SCALING BYZANTINE REPLICATION TO WIDE-AREA NETWORKS

by

John W. Lane

A dissertation submitted to The Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

October, 2008

Abstract

As network environments become increasingly hostile, even well secured distributed systems are likely to suffer at least one compromised subcomponent. During the past several years, Byzantine fault-tolerant replication has emerged as a promising technique for constructing intrusion-tolerant systems that function correctly even when an attacker controls part of the system. Prior to our work, Byzantine replication systems were based on flat (nonhierarchical) protocols and offered high performance only when deployed on local-area networks.

This dissertation presents the first two hierarchical Byzantine fault-tolerant replication architectures that scale to systems that span multiple wide-area sites. Our first architecture, Steward, confines the effects of a 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 commodity hardware. Our second architecture improves upon Steward by providing customizability of the fault tolerance approach used within each site and on the wide area and by including new optimizations. Prototype implementations are evaluated in several network topologies and compared with the previous state of the art. The experimental results show an order of magnitude improvement over flat Byzantine replication protocols in typical wide-area deployments.

Adviser: Yair Amir
Readers: Scott Smith
Cristina Nita-Rotaru

Acknowledgements

I especially want to thank my adviser and collaborator, Yair Amir, for his help and, even more importantly, encouragement and advice. Of all of the people whom I encountered while working and studying, Yair showed the most interest in my future. In fact, he convinced me to attend the Ph.D. program.

I also want to thank the other people with whom I have collaborated on this work. In particular, I want to thank Jonathan Kirsch. We worked together on all of the parts of this dissertation, and he is an exceptional researcher and a great collaborator and friend. Other collaborators include Brian Coan (who also provided a great opportunity for me when I interned at Telcordia), Claudiu Danilov, Danny Dolev (who kindly visited us to help design Steward), Josh Olsen, Cristina Nita-Rotaru (who I also want to thank for being a member of my dissertation committee), and David Zage.

I thank my friends and family for providing the emotional support that all graduate students need – especially those who are as test-phobic as I am. Thanks to: Dan Young, my best friend, for countless long telephone conversations about everything. Nilo Rivera for listening to me go on and on about Byzantine fault-tolerant state machine replication over about a thousand lunches. Raluca Musaloiu-Elefteri for always being a friendly face (and keeping our computers working). Claudiu Danilov for easing my transition into the lab and gluing everything together during my first years there.

Many other people at Johns Hopkins played an important role in my Ph.D. I thank Jonathan Shapiro for his part in my acceptance to the program, Scott Smith for being a dissertation reader (and for the PL class), Andreas Terzis for working with me on one of my qualifying projects, and the many people who work in the Computer Science Department

supporting the students.

I owe a great deal to Paul Fitzgerald, the best of my many friends from my past life in neuroscience. He kept me grounded during the last five years. I also thank Kathy Downey for being there when I needed someone the most.

I want to thank the members of my family, and especially my grandmother Marjory Ward and my great aunt Grace Ward, for providing a haven in Cape Cod far removed from the stresses of school.

Lastly, and most importantly, I want to thank my parents, John and Carol Lane, for providing me with a great home and encouraging (and putting up with) all of my childhood "interests." This dissertation is a testament to their understanding and hard work.

During the time that I was a graduate student, I received support from the Defense Advanced Research Projects Agency (grant FA8750-04-2-0232) and from the National Science Foundation (grant 0716620).

Dedication

This dissertation is dedicated to my grandfather, William F. Ward, an actuary who took great pleasure in using early computers.

Contents

List of Tables

List of Figures

Chapter 1
Introduction

Computer compromises have become a widespread and often highly publicized occurrence [1,2]. Researchers estimate that attackers gain control of hundreds of thousands of computers each day [3]. Some of these machines either contain sensitive information or are part of a system that provides a critical government or financial service. For example, recent incidents involving government computers have deservedly generated significant concern [1,4]. Although administrators spend a great deal of effort to secure such machines, we have seen firsthand that they can still be compromised. Even if an attacker cannot penetrate a computer remotely, he may be able to compromise the machine via physical access. The evidence suggests that for the foreseeable future, in spite of the effort that has been devoted to computer security, computer compromises will remain widespread.

Given the prevalence of security exploits, it is not surprising that there has been growing interest in intrusion-tolerant systems. These systems mitigate the damage that an attacker causes after a security breach occurs. Such systems, including the work presented in this dissertation, are designed under the assumption that attacks on subcomponents of the system will sometimes succeed [5]. An intrusion-tolerant system must continue to provide usable functionality even when an attacker controls part of it. We refer to these systems as survivable. Survivable systems include those for network routing [6,7], group communication [8,9], and state machine replication [10–12]. Such systems typically comprise several computers that work together to provide a service. Survivable systems are designed so that even if some of the computers constituting the system behave maliciously, they cannot cause the entire system to fail.

This dissertation presents the first intrusion-tolerant state machine replication architectures that scale to wide-area networks. Our architectures can be used to construct a wide variety of survivable, distributed services, including systems for banking and trading, military command and control, and emergency communications. In general, the architectures are applicable to any service that can be implemented as a deterministic state machine. The service is duplicated on many servers, called replicas, that are distributed across the Internet. Since the system stores redundant copies of the service's state, it can continue to function even when some replicas fail (as long as a threshold fraction of servers remain operational). A client submits queries to a server or set of servers located close to it geographically. Servers answer read-only queries, which do not modify the replicated state, as soon as the query is received. Write queries, or updates, which change the replicated state, must be handled with care to ensure that the replicas remain consistent and that users do not retrieve conflicting data from different servers.

State machine replication [13] preserves data consistency by totally ordering all updates. Many protocols, including ours, use a leader to propose a possible ordering. The replicas, which are modeled as deterministic state machines, apply updates in the same agreed upon order. Since replicas execute the same stream of ordered updates, they move through the same sequence of states, and remain consistent. State machine replication systems for use on the Internet should continue to function even during network partitions and merges and replica crashes and recoveries. We refer to this class of problems as benign faults. Replicas are guaranteed to remain consistent and to execute queries as long as some minimal number of replicas remains functional and able to exchange messages.

Large-scale state machine replication systems deployed on the Internet may run on tens or even hundreds of physical computers. These machines are located in different areas, and each is vulnerable to attack. As replication systems grow in size, the probability that an attacker will compromise one of the replicas through a software vulnerability or physical access grows. Systems that tolerate only benign faults, where servers can become disconnected or fail, do not guarantee correctness if a sophisticated attacker controls a replica. In this case, the replica essentially becomes an inside attacker with the cryptographic identity of a protocol participant and plays the global ordering pro-

tocol in a malicious manner in an attempt to cause data inconsistency or halt progress. These attacks are referred to as Byzantine [14, 15], and a compromised replica is referred to as a Byzantine fault.

Next generation wide-area replication systems should be designed to tolerate a limited number of compromised replicas. Our replication architectures provide a means to construct high performance replicated systems that survive even when an attacker controls some of the computers in the system. Thus, the work presented in this dissertation is a significant step towards developing survivable systems in an imperfectly secure world.

1.1 Survivable Wide-Area Byzantine Fault Tolerant Replication: Steward and the Composable Architecture

During the past few years, we developed Steward [16, 17] (Survivable Technology for Wide-Area Replication), the first Byzantine fault-tolerant replication architecture that scales to wide-area networks. The system is presented in Chapter 3. Steward provides a state machine replication service. Our tests demonstrated an order of magnitude performance improvement compared to the previous state-of-the-art, BFT [10], in typical wide-area environments. Chapter 3 includes a rigorous correctness proof of the Steward system. Steward is also the first Byzantine fault-tolerant replication system to use hierarchy to scale to wide-area networks.

Steward is designed for use in environments where many local-area sites are distributed across a wide-area network, such as the Internet. Each site contains several physical machines that are connected by a high-bandwidth, low-latency, local-area network. Therefore inter-machine communication is fast with respect to both the time that it takes for messages to move between machines and the amount of data that can be sent. The sites are connected by wide-area links, which have much lower bandwidth and higher latency. This environment and the associated topological structure are typical of real-world deployments. Steward exploits the difference in the cost of intra and inter-site communication.

Steward is based on constructing *logical machines* from the groups of physical machines within each site. The logical machines execute a global ordering protocol among themselves, and local-area protocols ensure that the individual machines act as a single entity by masking up to f malicious

servers out of a total of $3f + 1$ servers within the site. The use of logical machines reduces the number of messages sent across wide-area links, which improves performance. In fact, a reduction in wide-area message complexity is the primary reason that Steward scales so well. Since each site functions as a single logical entity and flat replication protocols use all-to-all message exchange, message complexity is reduced from O(N^2) to O(S^2) (where N is the number of servers in a flat architecture, and S is the number of sites in Steward). The reduction in message complexity translates directly to a reduction in wide-area bandwidth requirements. This is important because, as we show in Section 3.5, the throughput of Byzantine wide-area replication systems is often bandwidth limited.

Steward uses several Byzantine fault-tolerant local protocols and a benign fault-tolerant wide-area protocol. The protocol used to locally agree on a proposed ordering of updates is similar to BFT [10], the first practical Byzantine fault-tolerant state machine replication protocol for local-area networks. The wide-area protocol is similar to Paxos [18], a benign fault-tolerant state machine replication protocol. Paxos and BFT both use a leader server that coordinates the protocol and supporting servers that monitor the leader. The leader proposes an ordering on updates, and the supporting servers vote to accept consistent proposals. If the leader fails, the remaining connected servers attempt to install a new leader, a procedure called a view change. View changes are carefully designed so that the new leader does not violate orderings established by the old leader. The manner in which view changes provide consistency is one of the most fascinating parts of these protocols. (BFT and Paxos are described briefly in Chapter 2.) Since Steward's local-area protocols are Byzantine fault-tolerant, they effectively transform each site into a single trusted participant in the wide-area protocol. As long as no more than f servers within a site are Byzantine, the sites may crash or partition, but they will follow the wide-area protocol.

Each time a Steward logical machine sends an outgoing wide-area message, the servers within the corresponding site must agree on the contents of this message. Steward uses distinct protocols to generate each type of wide-area message. Correct servers within a site do not necessarily attempt to act in the same manner with respect to the global protocol. For example, one correct server may decide that the current global leader site is faulty and attempt to elect a new global leader, while

the other correct servers in the same site continue to support the current global leader. Ensuring the correctness of the global protocol in the presence of such divergences is complicated and difficult. A server that attempts to initiate a wide-area message must obtain validating messages from other correct servers. This prevents a malicious server from misrepresenting the logical machine by unilaterally generating a wide-area protocol message.

Since Steward uses a benign fault-tolerant protocol on the wide area, it may fail to preserve consistency if an adversary gains control of a single site by compromising more than f servers within a site. A site compromise is likely to occur if an attacker exploits correlated vulnerabilities among the servers within a site (e.g., by gaining physical access to the site). Perhaps the most natural solution to this problem is to replace the benign fault-tolerant replication protocol that runs among the logical machines with a Byzantine replication protocol. This can be accomplished using a similar approach to the one used in Steward. However, a series of new local protocols would need to be developed in order to generate the new messages associated with the Byzantine protocol. Therefore, Steward is inherently difficult to customize.

In an effort to address Steward's limitations, we developed the composable architecture [19], which allows customizability of the fault tolerance approach used within and among the sites. The composable architecture is designed for use in the same wide-area environment for which Steward is designed. The composable architecture improves upon Steward in a variety of practical ways. First, it enables the system administrator to run a Byzantine fault-tolerant wide-area protocol, which allows the system to survive a complete site compromise. Second, it includes a Byzantine fault-tolerant link protocol that provides efficient communication between logical machines. Third, the composable architecture includes new performance optimizations that mitigate the cost of customizability.

The optimized composable architecture achieved 340 updates per second in an 80 server, Byzantine fault-tolerant deployment, with 100 ms latency between sites. This is an improvement of four times over a comparable native Steward deployment without the new optimizations developed for the composable architecture. When the optimizations are applied to Steward, Steward slightly outperforms the composable architecture. The performance of both optimized hierarchical architec-

tures exceeds that of many common benign fault-tolerant protocols. To put this into perspective, state machine replication, the service that our systems provide, is often considered to be too costly to use in systems that have more than ten or twenty servers, even in a local-area network. Our research shows that, in fact, it is possible to build large-scale Byzantine fault-tolerant systems for the Internet that provide performance that surpasses what many believe is possible for much smaller scale systems offering only benign fault tolerance.

Although the composable architecture shares many high-level similarities with Steward, its underlying structure is fundamentally different. The logical machines in the two architectures are constructed in completely different ways. Steward uses a more efficient method to build logical machines, which reduces the number of expensive operations that need to be executed each time an update is globally ordered. The composable architecture trades performance for customizability and simplicity.

In the composable architecture, servers within a site agree on the order in which they will process wide-area protocol messages. Since all correct servers that constitute a logical machine are actually replicas of each other (with respect to the wide-area protocol), and proceed through the same states, they are guaranteed to attempt to send exactly the same wide-area protocol messages. As a result, servers do not need to use a separate protocol to agree on the contents of each outgoing wide-area message as they do in Steward. In fact, the composable architecture uses a standard local state machine replication protocol to implement its logical machines.

