ correct servers:

1) Pre-Prepare(gv, lv, seq, u)
2) Prepare-Certificate(*, *, seq, u)
3) Proposal(*, *, seq, u)
4) Globally_Ordered_Update(*, *, seq, u)

Then, neither a Prepare-Certificate(gv, lv, seq', u) nor a Proposal(gv, lv, seq', u) can be constructed, where $seq \neq seq'$.

Proof: When a stable server receives a Pre-Prepare(gv, lv, seq, u), it checks its Global_History and Local_History for any constraining entries that contains update u. Lemma B.21 lists the message types that are examined. If there exists a constraining entry binding update u to seq', where $seq \neq seq'$, then Pre-Prepare, p, is ignored (Figure A-8, lines 25-26).

A Prepare-Certificate consists of $2f$ Prepares and a Pre-Prepare message. We assume that there are no more than f malicious servers and a constraining entry binding (seq, u), b, exists, and we show that there is a contradiction if Prepare-Certificate(gv, lv, seq', u), pc, exists. At least $f + 1$ correct servers must have contributed to pc. By assumption (as stated in Lemma B.21), at least $f + 1$ correct servers have constraining entry b. This leaves $2f$ servers (at most f that are malicious and the remaining that are correct) that do not have b and could contribute to pc. Therefore, at least one correct server that had constraint b must have contributed to pc. It would not do this if it were correct; therefore, we have a contradiction.

A correct server will not invoke THRESHOLD-SIGN to create a Proposal message unless a corresponding Prepare-Certificate exists. Therefore, it follows that, if Prepare-Certificate(gv, lv, seq', u) cannot exist, then Proposal(gv, lv, seq', u) cannot exist. □

We now use Invariant A.1 from *Proof of Safety*:

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 A.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 $gv' \geq gv$.

We use the Invariant A.1 to show that if a Proposal(gv, lv, seq, u) is constructed for the first time in global view gv, then a constraining entry that binds u to seq will exist in all views (gv, lv'), where $lv' \geq lv$.

Lemma *B.22:* Let P(gv, lv, seq, u) be the first threshold-signed Proposal message constructed by any server in leader site S binding update u to sequence number seq in global view gv. No other Proposal binding u to seq' can be constructed in global view gv, where $seq \neq seq'$.

Proof: We show that Invariant A.1 holds within the same global view in *Proof of Safety*. We now show that two Proposals having different sequence numbers and *the same* update cannot be created within the same global view.

From Lemma B.21 , if Proposal(gv, lv, seq, u), P, is constructed, then no constraining entries binding u to seq' exist in (gv, lv). Therefore, from Invariant A.1, no Proposal(gv, lv'', seq', u), P' could have been constructed, where $lv'' \leq lv$. This follows, because, if P' was constructed, then Invariant A.1 states that a constraint binding u to seq' would exist in view (gv, lv), in which case P could not have been constructed. In summary, we have proved that if P, binding u to seq, is constructed for the first time in some local view in gv, then *no other proposal binding u to seq'* was constructed in global view gv or earlier.

We assume that we create P. From Invariant A.1, after P was constructed, constraining messages will exist in all local views $\geq lv$. These constraining messages will always bind u to seq. Therefore, from Lemma B.21 no Proposal can be constructed that binds u to a different sequence number than in P in any local view lv', where $lv' \geq lv$. □

We now use Invariant A.2 from *Proof of Safety* in a similar argument:

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 A.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 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 $u' \neq u$.

We use the Invariant A.2 to show that if Globally_Ordered_Update(gv, lv, seq, u) is constructed, then there will be a majority of sites where at least $f + 1$ correct servers in each site have a constraining entry that binds u to seq in all global views greater than or equal to gv. From this, it follows that any set of Global_Constraint messages from a majority of sites will contain an entry that binds u to seq.

Lemma *B.23:* Let G(gv, lv, seq, u) be the first Globally_Ordered_Update constructed by any server. No other Prepare-Certificate or Proposal binding u to seq' can be constructed.