Each approach has trade-offs and benefits. To summarize, servers in a Steward logical machine implement an agreement protocol on outgoing messages leaving the site and servers in the composable architecture agree on the order in which to process incoming wide-area messages. In general, the approach used in Steward has the potential to offer better performance, which is why we selected it initially. The servers in Steward logical machines apply wide-area messages as soon as the message is received, in any order, and only need to agree on the content of certain types of outgoing messages. In contrast, in the composable architecture, logical machines must agree on an ordering for *all* incoming messages. The performance difference stems from the fact that there are many more incoming messages than outgoing messages that must be agreed upon. We note that, in

practice, when optimizations are used, Steward has only a small performance advantage. Moreover, the composable architecture surpasses Steward in both customizability and simplicity.

This dissertation demonstrates the advantages that hierarchical, logical-machine based Byzantine fault-tolerant replication architectures offer in comparison to flat (non-hierarchical) Byzantine fault-tolerant protocols. Our hierarchical systems, regardless of the way in which the logical machines are constructed, offer superior scalability and are especially well suited to common Internet environments. Most importantly, our state machine replication architectures offer practical levels of performance, and thereby, for the first time, show that wide-area Byzantine fault-tolerant replication systems are feasible.

1.1.1 Model and Service Properties Overview

Steward and the composable architecture have similar models and service properties. Both architectures are designed for use on networks that may partition into multiple disconnected components and subsequently remerge. Both use digital signatures and threshold cryptography for authentication, and assume that the adversary cannot subvert these mechanisms. While the two architectures differ in the fault tolerance that they can provide, both can be configured to tolerate Byzantine faults, where faulty servers send arbitrary messages.

Both architectures guarantee two correctness criteria: *safety* and *liveness*. Safety means that two correct servers remain consistent replicas of each other. Liveness means that updates will eventually be executed. Steward and the composable architecture guarantee safety and liveness only if certain assumptions hold. We discuss this in more detail in the following two paragraphs.

Both architectures provide the following safety condition: if two correct servers execute an update with the same sequence number, then these updates are identical. Safety does not depend on synchrony assumptions. However, it is guaranteed only if fewer than a threshold number of servers become compromised. Since the two architectures provide different levels of fault tolerance, the number of server compromises that each can survive differs. For example, when appropriately configured, the composable architecture can tolerate a larger fraction of compromised servers than Steward. This leads to differences in the assumptions required for each architecture to guarantee

safety. A detailed description of the fault-tolerance provided by each architecture is included in the chapter in which it is presented.

Liveness is guaranteed only if the system meets a certain level of stability. The system is stable if a sufficient set of correct servers can exchange messages within a bounded amount of time. Thus, liveness depends on synchrony (i.e., timing) assumptions with respect to message delay. Steward and the composable architecture provide somewhat different liveness guarantees. This stems from differences in how wide-area messages are sent between sites and differences in how a server decides to install a new local or global leader. Steward servers decide to change leaders based solely on global progress, while servers in the composable architecture base their decisions on both global progress and local progress. As a consequence, Steward guarantees that an update will be executed eventually, while the composable architecture guarantees that a particular update received by a correct server will be executed eventually. We present the precise liveness guarantee that each system provides in their respective chapters.

Although both architectures provide similar correctness guarantees, they are not identical. Therefore, we chose to present a detailed model and service guarantees for each architecture in the chapter that describes that architecture.

1.2 Dissertation Organization

The remainder of this dissertation is organized as follows:

- Chapter 2 presents an overview of related work. It includes a summary of three important background topics that play a central role in our architectures: Paxos, BFT, and threshold cryptography.

- Chapter 3 presents a description of Steward including sections on the system model, service properties, protocol, performance, and correctness proof.

- Chapter 4 presents the composable architecture including sections describing the system model and service properties, system architecture, Byzantine link protocol, and correctness proof.

- Chapter 5 explains how our architectures can be used to construct a survivable system. It also sheds light on practical issues associated with Byzantine fault-tolerance.

- Chapter 6 summarizes the dissertation.

Chapters 3 and 4 have parallel structures. They are both largely self-contained and each can be read and on its own after completing this chapter (i.e., Chapter 1) and the background material in Section 2.2. The proofs and proof sketches presented in Chapters 3 and 4 refer only to service properties and terminology presented in the same chapter.

The material in Chapter 5 applies to both architectures, with a focus on using the composable architecture to develop a survivable replicated system. In addition to describing how a survivable system can be constructed using our architectures, Chapter 5 illustrates common difficulties that are likely to arise when using replication for survivability. Therefore, readers may find it beneficial to skim this chapter as they peruse the detailed description of Steward and the composable architecture.

Chapter 2

Related Work and Background

The concepts, algorithms, and techniques that form the foundation of the replication architectures presented in this dissertation have a long and rich history. Work that is related to our systems ranges from theory that elucidates fundamental principles to systems research that has yielded high performance, practical implementations. In Section 2.1 of this chapter, we present an overview of related work. Then, in Section 2.2, we present a more detailed description of the protocols and algorithms that our systems use.

2.1 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 [20]. The strongest fault model that researchers consider is the Byzantine model [14, 15], 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 $f + 2$ [21].

Byzantine Group Communication: Related with our work are group communication systems resilient to Byzantine failures. Two such systems are Rampart [8] and SecureRing [9]. 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 [22–25], 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. The reliability and ordering protocol is based on the following idea: the initiator of a message computes its hash, signs it, and sends it to other members. The initiator needs to gather enough signatures from other the members before sending the actual message in order to guarantee the uniqueness of the content for that message across the network even in the presence of malicious participants.

Drabkin, et al. [26] describe a Byzantine version of the JazzEnsemble system, providing a formal definition of Byzantine virtual synchrony. The system uses the idea of fuzzy membership: each node is given an indication of how fuzzy the other group members are, with low fuzziness indicating a well-responding server and high fuzziness indicating a server that is not very responsive. Detection of Byzantine behavior in this context is encapsulated by fuzzy mute and fuzzy verbose failure detectors.

Replication with Benign Faults: The two-phase commit (2PC) protocol [27] 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 [28] overcomes some of the availability problems of 2PC, paying the price of an additional communication round. Paxos [18, 29] is a very robust algorithm for benign fault-tolerant replication and is described in Section 2.2.

Replication with Byzantine Faults: The replication architectures that we describe next provide the same service that our architectures provide: a state machine replication service that survives Byzantine faults. Unlike the systems presented in this dissertation, previous systems were not hierarchical and they performed best in small-scale local-area networks. The first practical Byzantine fault-tolerant replication protocol was Castro and Liskov's BFT [10], which is described in Section 2.2.

Doudou, et al. [30] decompose the problem of Byzantine fault-tolerant state machine replication via a series of abstractions. The replication problem is reduced to an atomic multicast protocol, which itself is composed of a reliable multicast component and a solution to the weak interactive

consistency problem. The latter uses a Byzantine failure detector for detecting mute processes (i.e., processes from which, from some time on, a correct process stops receiving messages).

Yin et al. [11] 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 $2f+1$. This system also provides a privacy firewall, which prevents a compromised server from divulging sensitive information.

Correia, et al. [31] reduce the number of replicas needed for state machine replication from $3f+1$ to $2f+1$ by augmenting the Byzantine, asynchronous model with a distributed trusted component, the Trusted Timely Computing Base (TTCB). The local TTCBs of the replication servers are assumed not to be malicious, and they communicate over a synchronous control network providing real-time delay guarantees. The TTCBs run a fault-tolerant protocol to assign an ordering to protocol messages, and these protocol messages are then exchanged over the asynchronous payload network.

Martin and Alvisi [12] proposed 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.

The ShowByz system of Rodrigues et al. [32] 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. Each object is implemented by two BFT replica groups, which run a primary-backup protocol to allow progress to continue even if one of the groups halts.

Zyzzyva [33] uses speculative execution to reduce the cost of Byzantine fault tolerant state machine 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 in our architectures.

Quorum Systems with Byzantine Fault-Tolerance: Quorum systems obtain Byzantine fault tolerance by applying quorum replication methods. Examples of such systems include Phalanx [34,35] and Fleet [36,37]. 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. [38] 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. The HQ protocol [39] combines the use of quorum replication with Byzantine fault-tolerant agreement, using a more lightweight quorum-based protocol during normal-case operation and BFT to resolve contention when it arises.

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. Verissimo et al. propose such a hybrid approach [40,41], where synchronous, trusted nodes provide strong global timing guarantees. This inspired the Survivable Spread [42] work, where a few trusted nodes (at least one per site) are assumed impenetrable, but are not synchronous, may crash and recover, and may experience network partitions and merges. These trusted nodes were implemented by Boeing Secure Network Server (SNS) boxes, limited computers designed to be impenetrable. Both the hybrid approach and the architectures presented in this dissertation can scale Byzantine replication to wide area networks. The hybrid approach makes stronger assumptions, while our approach pays more hardware and computational costs.

State Machine Replication and Logical Machines: Lamport [13] and Schneider [43] introduced and popularized state machine replication, where deterministic replicas execute a totally ordered stream of events that cause state transitions. Therefore, all replicas proceed through exactly the same states. This technique can be used to implement replicated information access systems,

databases, and other services. The state machine approach has been used in many systems, including our composable architecture, to construct fault-tolerant logical machines out of collections of physical machines. We mention several of these systems here.

Schlichting and Schneider [44] 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. Benign fault-tolerant protocols can thus safely run on top of these logical processors. Unlike in our architectures, in which a site is live unless f+1 of its computers fail, the k-fail-stop processor described in [44] halts when even one of its constituent processors fails.

The Delta-4 system [45] uses an intrusion-tolerant architecture and provides services for data authentication, storage, and authorization. Like our composable architecture, the system constructs logical entities out of multiple physical machines via the state machine approach; it also employs protocols to make communication between the logical entities efficient. However, these protocols assume that the communicating parties are fail-silent, whereas the composable architecture constructs Byzantine fault-tolerant links between logical entities.

The Voltan system of Brasileiro, et al. [46] uses the state machine approach to construct two-processor fail-silent nodes that either work correctly or become silent if an internal failure of one of the processes is detected. Each message send from one logical node to another requires sending four physical messages over the network, reducing the system's applicability to bandwidth-constrained wide-area environments.

The Starfish system of Kihlstrom and Narasimhan [47] builds an intrusion-tolerant middleware service by using a hierarchical membership structure and end-to-end intrusion detection. The system uses a central, hardened core that offers strong security guarantees. The core is augmented by "arms," with weaker security guarantees, that can be removed in the case of a security breach.

The Thema system of Merideth, et al. [48] uses state machine replication to build Byzantine fault-tolerant Web Services. Standard Web Service clients access the Byzantine fault-tolerant service using a client library. Thema allows Byzantine fault-tolerant services to safely access non-replicated Web Services.

The MAFTIA system of Verissimo, et al. [49] uses architectural hybridization to build mechanisms for intrusion tolerance by transforming untrusted components into trusted components. The hybrid architecture is built in the wormhole model [41], where different parts of the system operate under different fault assumptions and are thus resilient to different types of attack. For example, if trusted components (e.g., a reliable channel, a processor whose results can be trusted) are available, the system can be configured to run protocols that take advantage of them to achieve increased performance (e.g., [40]). In contrast, our systems assume all components are untrusted.

Fault-tolerant CORBA: State machine replication has also been used to increase the fault-tolerance and availability of CORBA services.

The Object Group Service (OGS) of Felber, et al. [50] provides a composable, modular architecture for replicating CORBA objects. The OGS implements several component CORBA services, such as group multicast, group membership, and distributed consensus, which are then composed to implement group communication services; this group communication service is then used for replication.

The FTS system of Friedman and Hadad [51] uses active replication to construct a lightweight fault tolerance service for CORBA. The system survives network partitions, allowing updates in a single partition but allowing other partitions to remain alive until they reconnect.

The Immune system of Narasimhan, et al. [52] provides support for survivable CORBA applications by replicating both client and server objects. When a replicated client objects invokes an operation on a replicated server object, each client object sends a message to each server object via the SecureRing multicast protocol [9], and the servers employ majority voting to mask faulty behavior; the responses are sent from server to client in similar fashion. While the logical machines in our composable architecture could use SecureRing to communicate with one another (with one group for each pair of neighboring logical machines), doing so would result in many redundant messages being sent over the wide-area network during normal-case operation, greatly limiting performance.

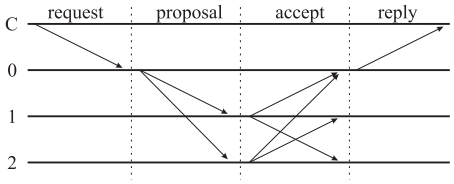

Figure 2.1: Common case operation of the Paxos algorithm when $f = 1$. Server 0 is the current leader.

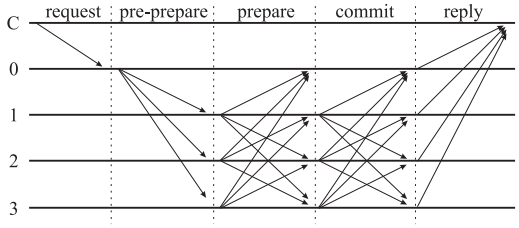

Figure 2.2: Common case operation of the BFT algorithm when $f = 1$. Server 0 is the current leader.

2.2 Background: Paxos, BFT, and RSA Threshold Signatures

Our work builds directly on two state machine replication protocols: Paxos and BFT. Both protocols establish an agreed order on a sequence of updates that cause state transitions in the replicas. Paxos tolerates benign faults, while BFT tolerates Byzantine faults. Our work directly uses threshold cryptography, which enables a site to produce a single RSA signature for each of its wide-area protocol messages. This section presents a brief overview of these three technologies.

2.2.1 Paxos Overview

Paxos [18,29] 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, a single reply needs to be delivered to the client.

In the common case (Figure 2.1), 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.

2.2.2 BFT Overview

The BFT [53] 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 (Figure 2.2), 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.

2.2.3 Threshold Digital Signatures

Threshold cryptography [54] distributes trust among a group of participants to protect information (e.g., threshold secret sharing [55]) or computation (e.g., threshold digital signatures [56]).

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, such that any set of k servers can pool their shares to generate a valid threshold signature on a message, m, while any set of fewer than k servers is unable to do so. Each server uses its key share to generate a partial signature on m and sends the partial signature to a *combiner* server, which combines the partial signatures into a threshold signature on m. The threshold signature is verified using the public key corresponding to the divided private key. One important property

provided by some threshold signature schemes is *verifiable secret sharing* [57], which guarantees the robustness [58] of the threshold signature generation by allowing 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).

A representative example of practical threshold digital signature schemes is the RSA Shoup [56] scheme, which allows participants to generate threshold signatures based on the standard RSA [59] digital signature. It provides verifiable secret sharing, which is critical in achieving signature robustness in the Byzantine environment we consider.

Chapter 3

Steward: Survivable Technology for Wide-Area Replication

This chapter presents Steward [16, 17, 60], the first hierarchical Byzantine fault-tolerant replication architecture that scales to systems that span multiple wide-area sites. It is joint work with Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, Cristina Nita-Rotaru, Josh Olsen, and David Zage.

3.1 Steward Overview

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

Steward uses threshold signatures to prevent malicious replicas from misrepresenting decisions that took place in their site. Messages sent between servers in 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 wide-area bandwidth associated with sending multiple individual signatures.

Steward's hierarchical architecture reduces the wide-area message complexity 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 (f is the upper bound on the number of malicious servers).

Steward allows read-only queries to be performed 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 [61], the common semantics provided by database products. Serializability is a weaker guarantee than the linearizability semantics [62] provided by some existing flat protocols (e.g., [10]). 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 these benefits 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 is able to overcome up to f malicious servers in *each* site. We believe that given the cost associated with computers today, this requirement is reasonable.

We demonstrate 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. We also show that Steward exhibits performance comparable to 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. We include a description of the red-team experiment in Section 3.5.

While solutions previously existed for Byzantine and benign fault-tolerant replication and for

providing practical threshold signatures, these concepts have never been used in a provably correct, hierarchical architecture that scales Byzantine fault-tolerant replication to large, wide-area systems. This chapter presents the design, implementation, and proofs of correctness for such an architecture.

The main contributions of our work on Steward are:

1. It presents the first hierarchical architecture and algorithm that scales Byzantine fault-tolerant replication to wide-area networks.

2. It provides a complete proof of correctness for this algorithm, demonstrating its safety and liveness propertes.

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.

3.2 System Model

Servers are implemented as deterministic state machines [13, 43]. 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 all sites may have at most f faulty servers.

Clients are distinguished by unique identifiers. Clients send updates to servers within their local site and receive responses from these servers. Each update is uniquely identified by a pair consisting of the identifier of the client that generated the update and a unique, monotonically increasing logical timestamp. Clients propose updates sequentially: a client may propose an update with timestamp $i + 1$ only after it receives a reply for an update with timestamp i.

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 secret shares.

3.3 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 correct server in the local site, and the correct 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 of one site is connected to every correct server of 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 3.3.1 S1 - SAFETY: *If two correct servers execute the i^{th} update, then these updates are identical.*

DEFINITION 3.3.2 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 [61]. Alternatively, a client can specify that its query should be linearizable [62], 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 [63], 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 (1) correct and (2) 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 (1) stable with respect to T and (2) 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 3.3.3 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.*

3.4 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 (e.g., BFT) among all *servers* in the system, Steward runs a Paxos-like 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 local (i.e., 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, the local protocols allow each site to emulate the behavior of a correct Paxos participant in the global protocol.

Similar to the rotating coordinator scheme used in BFT, the local, intra-site 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 is responsible for (1) disseminating messages in the global protocol originating from the local site to the other site representatives and (2) receiving global messages and distributing them to the local servers. If the site representative is suspected to be Byzantine, the other servers in the site run a local view change protocol to replace the representative and install a new view.

While Paxos uses a rotating leader server to coordinate the protocol, Steward uses a rotating *leader site* to coordinate the global protocol; the global protocol thus 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. As will

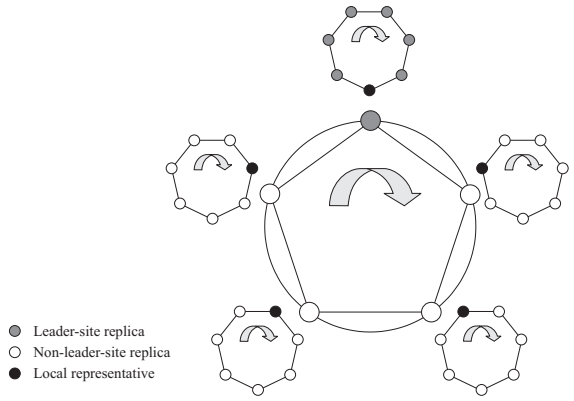

Figure 3.1: A Steward system having five sites with seven servers in each site. Each smaller, local wheel rotates when its representative is suspected of being faulty. The larger, global wheel rotates when the leader site is suspected to have partitioned away.

be described below, 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.

Figure 3.1 depicts a Steward system with five sites. As described above, the coordinators of the local and global protocols (i.e., site representatives and the leader site, respectively) are replaced when failures occur. Intuitively, the system proceeds through different configurations of representatives and leader sites via two levels of rotating "configuration wheels," one for each level of the hierarchy. At the local level, an intra-site wheel rotates when the representative of a site is suspected of being faulty. At the global level, an inter-site wheel rotates when enough sites decide that the current leader site has partitioned away. Servers within a site use the absence of global progress (as detected by timeout mechanisms) to trigger the appropriate view changes.

In the remainder of this section, we present the local and global protocols that Steward uses to totally order client updates. We first present the data structures and messages 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.

3.4.1 Data Structures and Message Types

To facilitate the presentation of Steward's local and global protocols, we first present the message types used by the protocols (Figure 3.2) and the data structures maintained by each server

```
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)
```

Figure 3.2: Message types used in the global and local protocols.

(Figure 3.3).

As listed in Figure 3.3, each server maintains variables for the global, Paxos-like protocol and the local, intra-site, Byzantine fault-tolerant protocols; we say that a server's state is divided into the *global context* and the *local context*, respectively, reflecting our hierarchical architecture. Within the global context, a server maintains (1) the state of its current global view and (2) a *Global_History*,

```
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
```

Figure 3.3: Global and Local data structures maintained by each server.

```
boolean Valid(message):
A1.   if message has threshold RSA signature S
A2.     if NOT VERIFY(S)
A3.       return FALSE
A4.   if message has RSA-signature S
A5.     if NOT VERIFY(S)
A6.       return FALSE
A7.   if message contains update with client signature C
A8.     if NOT VERIFY(C)
A9.       return FALSE
A10.  if message.sender is in Corrupted_Server_List
A11.    return FALSE
A12.  return TRUE
```

Figure 3.4: Validity checks run on each incoming message. Invalid messages are discarded.

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

```
boolean Conflict(message):
  case message
A1.  Proposal((site_id, gv, lv, seq, u):
A2.    if gv ≠ Global_view
A3.      return TRUE
A4.    if server in leader site
A5.      return TRUE
A6.    if Global_History[seq].Global_Ordered_Update(gv', seq, u') exists
A7.      if (u' ≠ u) or (gv' > gv)
A8.        return TRUE
A9.    if not Is-Contiguous(seq)
A10.     return TRUE
A11.   if not In-Window(seq)
A12.     return TRUE
A13. return FALSE

B1.  Accept(site_id, gv, lv, seq, digest):
B2.    if gv ≠ Global_view
B3.      return TRUE
B4.    if (Global_History[seq].Proposal(*, *, *, seq, u') exists) and (Digest(u') ≠
digest)
B5.      return TRUE
B6.    if Global_History[seq].Global_Ordered_Update(gv', seq, u') exists
B7.      if (Digest(u') ≠ digest) or (gv' > gv)
B8.        return TRUE
B9.  return FALSE

C1.  Aru_Message(site_id, gv, site_aru):
C2.    if gv ≠ Global_view
C3.      return TRUE
C4.    if server in leader site
C5.      return TRUE
C6.  return FALSE

D1.  Request_Global_State(server_id, gv, lv, aru):
D2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
D3.      return TRUE
D4.    if server_id ≠ lv mod num_servers_in_site
D5.      return TRUE
D6.  return FALSE

E1.  Global_Server_State(server_id, gv, lv, seq, state_set):
E2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
E3.      return TRUE
E4.    if not representative
E5.      return TRUE
E6.    if entries in state_set are not contiguous above seq
E7.      return TRUE
E8.   return FALSE

F1.  Global_Collected_Servers_State(server_id, gv, lv, gss_set):
F2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
F3.      return TRUE
F4.    if each message in gss_set is not contiguous above invocation_seq
F5.      return TRUE

G1.  Global_Constraint(site_id, gv, seq, state_set):
G2.    if gv ≠ Global_view
G3.      return TRUE
G4.    if server not in leader site
G5.      return TRUE
G6.  return FALSE

H1.  Collected_Global_Constraints(server_id, gv, lv, gc_set):
H2.    if gv ≠ Global_view
H3.      return TRUE
H4.    aru ← Extract-Aru(gc_set)
H5.    if Global_aru < aru
H6.      return TRUE
H7.  return FALSE
```

Figure 3.5: 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
A1.  Pre-Prepare(server_id, gv, lv, seq, u):
A2.    if not (globally_constrained && locally_constrained)
A3.      return TRUE
A4.    if server_id ≠ lv mod num_servers_in_site
A5.      return TRUE
A6.    if (gv ≠ Global_view) or (lv ≠ Local_view)
A7.      return TRUE
A8.    if Local_History[seq].Pre-Prepare(server_id, gv, lv, seq, u') exists and u' ≠ u
A9.      return TRUE
A10    if Local_History[seq].Prepare_Certificate.Pre-Prepare(gv, lv', seq, u') exists
and u' ≠ u
A11.     return TRUE
A12.   if Local_History[seq].Proposal(site_id, gv, lv', u') exists
A13.     if (u' ≠ u) or (lv' > lv)
A14.       return TRUE
A15.   if Global_History[seq].Proposal(site_id, gv', lv', seq, u') exists
A16.     if (u' ≠ u) or (gv' > gv)
A17.       return TRUE
A18.   if Global_History[seq].Globally_Ordered_Update(*, seq, u') exists
A19.     if (u' ≠ u)
A20.       return TRUE
A21.   if not Is-Contiguous(seq)
A22.     return TRUE
A23.   if not In-Window(seq)
A24.     return TRUE
A25.   if u is bound to seq' in Local_History or Global_History
A26.     return TRUE
A27.  return FALSE

B1.  Prepare(server_id, gv, lv, seq, digest):
B2.    if not (globally_constrained && locally_constrained)
B3.      return TRUE
B4.    if (gv ≠ Global_view) or (lv ≠ Local_view)
B5.      return TRUE
B6.    if Local_History[seq].Pre-Prepare(server_id', gv, lv, seq, u) exists
B7.      if digest ≠ Digest(u)
B8.        return TRUE
B9.    if Local_History[seq].Prepare_Certificate.Pre-Prepare(gv, lv', seq, u) exists
B10.     if (digest ≠ Digest(u)) or (lv' > lv)
B11.       return TRUE
B12.   if Local_History[seq].Proposal(gv, lv', seq, u) exists
B13.     if (digest ≠ Digest(u)) or (lv' > lv)
B14.       return TRUE
B15.  return FALSE

C1.  Request_Local_State(server_id, gv, lv, aru):
C2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
C3.      return TRUE
C4.    if server_id ≠ lv mod num_servers_in_site
C5.      return TRUE
C6.  return FALSE

D1.  Local_Server_State(server_id, gv, lv, seq, state_set):
D2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
D3.      return TRUE
D4.    if not representative
D5.      return TRUE
D6.    if entries in state_set are not contiguous above seq
D7.      return TRUE
D8.  return FALSE

E1.  Local_Collected_Servers_State(server_id, gv, lv, lss_set):
E2.    if (gv ≠ Global_view) or (lv ≠ Local_view)
E3.      return TRUE
E4.    if each message in lss_set is not contiguous above invocation_seq
E5.      return TRUE
E6.  return FALSE
```

Figure 3.6: Conflict checks run on incoming messages used in the local context. Messages that conflict with a server's current local state are discarded.

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 (Figure 3.4). The server then verifies that the message does not conflict with anything already in its data structures; these *conflicts* (Figures 3.5 and 3.6) are defined by our protocol and ensure that the servers preserve the critical safety property. If a message passes both the validity and conflict checks, the server applies the message to its local or global data structure according to a set of rules (Figures 3.7 and 3.8, respectively). These rules invoke several predicate functions (Figure 3.9) to determine if and how a message should be applied.