Proof: We show that Invariant A.2 holds across global views in *Proof of Safety*. We now show that if Globally_Ordered_Update(gv, lv, seq, u), G, is constructed at any server, then no Prepare-Certificate or Proposal having different sequence numbers and *the same* update can exist.

If G exists, then Proposal(gv, lv, seq, u), P, must have been created. From Lemma B.21, if P was constructed, then no constraining entries binding u to seq' exist in (gv, lv). Therefore, from Invariant A.2, no Globally_Ordered_Update(gv, lv'', seq', u), G' could have been constructed, where $lv'' \leq lv$. This follows, because, if G' was constructed, then Invariant A.1 implies that a constraint binding u to seq' would exist in views (gv, lv), in which case G could not have been constructed. *Proof of Satefy* proves this in detail. To summarize, if a majority of sites each have at least $f+1$ correct servers that have a global constraining entry, b, then these sites would all generate Global_Constraint messages that include b. To become globally constrained, correct servers must apply a bundle of Global_Constraint messages from a majority of sites, which includes one Global_Constraint message that contains b. A correct server will never send a Prepare or Pre-Prepare message without first becoming globally constrained. Therefore, if G' was constructed, then there would have been a constraint binding u to seq' in the site where G was constructed. We have already shown that this was not possible, because G was constructed. In summary, we have proved that if G, binding u to seq, is constructed for the first time in some global view gv, then *no Globally_Ordered_Update binding* u *to* seq' was constructed in global view gv or earlier.

We assume that we construct G. Invariant A.2, implies that in all global views $\geq gv$, constraining messages, binding u to seq, will exist in at least $f+1$ servers at the leader site when a leader site constructs a Proposal. Therefore, from Lemma B.21 no Proposal can be constructed that binds u to a different sequence number than in seq in any local view lv', where $lv' \geq lv$. □

We now return to the first case within Get_Next_To_Propose, where $(max\_stable\_seq + 1)$ is constrained at the Progress_Rep.

Lemma *B.24:* If sequence number $(max\_stable\_seq + 1)$ is constrained when a Progress_Rep calls Get_Next_To_Propose, then the function returns an update u that has not previously been executed by any stable server.

Proof: From Figure A-12 lines A2 - A5, if $(max\_stable\_seq + 1)$ is constrained at the Progress_Rep, then Get_Next_To_Propose returns the update u to which the sequence number is bound.

We assume that u has been executed by some stable server and show that this leads to a contradiction. Since u was executed by a stable server, it was executed with some sequence number s less than or equal to max_stable_seq. By Lemma B.23, if u has already been globally ordered with sequence number s, no Prepare Certificate, Proposal, or Globally_Ordered_Update can be constructed for any other sequence number s' (which includes $(max\_stable\_seq + 1)$). Thus, the constraining update u cannot have been executed by any stable server, since all executed updates have already been globally ordered. □

We now consider the second case within Get_Next_To_Propose, in which $(max\_stable\_seq + 1)$ is unconstrained at the Progress_Rep (Figure A-12, lines A6 - A7). In this part of the proof, we divide the Update_Pool data structure into two logical parts:

DEFINITION B.5: We say an update that was added to the Update_Pool is in a logical **Unconstrained_Updates** data structure if it does not appear as a Prepare Certificate, Proposal, or Globally_Ordered_Update in either the Local_History or Global_History data structure.

We begin by showing that, if some stable server in site R received an update u that it had not previously executed, then either global progress occurs or the Progress_Rep of R eventually has u either in its Unconstrained_Updates data structure or as a Prepare Certificate, Proposal, or Globally_Ordered_Update constraining some sequence number.