```
/* Notation: <== means append */
UPDATE-LOCAL-DATA-STRUCTURES:
  case message:
A1.   Pre-Prepare(server_id, *, lv, seq, u):
A2.     if Local_History[seq].Pre-Prepare is empty
A3.       Local_History[seq].Pre-Prepare ← Pre-Prepare
A4.     else
A5.       ignore Pre-Prepare

B1.   Prepare(server_id, *, lv, seq, digest):
B2.     if Local_History[seq].Pre-Prepare is empty
B3.       ignore Prepare
B4.     if Local_History[seq].Prepare_List contains a Prepare with server_id
B5.       ignore Prepare
B6.     Local_History[seq].Prepare_List <== Prepare
B7.     if Prepare-Certificate-Ready(seq)
B8.       pre-prepare ← Local_History[seq].Pre-Prepare
B9.       PC ← Construct_Prepare_Certificate(pre-prepare, Local_History[seq].Prepare_List)
B10.      Local_History[seq].Prepare_Certificate ← PC

C1.   Partial_Sig(server_id, data, partial_sig, verification_proof, sig):
C2.     if Local_History.Partial_Sigs{ data }[Server_id] is empty
C3.       ignore Partial_Sig
C4.     Local_History.Partial_Sigs{ data }[server_id] ← Partial_Sig

D1.   Local_Collected_Server_State(gv, lv, Local_Server_State[]):
D2.     union ← Compute-Local-Union(Local_Collected_Server_State)
D3.     invocation_aru ← Extract-Invocation-Aru(Local_Server_State[])
D4.     max_local_entry ← Extract-Max-Local-Entry(Local_History[])
D5.     for each seq from (invocation_aru+1) to max_local_entry
D6.       if Local_History[seq].Prepare_Certificate(*, lv', seq, *) exists and lv' < lv
D7.         clear Local_History[seq].Prepare_Certificate
D8.       if Local_History[seq].Proposal(*, lv', seq, *) exists and lv' < lv
D9.         clear Local_History[seq].Proposal
D10.      if Local_History[seq].Pre-Prepare(*, lv', seq, *) exists and lv' < lv
D11.        clear Local_History[seq].Pre-Prepare
D12.    for each Prepare_Certificate(*, *, seq, *), PC, in union
D13.      if Local_History[seq].Prepare_Certificate is empty
D14.        Local_History[seq].Prepare_Certificate ← PC
D15.    for each Proposal(*, *, seq, *), P, in union
D16.      if Local_History[seq].Proposal is empty
D17.        Local_History[seq].Proposal ← P

E1.   New_Rep(site_id,lv):
E2.     if (lv > Latest_New_Rep[site_id])
E3.       Latest_New_Rep[site_id] ← New_Rep
E4.     Local_preinstalled_view ← Latest_New_Rep[Site_id]

F1.   Update(u):
F2.     SEND to all servers in site: Update(u)
F3.     if representative of non-leader site
F4.       SEND to representative of leader site: Update(u)
F5.     Add Update(u) to Update_Pool
```

Figure 3.7: 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 ≤ 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[ ⌊N/2⌋ + 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
```

Figure 3.8: Rules for applying a message to the Global_History data structure. The rules assume that Conflict(message) == FALSE

```
A1. boolean Globally-Ordered(seq):
A2.   if Global_History[seq].Globally_Ordered_Update is not empty
A3.     return TRUE
A4.   return FALSE

B1. boolean Globally-Ordered-Ready(seq):
B2.   if Global_History.Proposal[seq] contains a Proposal(site_id, gv, lv, seq, u)
B3.     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
B4.       return TRUE
B5.     if Global_History[seq].Accept_List contains a majority of distinct
B6.       Accept(site_id(i), gv', lv, seq, Digest(u)) with gv >= gv'
B7.       return TRUE
B8.   return FALSE

C1. boolean Prepare-Certificate-Ready(seq):
C2.   if Local_History.Proposal[seq] contains a Pre-Prepare(server_id, gv, lv, seq, u)
C3.     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)
C4.       return TRUE
C5.   return FALSE

D1. boolean In-Window(seq):
D2.   if Global_aru < seq ≤ Global_aru + W
D3.     return TRUE
D4.   else
D5.     return FALSE

E1. boolean Is-Contiguous(seq):
E2.   for i from Global_aru+1 to seq-1
E3.     if Global_History[seq].Prepare-Certificate == NULL and
E4.          Global_History[seq].Proposal == NULL and
E5.          Global_History[seq].Globally_Ordered_Update == NULL and
E6.          Local_History[seq].Prepare-Certificate == NULL and
E7.          Local_History[seq].Proposal == NULL
E8.       return FALSE
E9.   return TRUE
```

Figure 3.9: 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.

3.4.2 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 (Figure 3.10) and ASSIGN-SEQUENCE (Figure 3.11), which we describe below. Pseudocode for the global ordering protocol (ASSIGN-GLOBAL-ORDER) is listed in Figure 3.12. 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.

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 $\lfloor N/2 \rfloor$ Accept messages from distinct sites (where N 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.

```
THRESHOLD-SIGN(Data_s data, int server_id):
A1. Partial_Sig ← GENERATE-PARTIAL-SIG(data, server_id)
A2. SEND to all local servers: Partial_Sig

B1. Upon receiving a set, PSig_Set, of 2f+1 Partial_Sigs from distinct servers:
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.        SEND to all local servers: Corrupted_Server
B11.        continue to wait for more Partial_Sig messages
```

Figure 3.10: 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.   SEND to all local servers: 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.   SEND to all local servers: 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.     prepare_certificate ← Local_History[seq].Prepare_Certificate
C5.     pre-prepare ← prepare_certificate.Pre-Prepare
C6.     unsigned_proposal ← ConstructProposal(pre-prepare)
C7.     invoke THRESHOLD_SIGN(unsigned_proposal) //returns signed_proposal

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
```

Figure 3.11: ASSIGN-SEQUENCE Protocol, used to bind an update to a sequence number and produce a threshold-signed Proposal message.

```
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) //returns Proposal

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.     SEND to all local servers: Proposal
C5.   unsigned_accept ← Construct-Accept(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: Accept

E1. Upon receiving Accept(site_id, gv, lv, seq, Digest(u)):
E2.   Apply Accept to Global_History
E3.   if representative
E4.     SEND to all local servers: Accept
E5.   if Globally-Ordered-Ready(seq)
E6.     globally_ordered_update ← ConstructOrderedUpdate(seq)
E7.     Apply globally_ordered_update to Global_History
```

Figure 3.12: 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():
A1. u ← NULL
A2. if(Global_History[Global_seq].Proposal is not empty)
A3.    u ← Global_History[Global_seq].Proposal.Update
A4. else if(Local_History[Global_seq].Prepare_Certificate is not empty)
A5.    u ← Local_History[Global_seq].Prepare_Certificate.Update
A6. else if(Unconstrained_Updates is not empty)
A7.    u ← Unconstrained_Updates.Pop-Front()
A8. return u
```

Figure 3.13: 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.

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

Threshold-Sign: The THRESHOLD-SIGN intra-site protocol (Figure 3.10) generates a ($2f + 1$, $3f + 1$) threshold signature on a given message.[1] Upon invoking the protocol, a server generates a Partial_Signature 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_Signature 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

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

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 (Figure 3.11) 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 (Figure 3.13), which is invoked by the representative of the leader site during ASSIGN-GLOBAL-ORDER (Figure 3.12, 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.

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. From Figure 3.6, a Pre-Prepare(seq, u) 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(seq, u) to all servers.

```
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) //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
```

Figure 3.14: 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)

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 ⌊N/2⌋ + 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
```

Figure 3.15: 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.

3.4.3 View Changes

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

```
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^⌈Global_view/N⌉
```

Figure 3.16: 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 ← Construct-Bundle(GC_SET)
E4.     SEND to all in site: Collected_Global_Constraints
E5.     Apply Collected_Global_Constraints to Global_History
E6.     globally_constrained ← True

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

Figure 3.17: 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.

safety and liveness.

To ensure that the system can continue to 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

```
CONSTRUCT-LOCAL-CONSTRAINT(int seq):
A1. if representative
A2.   Request_Local_State ← 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 ← s
B3.   if (Pending_Proposal_Aru < s)
B4.     Request missing Proposals or Globally_Ordered_Update messages from representative
B5.   if (Pending_Proposal_Aru ≥ s)
B6.     Local_Server_State ← 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 ← 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 ≥ invocation_aru)
D4.       Apply Local_Collected_Servers_State to Local_History
D5.       locally_constrained ← True
D6.       return Local_Collected_Servers_State
```

Figure 3.18: 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.

servers replace the current representative. Similarly, when the Global_T timers of $2f + 1$ servers in a majority of sites expire, the sites replace the current leader site. Our timeout mechanism is described in more detail in Section 3.4.4.

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 acts conservatively and respects the established binding in the new view. This guarantee also applies to those Proposals that may have been constructed in a previous local view within the current global view.

Steward uses a *constraining* mechanism to enforce this conservative behavior. Before participating in the global ordering protocol, a correct server must become both *locally constrained* and

```
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) //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) //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) //Returns Aru_Message

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

Figure 3.19: 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.

globally constrained by completing the LOCAL-VIEW-CHANGE and GLOBAL-VIEW-CHANGE protocols (Figures 3.14 and 3.17, respectively). The local constraint mechanism ensures continuity across local views (when the site representative changes), and the global constraint mechanism ensures continuity across global views (when the leader site changes). Since the site representative coordinating the global ordering protocol may ignore the constraints imposed by previous views if it is faulty, *all* servers in the leader site become constrained, allowing them to monitor the representative's behavior and 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.

Leader Election: Steward uses two Byzantine fault-tolerant leader election protocols. Each site runs the LOCAL-VIEW-CHANGE protocol (Figure 3.14) to elect its representative, and the system

```
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) //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) //Returns Global_Constraint

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

Figure 3.20: 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):
A1. state_set ← ∅
A2. For each sequence number i from (seq + 1) to (Global_Aru + W):
A3.   if Local_History[i].Proposal, P, exists
A4.      state_set ← state_set ∪ P
A5.   else if Local_History[i].Prepare_Certificate, PC, exists:
A6.      state_set ← state_set ∪ PC
A7. return Local_Server_State(Server_id, gv, lv, seq, state_set)

Construct-Global-Server-State(seq):
B1. state_set ← ∅
B2. For each sequence number i from (seq + 1) to (Global_aru + W):
B3.   if Global_History[i].Globally_Ordered_Update, G, exists
B4.      state_set ← state_set ∪ G
B5.   else if Global_History[i].Proposal, P, exists:
B6.      state_set ← state_set ∪ P
B7.   else if Global_History[i].Prepare_Certificate, PC, exists:
B8.      state_set ← state_set ∪ PC
B9. return Global_Server_State(Server_id, gv, lv, seq, state_set)
```

Figure 3.21: 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.

runs the GLOBAL-LEADER-ELECTION protocol (Figure 3.15) to elect the leader site. Both leader election protocols provide two important properties necessary for liveness. Specifically, 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. We make use of these

```
// Assumption: all entries in css are from Global_view
Compute-Local-Union(Local_Collected_Servers_State css):
A1. union ← ∅
A2. css_unique ← Remove duplicate entries from css
A3. seq_list ← Sort entries in css_unique by increasing (seq, lv)

B1. For each item in seq_list
B2.   if any Proposal P
B3.     P* ← Proposal from latest local view
B4.     union ← union ∪ P*
B5.   else if any Prepare Certificate PC
B6.     PC* ← PC from latest local view
B7.     union ← union ∪ PC*
B8. return union

Compute-Global-Union(Global_Collected_Servers_State css):
C1. union ← ∅
C2. css_unique ← Remove duplicate entries from css
C3. seq_list ← Sort entries in css_unique by increasing (seq, gv, lv)

D1. For each item in seq_list
D2.   if any Globally_Ordered_Update
D3.     G* ← Globally_Ordered_Update with Proposal from latest view (gv, lv)
D4.     union ← union ∪ G*
D5.   else
D6.     MAX_GV ← global view of entry with latest global view
D7.     if any Proposal from MAX_GV
D8.       P* ← Proposal from MAX_GV and latest local view
D9.       union ← union ∪ P*
D10.    else if any Prepare Certificate PC from MAX_GV
D11.      PC* ← PC from MAX_GV and latest local view
D12.      union ← union ∪ PC*
D13. return union

Compute-Constraint-Union(Collected_Global_Constraints cgc):
E1. union ← ∅
E2. css_unique ← Remove duplicate entries from cgc
E3. seq_list ← Sort entries in css_unique by increasing (seq, gv)

F1. For each item in seq_list
F2.   if any Globally_Ordered_Update
F3.     G* ← Globally_Ordered_Update with Proposal from latest view (gv, lv)
F4.     union ← union ∪ G*
F5.   else
F6.     MAX_GV ← global view of entry with latest global view
F7.     if any Proposal from MAX_GV
F8.       P* ← Proposal from MAX_GV and latest local view
F9.       union ← union ∪ P*
F10. return union
```

Figure 3.22: 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.

properties in Section 3.6. We now describe the protocols in detail.