Lemma *B.25:* If the system is stable with respect to time T, and some stable server r in site R receives an update u that it has not previously executed at some time $T' > T$, then either global progress occurs or there exists a view in which, if sequence number $(max\_stable\_seq + 1)$ is unconstrained when a Progress_Rep calls Get_Next_To_Propose, then the Progress_Rep has u either in its Unconstrained_Updates data structure or as a Prepare_Certificate, Proposal, or Globally_Ordered_Update.

Proof: If any stable server previously executed u, then by Claim B.1, all stable servers (including r) will eventually execute the update and global progress occurs.

When server r receives u, it broadcasts u within its site, R (Figure A-3, line F2). Since R is stable, all stable servers receive u within one local message delay. From Figure A-3, line F5, they place u in their Unconstrained_Updates data structure. By definition, u is only removed from the Unconstrained_Updates (although it remains in the Update_Pool) if the server obtains a Prepare Certificate, Proposal, or Globally_Ordered_Update binding u to a sequence number. If the server later removes this binding, the update is placed back into the Unconstrained_Updates data structure. Since u only moves between these two states, the lemma holds. □

Lemma B.25 allows us to consider two cases, in which some new update u, received by a stable server in site R, is either in the Unconstrained_Updates data structure of the Progress_Rep, or it is constraining some other sequence number. Since there are an infinite number of consecutive views in which a Progress_Rep exists, we consider those views in which R is the leader site. We first examine the former case:

Lemma *B.26:* If the system is stable with respect to time T, and some stable server r in site R receives an update u that it has not previously executed at some time $T' > T$, then if no global progress occurs, there exists a view in which, if sequence number $(max\_stable\_seq + 1)$ is unconstrained when a Progress_Rep calls Get_Next_To_Propose and u is in the Unconstrained_Updates data structure of the Progress_Rep, Get_Next_To_Propose will return an update not previously executed by any stable server.

Proof: By Lemma B.25, u is either in the Unconstrained_Updates data structure of the Progress_Rep or it is constraining some other sequence number. Since u is in the Unconstrained_Updates data structure of the Progress_Rep and $(max\_stable\_seq + 1)$ was unconstrained, u or some other unconstrained update will be returned from Get_Next_To_Propose (Figure A-12, line A7). The returned update cannot have been executed by any stable server, since by Claim B.1, all stable servers would have executed the update and global progress would have been made. □

We now examine the case in which $(max\_stable\_seq + 1)$ is unconstrained at the Progress_Rep, but the new update u is not in the Unconstrained_Updates data structure of the Progress_Rep. We will show that this case leads to a contradiction: since u is constraining some sequence number in the Progress_Rep's data structures other than $(max\_stable\_seq + 1)$, some other new update necessarily constrains $(max\_stable\_seq + 1)$. This implies that if $(max\_stable\_seq + 1)$ is unconstrained at the Progress_Rep, u *must* be in the Unconstrained_Updates data structure. In this case, Get_Next_To_Propose will return either u or some other unconstrained update that has not yet been executed by any stable server.

To aid in proving this, we introduce the following terms:

DEFINITION B.6: We say that a Prepare Certificate, Proposal, or Globally_Ordered_Update is a **constraining entry** in the Local_History and Global_History data structures.

DEFINITION B.7: We say that a server is **contiguous** if there exists a constraining entry in its Local_History or Global_History data structures for all sequence numbers up to and including the sequence number of the server's highest constraining entry.

We will now show that all correct servers are always contiguous. Since correct servers begin with empty data structures, they are trivially contiguous when the system starts. Moreover, all Local_Collected_Servers_State and Collected_Global_Constraints bundles are empty until the first view in which some server collects a constraining entry. We now show that, if a server begins a view as contiguous, it will remain contiguous. The following lemma considers data structure modifications made during normal case operation; specifically, we defer a discussion of modifications made to the data structures by applying a Local_Collected_Servers_State or Collected_Global_Constraints message, which we consider below.

Lemma *B.27:* If a correct server is contiguous before inserting a constraining entry into its data structure that is not part of a Local_Collected_Servers_State or Collected_Global_Constraints message, then it is contiguous after inserting the entry.

Proof: There are three types of constraining entries that must be considered. We examine each in turn.

When a correct server inserts a Prepare Certificate into either its Local_History or Global_History data structure, it collects a Pre-Prepare and $2f$ corresponding Prepare messages. From Figure A-3, lines B2 - B33, a correct server ignores a Prepare message unless it has a Pre-Prepare for the same sequence number. From Figure A-8, line A21, a correct server sees a conflict upon receiving a Pre-Prepare unless it is contiguous up to that sequence number. Thus, when the server collects the Prepare Certificate, it must be contiguous up to that sequence number.

Similarly, when a server in a non-leader site receives a Proposal message with a given sequence number, it only applies the update to its data structure if it is contiguous up to that sequence number (Figure A-7, line A9). For those servers in the leader site, a Proposal is generated when the THRESHOLD-SIGN protocol completes (Figure A-10, lines D2 and D3). Since a correct server only invokes THRESHOLD-SIGN when it collects a Prepare Certificate (line C7), the server at least has a Prepare Certificate, which is a constraining entry that satisfies the contiguous requirement.

A correct server will only apply a Globally_Ordered_Update to its Global_History data structure if it is contiguous up to that sequence number (Figure A-4, line C2).

During CONSTRUCT-ARU or CONSTRUCT-GLOBAL-CONSTRAINT, a server converts its Prepare Certificates to Proposals by invoking THRESHOLD-SIGN, but a constraining entry still remains for each sequence number that was in a Prepare Certificate after the conversion completes. □

The only other time a contiguous server modifies its data structures is when it applies a Local_Collected_Servers_State or Collected_Global_Constraints message to its data structures. We will now show that the union computed on any Local_Collected_Servers_State or Collected_Global_Constraints message will result in a contiguous set of constraining entries directly above the associated invocation sequence number. We will then show that, if a contiguous server applies the resultant union to its data structure, it will be contiguous after applying.

We begin by showing that any valid Local_Collected_Servers_State message contains contiguous constraining entries beginning above the invocation sequence number.

Lemma *B.28:* If all correct servers are contiguous during a run of CONSTRUCT-LOCAL-CONSTRAINT, then any contiguous server that applies the resultant Local_Collected_Servers_State message will be contiguous after applying.

Proof: A correct server sends a Local_Server_State message in response to a Request_Local_State message containing some invocation sequence number, seq (Figure A-17, line B7). The server includes all constraining entries directly above seq (Figure A-20, Block A). Each Local_Server_State message sent by a contiguous server will therefore contain contiguous constraining entries beginning at $seq + 1$. The representative collects $2f + 1$ Local_Server_State messages. By Figure A-8 line E4, each Local_Server_State message collected is enforced to be contiguous. When the Local_Collected_Servers_State bundle is received from the representative, it contains $2f + 1$ messages, each with contiguous constraining entries beginning at $seq + 1$. The Local_Collected_Servers_State message is only applied when a server's Pending_Proposal_Aru is at least as high as the invocation sequence number contained in the messages within (Figure A-17, lines D3 - D4). Since the Pending_Proposal_Aru reflects Proposals and Globally_Ordered_Updates, the server is contiguous up to and including the invocation sequence number when applying.

When Compute_Local_Union is computed on the bundle (Figure A-3, line D2), the result must contain contiguous constraining entries beginning at $seq + 1$, since it is the union of contiguous messages. After applying the union, the server removes all constraining entries above the highest sequence number for which a constraining entry appeared in the union, and thus it will still be contiguous. □

We now use a similar argument to show that any contiguous server applying a Collected_Global_Constraints message to its data structure will be contiguous after applying:

Lemma *B.29:* If all correct servers are contiguous during a run of GLOBAL-VIEW-CHANGE, then any contiguous server applying the resultant Collected_Global_Constraints message to its data structure will be contiguous after applying.