LOCAL-VIEW-CHANGE: When a server's local timer, Local_T, expires, it increments its local view to lv and suggests this view to the servers in its site by invoking THRESHOLD-SIGN on a New_Rep(lv) message. When $2f + 1$ stable servers move to local view lv, THRESHOLD-SIGN returns a signed New_Rep(lv) message to all stable servers in the site. Since a signed New_Rep(lv) message cannot be generated unless $2f+1$ servers suggest local view lv, such a message is proof that $f + 1$ correct servers within a site are in at least local view lv. We say a server has *preinstalled* local

```
RELIABLE-SEND-TO-ALL-SITES( message m ):
A1. Upon invoking:
A2.   rel_message ← 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 ≤ 2f+1:
D2.     my_server_id ← Server_id
D3.     for each site S:
D4.       SEND to server in site S with Server_id = my_server_id: m
```

Figure 3.23: 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.

view lv if it has a signed New_Rep(lv) message. Servers send their latest signed New_Rep message to all other servers in the site, and, therefore, all stable servers immediately move to the highest preinstalled view. Each server uses the following function to determine the id of its representative: Local_view mod $3f + 1$. A server starts its Local_T timer *only* when its preinstalled view equals its local view (i.e., it has a New_Rep(lv) message where its Local_view $= lv$). Since at least $f + 1$ correct servers must timeout (i.e., Local_T must expire) before a New_Rep message can be created for the next local view, the servers in the site increment their views consecutively and remain in each local view for at least a local timeout period. Moreover, if global progress does not occur, then stable servers will remain in a local view for one local timeout period.

GLOBAL-LEADER-ELECTION: When a server's global timer, Global_T, expires, it increments its global view to gv and suggests this global view to other servers in its site, S, by invoking THRESHOLD-SIGN on a Global_VC(S,gv) message. A theshold signed Global_VC(S,gv) message proves that at least $f + 1$ servers in site S must be in global view gv or above. Site S attempts to preinstall global view gv by sending this message to all other sites. A set of a majority of of Global_VC(gv) messages (i.e., *global preinstall proof*) proves that at least $f + 1$ correct servers in a majority of sites have moved to at least global view gv. If a server collects a global preinstall proof for gv, we say it has preinstalled global view gv. When a server preinstalls a new global view, it sends the corresponding global preinstall proof to all connected servers using RELIABLE-SEND-TO-

ALL-SITES (Figure 3.23). Therefore, as soon as any stable server preinstalls a new global view, all stable servers will preinstall this view. Each server uses the following function to determine the id of the leader site: Global_view mod N, where N is the number of sites in the system. As in the local representative election protocol, a server starts its Global_T timer *only* when its preinstalled view equals its global view (i.e., it has a set of Global_VC(gv) messages from a majority of sites where its Global_view $= gv$). Since at least $f + 1$ correct servers must timeout in a site (i.e., Global_T must expire) before the site can construct a Global_VC message for the next global view, stable servers increment their global views consecutively and remain in each global view for at least one global timeout period.

Construct-Local-Constraint: The CONSTRUCT-LOCAL-CONSTRAINT protocol (Figure 3.18) is invoked by a newly elected leader site representative (Figure 3.14, line D6). The protocol guarantees sufficient intra-site reconciliation to safely make progress after changing the site representative. As a result of the protocol, servers become *locally constrained*, meaning their Local_History data structures 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.

A site representative invokes the protocol by sending a sequence number, seq, to all servers within the site. A server invokes the Construct_Local_Server_State procedure (Figure 3.21, block A) and responds with a message containing all Prepare_Certificates and Proposals with a higher sequence number than seq. The representative computes the union of $2f + 1$ responses, eliminating duplicates and using the entry from the latest view if multiple updates have the same sequence number (Figure 3.22, block A); it then broadcasts the union within the site in the form of a Local_Collected_Servers_State message. When a server receives this message, it applies it to its Local_History, adopting the bindings contained within the union.

Construct-ARU: The CONSTRUCT-ARU protocol (Figure 3.19) is used by the leader site during a global view change. It is similar to CONSTRUCT-LOCAL-CONSTRAINT in that it provides intra-site reconciliation, but it functions in the global context. The protocol generates an Aru_Message reflecting the sequence number up to which at least $f + 1$ correct servers in the leader site have

globally ordered all previous updates.

Construct-Global-Constraint: The CONSTRUCT-GLOBAL-CONSTRAINT protocol (Figure 3.20) is used by the non-leader sites during a global view change. It generates a message reflecting the state of the site's knowledge above the sequence number contained in the result of CONSTRUCT-ARU. The leader site collects these Global_Constraint messages from a majority of sites.

Global View Change: The GLOBAL-VIEW-CHANGE protocol (Figure 3.17) is triggered after a leader site election. The representative of the new leader site invokes CONSTRUCT-ARU with its Global_aru (i.e., the sequence number up to which it has globally ordered all updates). The resulting threshold-signed Aru_Message contains the sequence number up to which at least $f + 1$ correct servers within the leader site have globally ordered all 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 and sends the resultant Global_Constraint message to the representative of the new leader site. 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.

3.4.4 Timeouts

Steward uses timeouts to detect failures. 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. In this section, we describe the timeouts that we use and how their relative values ensure liveness. The timeouts in the servers have been carefully engineered to allow a correct representative of the leader site to eventually order an update.

Steward uses timeout-triggered protocols to elect new coordinators. 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. In order to provide liveness, Steward changes coordinators using three timeout values. These values cause the coordinators of

the global and local protocols to be elected at different rates, guaranteeing 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 the server is in the leader site. At least two correct representatives in the leader site must serve complete terms during each global view. 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, then the first correct server to serve a full reign as representative at the leader site will complete GLOBAL-VIEW-CHANGE. The second correct server will be able to globally order and execute a new update, thereby making global progress.

Our protocols do not assume synchronized clocks; however, we do assume that the drift of the clocks at different servers is bounded. This assumption is valid considering today's technology. In order to tolerate different clock rates at different correct servers, each of the relationships given

above can be multiplied by the ratio of the fastest clock to the slowest. Such a modification ensures that each server in the leader site will remain in power for long enough to contact at least one correct server in each site, even if the clocks in the servers in the leader site run at faster rates than the clocks in the servers in the other sites. We assume that the adversary is unable to control the clock drift. Our liveness proof (see Section 3.6) does not consider clock drift. However, the proof can be modified to include a factor that accounts for clock drift among the correct servers.

Timeout management: We compute our timeout values based on the global view as shown in Figure 3.16. If the system is stable, all stable servers will move to the same Global view (Figure 3.15). Timeouts T1, T2, and T3 are deterministic functions of the global view guaranteeing that the timeout relationships described above are met at *every* stable server. Timeouts double every N global views, where N 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 so that global progress can 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.

Like many other protocols that rely on relatively weak synchrony assumptions for liveness, Steward does not provide a means to reduce timeout values. Timeout values can grow to arbitrarily large values during periods of network instability. As a consequence, at the time when the system becomes stable, timeouts may be much longer than necessary, and, thus, faults may take longer to identify than necessary. In practice, timeouts could be capped at some maximum expected value, but a system modified in this way requires stronger synchrony assumptions to provide liveness.

Adaptively selecting timeouts in a Byzantine environment is an open research problem. Our recent protocol, Prime [64], includes a SuspectLeader protocol that uses round trip times between all pairs of servers to adaptively decide on a level of performance that a correct leader should meet. However, Prime requires stronger synchrony assumptions for liveness than Steward.

```
LOCAL-RECONCILIATION:
A1. Upon expiration of LOCAL_RECON_TIMER:
A2.   local_session_seq++
A3.   requested_aru ← Global_aru
A4.   Local_Recon_Request ← 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 ≤ 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] ← local_session_seq
B9.   last_local_request_time[server_id] ← current_time
B10.  last_local_requested_aru[server_id] ← requested_aru
B11.  if Global_aru > requested_aru
B12.    THROTTLE-SEND(requested_aru, Global_aru, LOCAL_RATE, W) to server_id
```

Figure 3.24: LOCAL-RECONCILIATION Protocol. Recovers missing Globally_Ordered_Updates within a site. Servers limit the rate at which they respond to requests and the rate at which they send requested messages.

3.4.5 Reconciliation

Steward uses two reconciliation protocols to recover missing globally ordered updates. These protocols are used to overcome message loss, which can result from network failures and/or a faulty server's refusal to send or receive a message. Since wide-area bandwidth is limited, Steward attempts to reconcile missing updates locally (i.e., within a site) when possible to avoid triggering global reconciliation. We highlight the important details of these protocols below.

Local-Reconcilation The LOCAL-RECONCILIATION protocol (Figure 3.24) uses a request/response mechanism to provide intra-site reconciliation. Each server periodically broadcasts a Local_Recon_Request message, containing that server's Global_aru value. Upon receiving a reconciliation request, a server responds to the request if it has a Global_aru higher than the requested aru, and (1) the request is fresh, (2), the request arrived sufficiently long after the last request was received, and (3) the request contains a sequence number at least as high as the previous request. When correct servers within a site are sufficiently connected, they will eventually reconcile all Globally_Ordered_Updates through the maximum Global_aru of any correct server.

Global-Reconciliation The GLOBAL-RECONCILIATION protocol (Figure 3.25) consists of two stages. In the first stage, each server periodically attempts to construct a threshold-signed global reconciliation request. The requesting server broadcasts a Global_Recon_Request message to the

```
GLOBAL-RECONCILIATION:
A1. Upon expiration of GLOBAL_RECON_TIMER:
A2.   global_session_seq++
A3.   requested_aru ← Global_aru
A4.   g ← Global_History[requested_aru].Globally_Ordered_Update
A5.   Global_Recon_Request ← 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 ≤ 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] ← global_session_seq
B11.  last_global_request_time[server_id] ← current_time
B12.  last_global_requested_aru[server_id] ← requested_aru
B13.  if Global_aru ≥ requested_aru
B14.    sig_share ← GENERATE-SIGNATURE-SHARE()
B15.    SEND to server_id: sig_share
B16.  if Global_aru < requested_aru
B17.    when Global_aru ≥ requested_aru:
B18.      sig_share ← 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 ← 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] ≤ requested_aru
D3.     max_global_requested_aru[site_id] ← requested_aru
D4.   else
D5.     ignore GLOBAL_RECON
D6.   if (site_id == Site_id) or (server_id ≠ Server_id)
D7.     ignore GLOBAL_RECON
D8.   if global_session_seq ≤ 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] ← global_session_seq
D14.  last_global_request_time[site_id] ← current_time
D15.  if Global_aru > requested_aru
D16.    THROTTLE-SEND(requested_aru, Global_aru, GLOBAL_RATE, W) to server_id
```

Figure 3.25: 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.

servers in its local site and obtains partial signatures from those servers with a Global_aru at least as high as the sequence number contained in the request. The requesting server then combines the partial signatures to form a GLOBAL-RECON message. In the second stage, the requesting server sends the GLOBAL-RECON message to its peer servers in the other sites (i.e., the servers with the same Server_id), which then respond to the request by sending the missing Globally_Ordered_Updates to the requesting server.

The protocol uses two techniques to prevent a faulty server from consuming the resources of the

correct servers (both processing time and wide-area bandwidth). First, the protocol uses a throttling mechanism to limit the number of partial signatures a correct server in a requester's local site will generate; a correct server only responds to fresh reconciliation request messages at a particular rate. Second, upon receiving a GLOBAL-RECON message from another site, a peer server broadcasts the request to the other servers in its site. For each site, the servers in the peer site store the highest sequence number contained in a request received from that site. Since GLOBAL-RECON requests are threshold-signed, at least $f + 1$ correct servers in the requester's local site have a Global_aru at least as high as the sequence number in the request. Thus, if a subsequent GLOBAL-RECON message arrives containing a sequence number less than or equal to the highest requested sequence number, it can be ignored, since the requester can recover the message locally from a correct server via LOCAL-RECONCILATION.

3.5 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.

Test Bed 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 test bed 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 test bed 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 [65] 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 BFT [53] on the same network setup with five sites, 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 the implementations performed almost identically, 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 milliseconds latency links (emulating crossing the continental US).