Proof: Using the same logic as in Lemma B.28 (but using the Global_History and Global_aru instead of the Local_History and Pending_Proposal_Aru), any Global_Constraint message generated will contain contiguous entries beginning directly above the invocation sequence number contained in the leader site's Aru_Message. The Collected_Global_Constraints message thus consists of a majority of Global_Constraints messages, each with contiguous constraining entries beginning directly above the invocation sequence number. When Compute_Constraint_Union is run (Figure A-4, line D2), the resultant union will be contiguous. A contiguous server only applies the Collected_Global_Constraints message if its Global_aru is at least as high as the invocation sequence number reflected in the messages therein (Figure A-7, lines H5 - H6), and thus it is contiguous up to that sequence number. When Compute_Constraint_Union is applied (Figure A-21, Blocks E and F) the server only removes constraining entries for those sequence numbers above the sequence number of the highest constraining entry in the union, and thus the server remains contiguous after applying. □

We can now make the following claim regarding contiguous servers:

Claim *B.11:* All correct servers are always contiguous.

Proof: When the system starts, a correct server has no constraining entries in its data structures. Thus, it is trivially contiguous. We now consider the first view in which any constraining entry was constructed. Since no constraining entries were previously constructed, any Local_Collected_Servers_State or Collected_Global_Constraints message applied during this view must be empty. By Lemma B.27, a contiguous server inserting a Prepare Certificate, Proposal, or Globally_Ordered_Update into its data structure during this view remains contiguous. Thus, when CONSTRUCT-LOCAL-CONSTRAINT or GLOBAL-VIEW-CHANGE are invoked, all correct servers are still contiguous. By Lemma B.28, any contiguous server that becomes locally constrained by applying a Local_Collected_Servers_State message to its data structure remains contiguous after applying. By Lemma B.29, any contiguous server that becomes globally constrained by applying a Collected_Global_Constraints message remains contiguous after applying. Since these are the only cases in which a contiguous server modifies its data structures, the claim holds. □

We can now return to our examination of the Get_Next_To_Propose function to show that, if $(max\_stable\_seq + 1)$ is unconstrained at the Progress_Rep, then some new update must be in the Unconstrained_Updates data structure of the Progress_Rep.

Lemma *B.30:* If the system is stable with respect to time T, and some stable server r in site R receives an update u that it has not previously executed at some time $T' > T$, then if no global progress occurs, there exists a view in which, if sequence number $(max\_stable\_seq + 1)$ is unconstrained when a Progress_Rep calls Get_Next_To_Propose, u must be in the Unconstrained_Updates data structure of the Progress_Rep.

Proof: Since the Progress_Rep is a stable, correct server, by Claim B.11, it is contiguous. This implies that, since $(max\_stable\_seq + 1)$ is unconstrained, the Progress_Rep does not have any constraining entry (i.e., Prepare Certificate, Proposal, or Globally_Ordered_Update) for any sequence number higher than $(max\_stable\_seq{+}1)$. By Lemma B.25, u must either be in the Unconstrained_Updates data structure or as a constrained entry. It is not a constrained entry, since the Progress_Rep has a Global_aru of max_stable_seq and has not executed u (since otherwise progress would have been made). Thus, u must appear in the Unconstrained_Updates data structure. □

Corollary *B.31:* If the system is stable with respect to time T, and some stable server r in site R receives an update u that it has not previously executed at some time $T' > T$, then if no global progress occurs, there exists an infinite set of views in which, if the Progress_Rep invokes Get_Next_To_Propose, it will return an update u that has not been executed by any stable server.

Proof: Follows immediately from Lemmas B.26 and B.30. □

Corollary B.31 implies that there exists a view in which a Progress_Rep will invoke ASSIGN-SEQUENCE with an update that has not been executed by any stable server, since it does so when Get_Next_To_Propose does not return NULL. We now show that there exists an infinite set of global views in which ASSIGN-SEQUENCE will complete in some bounded finite time.

Lemma *B.32:* If global progress does not occur, and the system is stable with respect to time T, then there exists an infinite set of views in which, if a stable server invokes ASSIGN-SEQUENCE when Global_seq $=$ seq, then ASSIGN-SEQUENCE will return Proposal with sequence number seq in finite time.

Proof: From Lemma B.14, there exists a view (gv, lv) where a stable representative, r, in the leader site S has Global_Constraint(gv) messages from a majority of sites. Server r will send construct and send a Collected_Global_Constraints(gv) to all stable servers in S. The servers become globally constrained when they process this message. From Lemma B.16, all stable servers in S will become locally constrained. To summarize, there exists a view (gv, lv) in which:

1) Stable representative r has sent Collected_Global_Constraints and a Local_Collected_Servers_State message to all stable servers. This message arrives at all stable servers in one local area message delay.
2) All stable servers in S have processed the constrain collections sent by the representative, and, therefore, all stable servers in S are globally and locally constrained.

We now proceed to prove that ASSIGN-SEQUENCE will complete in a finite time in two steps. First we show that the protocol will complete if there are no conflicts when the stable servers process the Pre-Prepare message from r. Then we show that there will be no conflicts.

When r invokes ASSIGN-SEQUENCE, it sends a Pre-Prepare(gv, lv, seq, u) to all servers in site S (Figure A-10, line A2). All stable servers in S will receive this message in one local area message delay. When a non-representative stable server receives a Pre-Prepare message (and there is no conflict), it will send a Prepare(gv, lv, seq, u) message to all servers in S (line B3). Therefore, since there are $2f$ stable servers that are not the representative, all stable servers in S will receive $2f$ Prepare messages and a Pre-Prepare message for (gv, lv, seq, u) (line C3). This set of $2f + 1$ messages forms a Prepare-Certificate(gv, lv, seq, u), pc. When a stable server receives pc, it invokes THRESHOLD-SIGN on an unsigned Proposal(gv, lv, seq, u) message (line C7). By Claim B.3, THRESHOLD-SIGN will return a correctly threshold signed Proposal(gv, lv, seq, u) message to all stable servers.

Now we must show that there are no conflicts when stable servers receive the Pre-Prepare message from r. Intuitively, there will be no conflicts because the representative of the leader site coordinates the constrained state of all stable servers in the site. To formally prove that there will not be a conflict when a stable server receives a Pre-Prepare(gv, lv, seq, u), $preprep$ from r, we consider block A of Figure A-8. We address each case in the following list. We first state the condition that must be true for there to be a conflict, then, after a colon, we state why this case cannot occur.

1) not locally constrained or not globally constrained: from the above argument, all servers are locally and globally constrained
2) $preprep$ is not from r: in our scenario, r sent the message
3) $gv \neq$ Global_view or $lv \neq$ Local_view: all servers in site S are in the same local and global views
4) There exists a Local_History[seq].Pre-Prepare(gv, lv, seq, u'), where $u' \neq u$: If there are two conflicting Pre-Prepare messages for the same

global and local views, then the representative at the leader site must have generated both messages. This will not happen, because r is a correct server and will not send two conflicting Pre-Prepares.