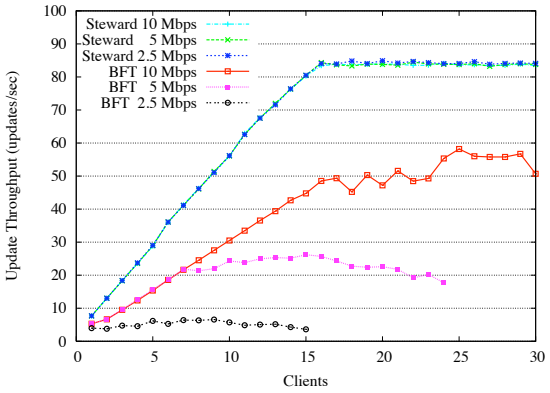

Figure 3.26: Write Update Throughput

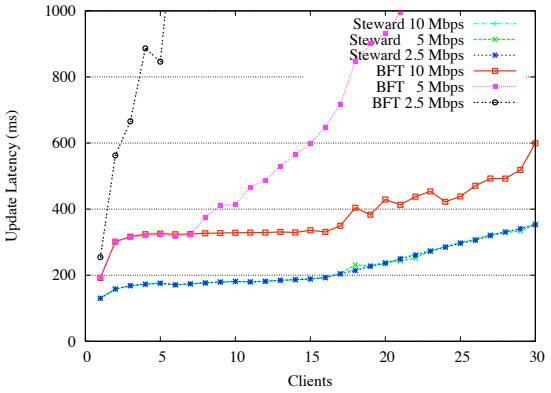

Figure 3.27: Write Update Latency

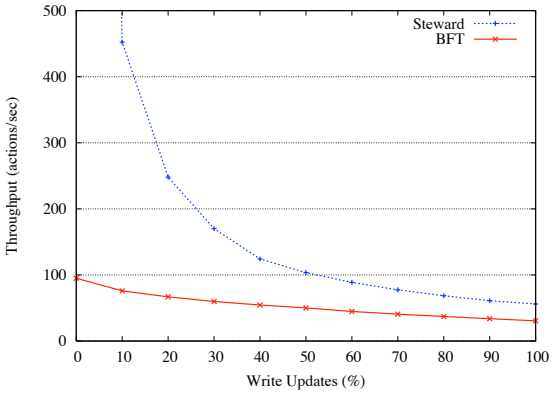

Figure 3.28: Update Mix Throughput - 10 Clients

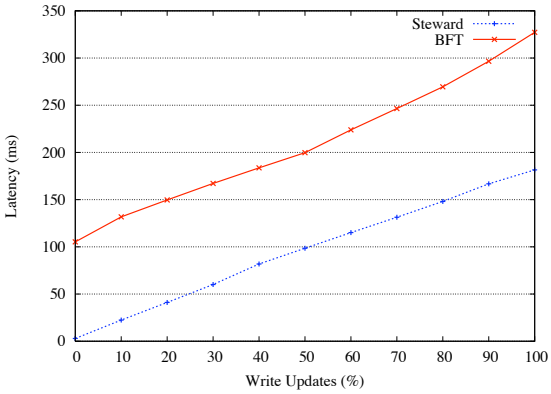

Figure 3.29: Update Mix Latency - 10 Clients

In the first experiment, clients inject write updates. Figure 3.26 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.

Figure 3.27 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 our more recent work [19].

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. Figures 3.28 and 3.29 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

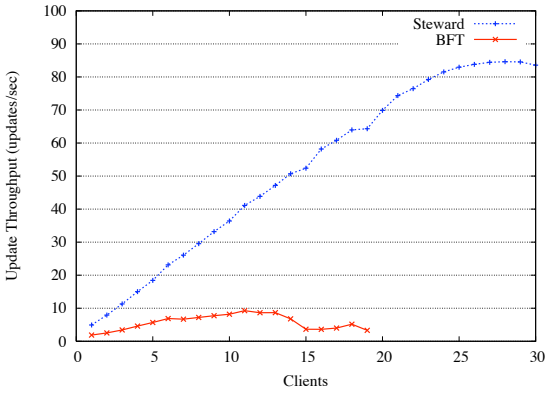

Figure 3.30: WAN Emulation - Write Update Throughput

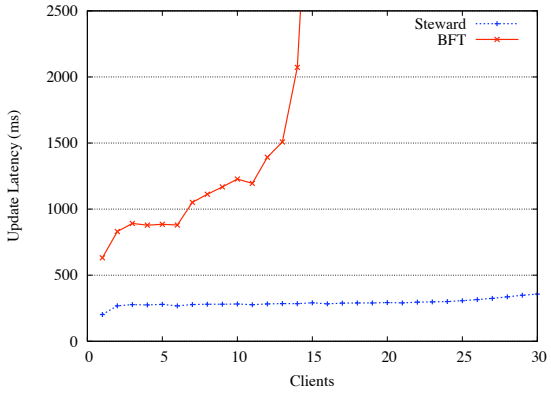

Figure 3.31: WAN Emulation - Write Update Latency

performance.

Wide Area Scalability: To demonstrate Steward's scalability on real networks, we conducted an experiment that emulated a wide-area network spanning several continents. We selected five sites on the Planetlab network [66], 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 are 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).

Figure 3.30 shows the average write update throughput as we increased the number of clients in the system, while Figure 3.31 shows the average update latency. Steward is able to achieve its maximum throughput of 84 updates/sec with 27 clients. The latency increases from about 200 ms for 1 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 increases to several seconds with more than 6 clients.

Comparison with Non-Byzantine Protocols: Since Steward deploys a lightweight fault-tolerant protocol between the wide-area sites, we expect it to achieve performance comparable to existing benign fault-tolerant replication protocols. We compare the performance of our hierarchical

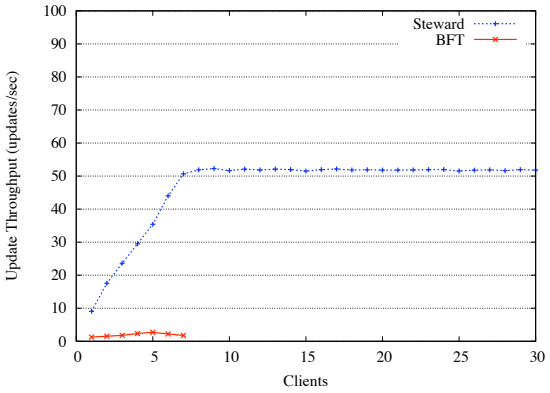

Figure 3.32: CAIRN Emulation - Write Update Throughput

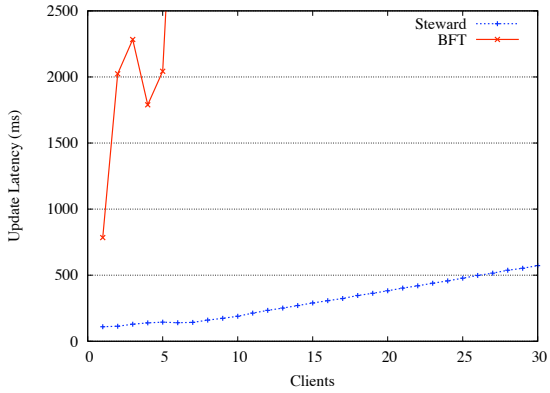

Figure 3.33: CAIRN Emulation - Write Update Latency

Byzantine architecture with that of two-phase commit protocols. In [67] we evaluated the performance of two-phase commit protocols [27] using a wide area network setup across the US, called CAIRN [68]. We emulated the topology of the CAIRN network using the Spines messaging system, and we ran Steward and BFT on top of it. The main characteristic of the CAIRN topology is that East and West Coast sites were connected through a single link of 38 ms and 1.86 Mbps.

Figures 3.32 and 3.33 show the average throughput and latency of write updates, respectively, of Steward and BFT on the CAIRN network topology. Steward achieved 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 [67] 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. 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.

Red-Team Results: In December 2005, DARPA conducted a red-team experiment on our Steward implementation to determine its practical survivability in the face of white-box attacks. We provided the red team with system design documents and gave them access to our source code; we also worked closely with them to explain some of the delicate issues in our protocol concerning safety and liveness. Per the rules of engagement, the red team had complete control over f replicas in each site and could declare success if it (1) stopped progress or (2) caused consistency errors

among the replicas. The red team used both benign attacks, such as packet reordering, packet duplication, and packet delay, and Byzantine attacks, in which the red team ran its own malicious server code. While progress was slowed down in several of the tests, such as when all messages sent by the representative of the leader site were delayed, the red team was unable to block progress indefinitely and never caused inconsistency. Thus, according to the rules of engagement, none of the attacks succeeded. We plan to investigate ways to ensure high performance under attack (which is stronger than the eventual progress afforded by system liveness) in future work.

3.6 Proof of Correctness

In this section we show that Steward provides the service properties specified in Section 3.3. We begin with a proof of safety and then consider liveness.

3.6.1 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 3.6.1 *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 3.6.1 *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 3.6.1 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 3.6.2 *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 3.11 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 3.6, line A8). Thus, this server did not contribute to PC2, a contradiction. □

Lemma 3.6.3 *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 3.6.2, 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 3.11, 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 3.6.2. □

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 3.6.1 *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 3.6.1 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 3.6.4 *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 3.6.1 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 3.11, line C7). These servers do so after collecting a Prepare Certificate(gv, lv, seq, u) binding seq to u (Figure 3.11, line C3). By Lemmas 3.6.2 and 3.6.3, 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 3.11, line D3). By Lemma 3.6.3, 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 3.6.1 holds with respect to P for the remainder of (gv, lv). □

We now proceed to show that Invariant 3.6.1 holds across local view changes. Before proceeding, we introduce the following terminology:

DEFINITION 3.6.1 *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 3.6.5 *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 3.6.1 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 3.18 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 3.6.1 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 3.7, 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 3.6.1 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 3.21, Block A), after bringing its Pending_Proposal_Aru up to the invocation sequence number (if necessary). Invariant 3.6.1 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 3.22 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 3.18, 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 3.18, 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 3.6.1. □

Our next goal is to show that if Invariant 3.6.1 holds at the beginning of a view after the view in which a Proposal has been constructed, then it holds throughout the view.

Lemma 3.6.6 *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 3.6.1 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 3.6.1 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 3.6, A8) or in its Global_History[seq] data structure (Figure 3.6, 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 3.6.1 holds throughout (gv, lv'). □

We can now prove Lemma 3.6.1:

Proof: By Lemma 3.6.4, Invariant 3.6.1 holds with respect to P throughout (gv, lv). By Lemma 3.6.5, the invariant holds with respect to P during and after CONSTRUCT-LOCAL-CONSTRAINT. By Lemma 3.6.6, the invariant holds at the beginning and end of view (gv, $lv + 1$). Repeated applications of Lemma 3.6.5 and Lemma 3.6.6 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 3.11, 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. □

Finally, we can prove Claim 3.6.1:

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 3.6.1, 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 3.6.2 *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 3.6.2 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 3.6.7 *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 3.6.7 using a series of lemmas. We use a strategy similar to the one used in proving Lemma 3.6.1 above, and we maintain the following invariant:

INVARIANT 3.6.2 *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 3.6.2 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 aGlobally_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 3.6.2 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 3.6.8 *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 3.6.2 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 3.8, 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 3.6.1, 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 3.6.1 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 3.8, 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 3.6.2 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 3.6.9 *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 3.6.2 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 3.6.2 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 3.6.2 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 3.6.2 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 3.19, 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 3.19, line G2). Since Invariant 3.6.2 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 3.6.10 *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 3.6.2 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 3.20, 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 3.6.2 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 3.6.11 *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 3.6.2 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 3.17, line A2). By Lemma 3.6.9, Invariant 3.6.2 is not violated during this protocol. The second case occurs at the non-leader sites, during a run of CONSTRUCT-GLOBAL-CONSTRAINT (Figure 3.17, line C4). By Lemma 3.6.10, Invariant 3.6.2 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 3.17, lines E5 and F2). We must now show that Invariant 3.6.2 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 3.6.2, 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 3.6.2 reports a Proposal message binding seq to u in its Global_Server_State message (Figure 3.21, 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 3.6.2 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 3.22, Blocks C and D). The resultant Global_Constraint message therefore binds seq to u. Invariant 3.6.2 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 3.22, Blocks E and F) By the rules of applying the Collected_Global_Constraints message (Figure 3.8, 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 3.6.2 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 3.19, 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 3.19, 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 3.6.2 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 3.19, lines B5 and B7). Invariant 3.6.2 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 3.22, 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 3.6.2 is not violated when the correct servers in the leader site become globally constrained by applying the mesasge (Figure 3.8, 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 3.19, 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 3.19, 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 3.19, 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 3.6.2. □

Corollary 3.6.12 *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 3.6.2 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 3.6.2.*

Proof: We consider each of the four sub-cases described in Lemma 3.6.11. 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 3.6.2. □

Our next goal is to show that if Invariant 3.6.2 holds at the beginning of a global view after which an update has been globally ordered, then it holds throughout the view.

Lemma 3.6.13 *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 3.6.2 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 3.6.12, since at least $f + 1$ correct servers became globally constrained in gv', the leader site meets Condition 1 of Invariant 3.6.2, 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 3.13, line A3 or A5). Similarly, a correct server would see a conflict (Figure 3.6, 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 3.6.2 holds throughout global view gv'. □

We can now prove Lemma 3.6.7:

Proof: By Lemma 3.6.8, Invariant 3.6.2 holds with respect to P1 throughout global view gv. By Lemma 3.6.11, the invariant holds with respect to P1 during and after the GLOBAL-VIEW-CHANGE protocol. By Lemma 3.6.13, the invariant holds at the beginning and end of global view $gv + 1$. Repeated application of Lemma 3.6.11 and Lemma 3.6.13 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 3.11, 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 3.6.2:

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 3.6.7, 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 3.6.1 and 3.6.2, 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 3.4, 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. □

3.6.2 Proof of Liveness

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 3.6.2 *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 3.6.3 *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 3.6.3, we first prove two lemmas relating to LOCAL-RECONCILIATION and GLOBAL-RECONCILIATION.