5) There exists either a Prepare-Certificate(gv, lv, seq, u') or a Proposal(gv, lv, seq, u') in Local_History[seq], where $u' \neq u$: A correct representative makes a single Local_Collected_Servers_State message, $lcss$. All stable servers become locally constrained by applying $lcss$ to their local data structures. Block D of Figure A-3 shows how this message is processed. First, the union is computed using a deterministic function that returns a list of Proposals and Prepare-Certificates having unique sequence numbers. The union also contains the invocation aru, the aru on which it was invoked. On Lines D5 - D11, all Pre-Prepares, Prepare-Certificates, and Proposals with local views $< lv$ (where lv is the local view of both the server and the Local_Collected_Servers_State message) are removed from the Local_History. Since no Pre-Prepares have been created in (gv, lv), no Prepare-Certificates or Proposals exist with higher local views than lv. Then, on D12 - D17, all Proposals and Prepare-Certificates in the union are added to the Local_History. Since all stable servers compute identical unions, these two steps guarantee that all stable servers will have identical Local_History data structures after they apply $lcss$. A correct representative will never invoke ASSIGN-SEQUENCE such that it sends Pre-Prepare(*, *, seq', *) where $seq' \leq$ the invocation aru. Therefore, when r invokes ASSIGN-SEQUENCE, it will send a Pre-Prepare(gv, lv, seq, u) that doesn't conflict with the Local_History of any stable server in S.
6) There exists either a Proposal(gv, lv, seq, u') or a Globally_Ordered_Update(gv, lv, seq, u') in Global_History[seq], where $u' \neq u$: A correct representative makes a single Collected_Global_Constraints message, cgc. All stable servers become globally constrained by applying cgc to their global data structures. Block D of Figure A-4 shows how this message is processed. First, the union is computed using a deterministic function that returns a list of Proposals and Globally_Ordered_Updates having unique sequence numbers. The union also contains the invocation aru, the aru on which GLOBAL-VIEW-CHANGE was invoked. On Lines D5 - D9, all Prepare-Certificates and Proposals with global views $< gv$ (where gv is the local view of both the server and the Collected_Global_Constraints message) are removed from the Global_History. Any Pre-Prepares or Proposals that have global views equal to gv must also be in the union and be consistent with the entry in the union. Then, on D10 - D14, all Proposals and Globally_Ordered_Updates in the union are added to the Global_History. Since all stable servers compute identical unions, these two steps guarantee that all stable servers will have identical Global_History data structures after they apply cgc. A correct representative will never invoke ASSIGN-SEQUENCE such that it sends Pre-Prepare(*, *, seq', *) where $seq' \leq$ the invocation aru. Therefore, when r invokes ASSIGN-SEQUENCE, it will send a Pre-Prepare(gv, lv, seq, u) than doesn't conflict with the Global_History of any stable server in S.
7) The server is not contiguous up to seq: A correct server applies the same Local_Collected_Servers_State and Collected_Global_Constraints messages as r. Therefore, as described in the previous two cases, the correct server has the same constraints in its Local_History and Global_History as r. By Lemma B.11, all correct servers are contiguous. Therefore, there will never be a conflict when a correct server receives an update from r that is one above r's Global_aru.
8) seq is not in the servers window: If there is no global progress, all servers will reconcile up to the same global sequence number, max_stable_seq. Therefore, there will be no conflict when a correct server receives an update from r that is one above r's Global_aru.
9) There exists a constraint binding update u to seq' in either the Local_History or Global_History: Since a correct server applies the same Local_Collected_Servers_State and Collected_Global_Constraints messages as r, the correct server has the same constraints in its Local_History and Global_History as r. Representative r will send a Pre-Prepare(*, *, seq, u) where either (1) u is in r's unconstrained update pool or (2) u is constrained. If u is constrained, then from Lemmas B.21, B.22, and B.23 the u must be bound to seq at both r and the correct server. This follows because two bindings (seq, u) and (seq', u) cannot exist in any correct server.

We have shown that a Pre-Prepare sent by r will not cause a conflict at any stable server. This follows from the fact that the local and global data structures of all stable servers will be in the same state for any sequence number where r sends Pre-Prepare(gv, lv, seq, u), as shown above. Therefore, Prepare messages sent by stable servers in response to the first Pre-Prepare message sent by r in (gv, lv) will also not cause conflicts. The arguments are parallel to those given in detail in the above cases.

We have shown that Pre-Prepare and Prepare messages sent by the stable servers will not cause conflicts when received by the stable servers. We have also shown that ASSIGN-SEQUENCE will correctly return a Proposal message if this is true, proving Lemma B.20. □

Having shown that ASSIGN-SEQUENCE will complete in a finite amount of time, we now show that the stable non-leader sites will construct Accept messages in a finite time. Since Claim B.3 states that THRESHOLD-SIGN completes in finite time if all stable servers invoke it on the same message, we must simply show that all stable servers will invoke THRESHOLD-SIGN upon receiving the Proposal message generated by ASSIGN-SEQUENCE.