Lemma 3.6.14 *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 3.24, 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 3.6.15 *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 3.6.14, 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 3.25, 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 3.25, 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 3.6.3:

Proof: Assume, without loss of generality, that stable site S has a stable server with a Global_aru of max_stable_seq. By Lemma 3.6.14, 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 3.6.15, 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 3.6.14 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 3.6.4 *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 3.6.5 *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 3.6.5, we use the following lemma:

Lemma 3.6.16 *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 3.10, 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 3.6.5.

Proof: A correct server combines $2f+1$ correct partial signatures to generate a threshold signature on m. From Lemma 3.6.16, 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 3.10, 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 3.4, 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 3.6.3 *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 3.6.4 *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 3.6.6 *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, 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 3.6.6, we maintain the following invariant and show that it always holds:

INVARIANT 3.6.3 *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 3.6.17 *Invariant 3.6.3 always holds.*

Proof: We show that Invariant 3.6.3 holds using an argument based on a state machine, SM. SM has the two states listed in Invariant 3.6.3.

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 3.6.3 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 3.6.3 holds before a state transition, it will hold after a state transition.

We first consider transition (1). We assume that Invariant 3.6.3 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 3.6.3 holds. This transition corresponds to Figure 3.15, 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 3.6.3 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 3.15, block E). Line E6 of Figure 3.15 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 3.8, 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 3.6.3 holds after transition (2).

We have shown that Invariant 3.6.3 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 3.6.6:

Claim 3.6.7 *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 3.23, 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 3.6.18 *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 3.15, lines E4 and E5) By Claim 3.6.7, 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 3.15, 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 3.6.5, 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 3.6.7, 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 3.6.19 *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 3.6.3, 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 3.6.7, 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 3.6.3, 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 3.6.17, 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 3.15. 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 3.8, 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 3.6.18, 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 3.6.6:

Lemma 3.6.20 *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 3.6.18, 3.6.19, and 3.6.20 to prove Claim 3.6.6:

Proof: By Lemma 3.6.19, 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 3.6.18, all stable servers will preinstall $gv+1$. By Lemma 3.6.18, 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 3.15, 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 3.6.8 *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 3.6.8, we use a state machine based argument to show that the following invariant holds:

INVARIANT 3.6.4 *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 3.6.21 *Invariant 3.6.4 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 3.6.4 holds before a state transition, it will hold after a state transition.

We first consider transition (1). We assume that Invariant 3.6.4 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 3.6.4 holds. This transition corresponds to lines A1 and A2 in Figure 3.14. 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 3.6.4 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 3.14, 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 3.7, 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 3.6.4 holds after transition (2).

We have shown that Invariant 3.6.4 holds when a server starts and that it holds after each state

transition, completing the proof. □

We can now prove Claim 3.6.8.

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 3.14, line D3). Therefore, all stable servers in site S will receive the New_Rep message and preinstall lp_{max}.

From Invariant 3.6.4, $lp_{max} = lv_{max} - 1$ 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 3.14, line D3). Therefore, all stable servers will increment their local views to lv_{max}, and invoke THRESHOLD-SIGN on New_Rep(lv_{max}) (Figure 3.14, line A5). By Claim 3.6.5, 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 3.6.9 *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 3.16. 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 3.6.9. 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 3.6.10 *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 3.6.8, 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 3.6.9, 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 $\bmod(3f+1)$, 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 3.6.9, 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 $\mathrm{mod} N$, 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 3.6.11 *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 3.6.8, 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 3.6.6, 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 3.6.9, $global\_to = (f + 3)local\_to\_ls$. 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 $\mathrm{mod}(3f + 1)$, 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 3.6.4, 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 3.6.22 *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 3.6.6, 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 3.6.11, 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 3.17, line A2), which occurs either after preinstalling the global view (Figure 3.15, line E8) or after completing a local view change when not globally constrained (Figure 3.14, 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 3.6.10, 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 3.6.22 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 3.6.22, 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 3.6.3 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 3.6.23 *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 3.12, line A3), and a correct server only sends a Prepare message if its Global_aru is at least $(seq - W)$ (Figure 3.6, 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 3.6.24 *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 3.6.23, 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 3.21, 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 3.6.25 *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 3.6.3, 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 3.17, 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 3.19, lines B5-B7), because it has a Global_aru of max_stable_seq. By Lemma 3.6.24, any valid Global_Server_State message can contain entries for at most $2 * W$ sequence numbers. We show below in Claim 3.6.13 that all correct servers have contiguous entries above the invocation sequence number in their Global_History data structures. From Figure 3.21 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 3.19, 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 3.19, lines D2 and D3. They do not see a conflict at Figure 3.5, 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 3.6.5, all stable servers convert the Prepare Certificates into Proposals and invoke THRESHOLD-SIGN on the union (line F2). By Lemma 3.6.5, all stable servers generate the Global_Constraint message (line G1) and invoke THRESHOLD-SIGN on the extracted union_aru (line G4). By Lemma 3.6.5, 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 3.6.26 *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 3.6.3, 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 3.6.24, the Global_Server_State messages contain entries for at most $2 * W$ sequence numbers. We show below in Claim 3.6.13 that all correct servers have contiguous entries above the invocation sequence number in their Global_History data structures. From Figure 3.21 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 3.20, 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 3.5, 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 3.6.5, all stable servers convert the Prepare Certificates into Proposals and invoke THRESHOLD-SIGN on the union (line E2). By Lemma 3.6.5, 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 3.6.4:

Proof: By Lemma 3.6.22, 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 3.6.25, 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 3.6.26, 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 3.6.27 *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 3.6.4, 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 3.14, 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 3.6.28 *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 3.6.27, 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 3.6.10, 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 3.6.29 *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 3.6.29, we use the following two lemmas to bound the size of the messages sent in CONSTRUCT-LOCAL-CONSTRAINT:

Lemma 3.6.30 *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 3.12, 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 3.6, 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 3.6.31 *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 3.17, 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 3.6, line D6). By Lemma 3.6.30, 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 3.6.29:

Proof: By Claim 3.6.3, 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 3.17, 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 3.8, 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 3.6.30, 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 3.18, 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 3.6.31, any valid Local_Server_State message contains at most W entries. We show below in Claim 3.6.13 that all correct servers have contiguous entries above the invocation sequence number in their Local_History data structures. From Figure 3.21 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 3.18, lines D2 and D3, at any stable server that sent a Local_Server_State message. They do not see a conflict at Figure 3.6, line E4, be-

cause 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 3.6.32 *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 3.6.27, 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 3.14, 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 3.6.29 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 3.6.5 *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 3.6.33 *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 3.6.3, 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 3.6.10, 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 3.6.3 has already completed (i.e., all stable servers have a Global_aru equal to max_stable_seq). From the pseudocode (Figure 3.12, 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 3.6.12 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 3.6.12.

Claim 3.6.12 *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 3.6.34 *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 3.6.34 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 3.6, 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 3.6.34), 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 3.6.1 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 3.6.1 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 3.6.1 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 3.6.35 *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 3.6.1 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 3.6.34 , if Proposal(gv, lv, seq, u), P, is constructed, then no constraining entries binding u to seq' exist in (gv, lv). Therefore, from Invariant 3.6.1, no Proposal(gv, lv'', seq', u), P' could have been constructed, where $lv'' \leq lv$. This follows, because, if P' was constructed, then Invariant 3.6.1 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 3.6.1, 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 3.6.34 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 3.6.2 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 3.6.2 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 3.6.2 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 3.6.36 *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 3.6.2 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 3.6.34, if P was constructed, then no constraining entries binding u to seq' exist in (gv, lv). Therefore, from Invariant 3.6.2, 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 3.6.1 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 3.6.2, 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 3.6.34 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 3.6.37 *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 3.13 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 3.6.36, 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 3.13, lines A6 - A7). In this part of the proof, we divide the Update_Pool data structure into two logical parts:

DEFINITION 3.6.6 *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 3.6.38 *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 3.6.3, 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 3.7, line F2). Since R is stable, all stable servers receive u within one local message delay. From Figure 3.7, 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 3.6.38 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 3.6.39 *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 3.6.38, 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 3.13, line A7). The returned update cannot have been executed by any stable server, since by Claim 3.6.3, 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 3.6.7 *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 3.6.8 *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 Lo-

cal_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 3.6.40 *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 3.7, lines B2 - B33, a correct server ignores a Prepare message unless it has a Pre-Prepare for the same sequence number. From Figure 3.6, 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 3.5, line A9). For those servers in the leader site, a Proposal is generated when the THRESHOLD-SIGN protocol completes (Figure 3.11, 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 3.8, 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 3.6.41 *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 3.18, line B7). The server includes all constraining entries directly above seq (Figure 3.21, 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 3.6 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 3.18, 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 3.7, 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 3.6.42 *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 3.6.41 (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 3.8, 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 3.5, lines H5 - H6), and thus it is contiguous up to that sequence number. When Compute_Constraint_Union is applied (Figure 3.22, 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 3.6.13 *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 constrain-

ing 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 3.6.40, 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 3.6.41, 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 3.6.42, 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 3.6.43 *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 3.6.13, 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 3.6.38, 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 3.6.44 *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 3.6.39 and 3.6.43. □

Corollary 3.6.44 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 3.6.45 *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 3.6.27, 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 3.6.29, 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 3.11, 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 3.6.5, 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 3.6. 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 3.7 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 3.8 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 3.6.13, 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 3.6.34, 3.6.35, and 3.6.36 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 3.6.33. □

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 3.6.5 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 3.6.46 *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 3.6.44, the Progress_Rep will invoke ASSIGN-SEQUENCE when Global_seq is equal to $max\_stable\_seq + 1$. By Lemma 3.6.45, there exists an infinite set of views in which ASSIGN-SEQUENCE will return a Proposal in bounded finite time. By Claim 3.6.10, 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 3.5, 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 3.8, 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 3.6.3, 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 3.6.32, 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 3.12, line A4). By Corollary 3.6.44, 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 3.12, line A6).

By Lemma 3.6.33, 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 3.6.45, ASSIGN-SEQUENCE will complete in bounded finite time, and by Lemma 3.6.46,

THRESHOLD-SIGN will be invoked by all stable servers at the non-leader sites. By Claim 3.6.5, 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. □

3.7 Steward Summary

This chapter 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.

Chapter 4

Customizable Fault Tolerance for Wide-Area Replication

This chapter presents the composable architecture, a customizable Byzantine fault-tolerant state machine replication architecture for wide-area networks. The composable architecture is designed for the same wide-area environments as Steward, and provides similar scalability to Steward. It improves upon Steward by offering superior customizability and simplicity. The work presented in this chapter was done in collaboration with Yair Amir, Brian Coan, and Jonathan Kirsch.

4.1 Composable Architecture Overview

This chapter presents the first scalable wide-area replication system that (1) achieves high performance through the efficient use of wide-area bandwidth and (2) allows customization of the fault tolerance approach used within and among the local-area sites. The composable architecture uses the state machine (SM) approach [13, 43] to transform the physical machines in each site into a *logical machine* (LM), and the logical machines run a wide-area protocol.

Using the state machine approach to build logical machines is a well-known technique for cleanly separating the protocol used to implement the logical machine from the protocol running on top of it. Representative systems include Voltan [46], Immune [52], BASE [69], Starfish [47], and Thema [48], which are described in more detail in Chapter 2. The state machine approach affords free substitution of the fault tolerance method used in each site and in the wide-area replication protocol, allowing a Byzantine or benign fault-tolerant protocol to be selected depending on system requirements and perceived risks. Thus, the system is composable with respect to the protocols run

within and among the sites.

All previous Byzantine fault-tolerant SM-based logical machine abstractions send messages redundantly in order to guarantee reliable communication in the presence of malicious protocol participants. Typically, to prevent malicious servers from blocking the message transmission, at least $f + 1$ servers in the sending LM will each send to $f + 1$ servers in the receiving LM, where f is the number of potential faults in each LM.[1] While this strategy works well on local-area networks, where bandwidth is plentiful, it is impractical for replication systems that must send many messages over wide-area links. In our experience, it is wide-area bandwidth and not computational constraints that limits the performance of well-engineered wide-area replication systems. To address this weakness, we present BLink, the first Byzantine fault-tolerant communication protocol that guarantees efficient wide-area communication between logical machines. BLink is specifically designed for use in systems where (1) the physical machines comprising an LM are located in a LAN that provides low-latency, high-bandwidth communication, and (2) the LMs are located in different LANs, and are connected by high-latency, low-bandwidth links. BLink usually requires only one physical message to be sent over the wide-area network for each message sent by the logical machine.

As explained in Chapter 1, Steward [16], shares some similarities with the composable architecture presented in this chapter. Specifically, both systems use a hierarchical logical machine architecture and provide high performance by efficiently utilizing wide-area bandwidth. However, they use fundamentally different techniques to construct their logical machines. The servers comprising each LM in the composable architecture totally order *all* events that cause a state transition in the protocol running on top of them (i.e., updates, acknowledgements, and wide-area timeouts), and execute these events in the same order.[2] This contrasts with the approach taken in Steward, where the wide-area protocol makes state transitions based on unordered events. As a result, in Steward, the protocols running within the sites and those running among the sites are interdependent and

[1] It may be possible to use a peer-based protocol in which each of $2f + 1$ servers sends to a unique peer. To the best of our knowledge, no existing system uses this method, except for Steward [16] (see Chapter 3), which uses it sparingly to send global view change messages.