Lemma *B.33:* If the system is stable with respect to time T and no global progress occurs, then there exists an infinite set of views (gv, lv) in which all stable servers at all non-leader sites invoke THRESHOLD-SIGN on a Proposal(gv, *, seq, u).

Proof: We consider the global views in which all stable servers have already reconciled their Global_aru to max_stable_seq and in which a Progress_Rep exists. By Corollary B.31, the Progress_Rep will invoke ASSIGN-SEQUENCE when Global_seq is equal to $max\_stable\_seq + 1$. By Lemma B.32, there exists an infinite set of views in which ASSIGN-SEQUENCE will return a Proposal in bounded finite time. By Claim B.8, there exists a view in which the Progress_Rep has enough time to send the Proposal to a stable representative in each stable non-leader site.

We must show that all stable servers in all stable non-leader sites will invoke THRESHOLD-SIGN on an Accept message upon receiving the Proposal. We first show that no conflict will exist at any stable server. The first two conflicts cannot exist (Figure A-7, lines A2 and A4), because the stable server is in the same global view as the stable servers in the leader site, and the server is in a non-leader site. The stable server cannot have a Globally_Ordered_Update in its Global_History data structure for this sequence number (line A6) because otherwise it would have executed the update, violating the definition of max_stable_seq. The server is contiguous up to $(max\_stable\_seq + 1)$ (line A9) because its Global_aru is max_stable_seq and it has a Globally_Ordered_Update for all previous sequence numbers. The sequence number is in its window (line A11) since $max\_stable\_seq < (max\_stable\_seq + 1) \leq (max\_stable\_seq + W)$.

We now show that all stable servers will apply the Proposal to their data structures. From Figure A-4, Block A, the server has either applied a Proposal from this view already (from some previous representative), in which case it would have invoked THRESHOLD-SIGN when it applied the Proposal, or it will apply the Proposal just received because it is from the latest global view. In both cases, all stable servers have invoked THRESHOLD-SIGN on the same message. □

Finally, we can prove L1 - GLOBAL LIVENESS:

Proof: By Claim B.1, if no global progress occurs, then all stable servers eventually reconcile their Global_aru to max_stable_seq. We consider those views in which this reconciliation has completed. By Lemma B.19, there exists an infinite set of views in which all stable servers become globally and locally constrained within a bounded finite time Δ_{lc} of the election of the second stable representative serving for at least a local timeout period (i.e., the Progress_Rep). After becoming globally and locally constrained, the Progress_Rep calls Get_Next_To_Propose to get an update to propose for global ordering (Figure A-11, line A4). By Corollary B.31, there exists an infinite set of views in which, if some stable server receives an update that it has not previously executed and no global progress has otherwise occurred, Get_Next_To_Propose returns an update that has not previously been executed by any stable server. Thus, the Progress_Rep will invoke ASSIGN-SEQUENCE (Figure A-11, line A6).

By Lemma B.20, some Progress_Rep will have enough time to globally order the new update if ASSIGN-SEQUENCE and THRESHOLD-SIGN complete in bounded time (where THRESHOLD-SIGN is invoked both during ASSIGN-SEQUENCE and at the non-leader sites upon receiving the Proposal). By Lemma B.32, ASSIGN-SEQUENCE will complete in bounded finite time, and by Lemma B.33, THRESHOLD-SIGN will be invoked by all stable servers at the non-leader sites. By Claim B.3, THRESHOLD-SIGN completes in bounded finite time in this case. Thus, the Progress_Rep will globally order the update for sequence number $(max\_stable\_seq + 1)$. It will then execute the update and make global progress, completing the proof. □