[2] An optimization in the protocol used to forward updates to the leader LM allows the updates to pass through a non-leader LM unordered. These updates are ordered by the leader LM, as described in Section 4.5.

cannot be separated. Consequently, the fault tolerance approach within and among the sites cannot be customized. Since Steward runs a benign fault-tolerant wide-area protocol, it cannot survive a site compromise. It was this deficit in particular, coupled with Steward's lack of customizability, that led us to develop the composable architecture.

To mitigate the high cost of the additional ordering required by the state machine approach, the composable architecture uses two optimizations. First, we amortize the computational costs associated with digital signatures within the LM ordering protocol using known aggregation techniques. Second, we use a Merkle tree [70] mechanism to amortize the cost of threshold signatures while producing a self-contained, threshold-signed wide-area message. Amortizing optimizations enable an LM to process and send on the order of a thousand wide-area messages per second, preventing LM throughput from limiting overall performance. State machine based LMs augmented with BLink and the Merkle tree optimization have precisely the necessary properties to build a customizable fault-tolerant replication system without sacrificing performance.

The contributions made by the our work on the composable architecture are:

1. It presents a new hierarchical replication architecture for wide-area networks that combines high performance and customizability of the fault tolerance approach used within each site and among the sites. Using a Byzantine fault-tolerant protocol on the wide area protects against site compromises and offers fundamentally stronger security guarantees than our previous system.

2. It presents a new Byzantine fault-tolerant protocol, BLink, that guarantees efficient wide-area communication between logical machines, each of which is constructed from several non-trusted entities, such that messages usually require one send over the wide-area network. The use of BLink increases performance by over an order of magnitude in comparison to an SM-based logical machine approach that uses previous communication protocols, which require at least $2f + 1$, and typically $(f + 1)^2$, redundant sends.

3. It shows that by using optimizations that amortize the computational cost of the logical machine ordering, the composable architecture achieves high performance, outperforming the

Steward system by a factor of 4 when running a composition with the same level of fault tolerance.

We compare four possible compositions of the architecture, plus the Steward architecture, over emulated wide-area networks. The experiments show that the composable architecture that runs a wide-area benign fault-tolerant protocol and Byzantine local-area protocols within each site has performance that is 4 fold better than the original Steward architecture, which was the previous state of the art. The composable architecture achieves 12 percent lower performance than a new version of Steward that we developed for comparison that uses similar amortizing optimizations. This performance difference is the cost of providing clean separation and customizability. We also benchmarked a Byzantine over Byzantine composition, which provides fundamentally stronger fault tolerance than Steward, since Steward cannot survive a site compromise. While the systems are not strictly comparable because they offer different guarantees, the Byzantine over Byzantine composition performs 3 times better than the original Steward and achieves 35 percent lower performance than the new version of Steward that uses amortizing optimizations.

4.2 System Model and Service Guarantees

Servers are organized into wide-area *sites*; each site has a unique identifier known to all servers. 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. We can use a state transfer mechanism (as in [53]) or an update reconciliation mechanism (as in [71]) to reconcile states after a merge.

The free substitution property afforded by using SM-based logical machines allows our architecture to support a rich configuration space. Each site can employ either a Byzantine or a benign fault-tolerant SM replication protocol to implement its LM, and the system can run either a benign fault-tolerant or a Byzantine fault-tolerant wide-area protocol. We classify both servers and sites as either correct or faulty (benign or Byzantine). A correct server adheres to its protocol specification. A benign faulty server can crash but otherwise adheres to the protocol. A Byzantine server can

deviate from its protocol specification in an arbitrary way.

In what follows, we assume that Paxos is used as our benign fault-tolerant protocol and BFT is used as our Byzantine fault-tolerant protocol. Different protocol choices may require different assumptions (e.g., some Byzantine fault-tolerant protocols require a smaller fraction of faulty servers). A site running Paxos locally is benign faulty if more than f servers in the site are benign faulty, where the site has $2f + 1$ servers. A site running Paxos locally is Byzantine faulty if at least one server is Byzantine. Otherwise, the site is correct. A site running BFT is Byzantine faulty if more than f servers in the site are Byzantine, where the site has $3f + 1$ servers; otherwise the site is correct. When run on the wide area, Paxos can tolerate F benign faulty sites, where there are $2F + 1$ sites, but cannot tolerate a single Byzantine site; BFT can tolerate F Byzantine sites, where there are $3F + 1$ sites. [3]

Clients introduce updates into the system by communicating with the servers in their local site. Each update is uniquely identified by a pair consisting of the identifier of the client that generated the update and a unique, monotonically increasing sequence number. We say that a client *proposes* an update when the client sends the update to a server in the local site. A client receives a reply to its update after the update has been globally ordered and executed. Clients propose updates sequentially: a client, c, may propose an update with sequence number $i_c + 1$ only after it receives a reply for an update with sequence number i_c. A client retransmits its last update if no reply is received within a timeout period. Clients may be faulty; updates from faulty clients will be replicated consistently. Access control techniques can be used to restrict the impact of faulty clients.

We employ digital signatures, and we make use of a cryptographic hash function to compute message digests. We assume that all adversaries are computationally bounded such that they cannot subvert these cryptographic mechanisms. When BFT is deployed within a site, the servers in that site use an ($f + 1$, $3f + 1$) threshold digital signature scheme [56]. 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

[3] We use capital F to denote the number of faulty sites that the wide-area protocol can tolerate and lowercase f to denote the number of faulty servers that the local-area protocol can tolerate.

without knowing $f + 1$ or more shares. We assume each server knows (1) the public keys of the other servers in its site, (2) the public keys for each of the other sites (used to verify threshold-signed messages), and (3) the public keys of all clients.

Our system achieves replication via the state machine approach, establishing a global, total order on client updates in the wide-area protocol. Each server executes an update with global 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.

Our replication system provides the following two safety conditions:

DEFINITION 4.2.1 S1 - SAFETY: *If two correct servers execute the i^{th} update, then these updates are identical.*

DEFINITION 4.2.2 S2 - VALIDITY: *Only an update that was proposed by a client may be executed.*

When running Paxos on the wide area, these safety conditions hold as long as no site is Byzantine. When running BFT on the wide area, the conditions hold as long as no more than F sites are Byzantine. We refer to these conditions as the *fault assumptions needed for safety*. Since no asynchronous, fault-tolerant replication protocol tolerating even one failure can always be both safe and live [63], we provide liveness under certain synchrony conditions. We first define the following terminology and then specify our liveness guarantee:

- *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.

- *A client is connected to a site* if it can communicate with all correct servers in that site.

- A *site is stable* with respect to time T if there exists a set, S, of c servers within the site (with $c = 2f + 1$ for sites tolerant to Byzantine failures and $c = f + 1$ for sites tolerant to benign failures), where, for all times $T' > T$, the members of S are (1) correct and (2) connected. We call the members of S *stable servers*.

- Let F be the maximum number of sites that may be faulty. The *system is stable* with respect to time T if there exists a set, W, of r wide-area sites (with $r = F + 1$ when sites may exhibit benign failures and $r = 2F + 1$ when sites may be Byzantine) where, for all times $T' > T$, the sites in W are (1) stable with respect to T and (2) connected. We call W the STABLE-CONNECTED-SITES.

DEFINITION 4.2.3 L1 - GLOBAL LIVENESS: *If the system is stable with respect to time T and the fault assumptions needed for safety are met, then if, after time T, a stable server in the* STABLE-CONNECTED-SITES *receives an update which it has not executed, then that update will eventually be executed by all stable servers in the* STABLE-CONNECTED-SITES.

4.3 Customizable Replication System Architecture

In our composable architecture, the physical machines in each site implement a *logical machine* by running a local state machine replication protocol [13,43]. We then run a state machine replication protocol on top of these logical machines, among the sites. Using SM-based logical machines is an established technique for cleanly separating the implementation of the LM from the protocol running on top of it. Our architecture leverages the flexibility afforded by this technique, allowing one to customize the protocol and type of fault tolerance desired, both within each LM and among the LMs. Further, we can use the known safety proof for the wide-area protocol (when run among single machines), together with one for the local SM replication protocol, to trivially prove safety for the composition. The liveness proof is more complicated, but much simpler than what is necessary when the wide-area and local-area protocols are interdependent. See Section 4.8 for a more formal discussion of the safety and liveness properties. In the remainder of this section, we first review how we use the SM approach to build our logical machines, and then present several compositions

of our architecture.

Implementing Logical Machines: The wide-area replication protocol running on top of our LMs runs just as it would if it were run among a group of single machines, each located in its own site. Each LM sends the same types of wide-area messages and makes the same state transitions as would a single machine running the wide-area replication protocol. To support this abstraction, the physical machines in each site use an agreement protocol to totally order all events (messages and timeouts) that cause state transitions in the wide-area protocol. The physical machines then execute the events in the agreed upon order. Thus, the LM conceptually executes a single stream of wide-area protocol events. The LMs communicate using BLink to avoid sending redundant wide-area messages.

The SM approach assumes that all events are deterministic. As a result, we must prevent the physical machines from diverging in response to non-deterministic events. For example, although the physical machines within a site may fire a local timeout asynchronously, they must not act on the timeout until its order is agreed upon. We use a technique similar to BASE [69] to handle non-deterministic events. To implement an LM timeout when a Byzantine fault-tolerant agreement protocol is used, each server in the site sets a local timer, and when this timer expires, it sends a signed message to the leader of the agreement protocol. The leader waits for $f + 1$ signed messages proving that the timer expired at at least one correct server and then orders a logical timeout message (containing this proof).

Outgoing wide-area messages carry an RSA signature [59]. When a logical machine is implemented with a benign fault-tolerant protocol, the message carries a standard RSA signature. When running a Byzantine fault-tolerant local protocol, the physical machines within the site generate an RSA threshold signature, attesting to the fact that $f + 1$ servers agreed on the message. This prevents malicious servers within a site from forging a message. Moreover, outgoing messages carry only a single RSA (threshold) signature, saving wide-area bandwidth. Our architecture amortizes the high cost of threshold cryptography over many outgoing messages. We use a technique similar to Steward to prevent malicious servers from disrupting the threshold signature protocol.

Protocol Compositions: The free substitution property of our architecture makes it extensible,

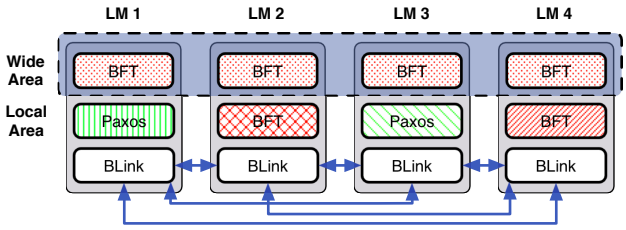

Figure 4.1: An example composition of four logical machines, each comprising several physical machines. The LMs receive wide-area protocol messages via BLink, which passes these messages to the local-area ordering protocol (an independent instance of either Paxos or BFT). The local-area protocol passes locally ordered messages up to the wide-area protocol (a single global instance of BFT), which executes them immediately. If a state transition causes the wide-area protocol to send a message, the LM generates a threshold signed message and passes it to Blink, which reliably transmits it to the destination logical machine.

allowing one to use any of several existing state of the art replication protocols, both within each site and on the wide area. In this paper, we focus on four compositions of our architecture, using two well known, flat replication protocols: Paxos [18,29] as our benign fault-tolerant protocol, and BFT [10] as our Byzantine fault-tolerant replication protocol. We refer to compositions as *wide-area protocol/local-area protocol*. For example, we refer to a composition which runs BFT on the wide area and Paxos on the local area as BFT/Paxos.

Figure 4.1 shows a representative system having four logical machines, each running an independent local ordering protocol. The logical machines run a single instance of BFT on the wide-area to globally order client updates. Each LM can be configured with any number of physical machines. Since the wide-area protocol is BFT, the system can withstand a site compromise, which occurs when more than f servers in the site are faulty.

We conclude by providing an example of Paxos/BFT that traces the flow of a client update through the system during normal-case operation. First, a client sends an update to a server in its own site, which forwards the update to the leader site (i.e., the site coordinating the Paxos wide-area protocol). Client updates are sent from a local server to the leader site using a separate protocol, which is described in Section 4.5. The leader site LM uses BFT (requiring three local communication rounds), to locally order the message event corresponding to the reception of the update by the LM. The LM generates a wide-area proposal message, binding a global sequence number to the update. The message is then threshold signed by the leader site LM via a one-round protocol. The

threshold-signed proposal is then sent (using BLink) to the other sites. Each non-leader LM orders the incoming proposal, generates an acknowledgement (accept) message for the proposal, and then sends the acknowledgement (using BLink) to the other LMs. Each LM then orders the reception of the accept message. When the proposal and a majority of accepts are collected, the LM globally orders the client update, completing the protocol. We observe that the protocol consists of many rounds, most of which are associated with ordering incoming messages; this is the price to achieve protocol separation.

4.4 The BLink Protocol

To achieve high performance over the low-bandwidth links characteristic of wide-area networks, our architecture requires an efficient mechanism for passing messages between logical machines. As described in Section 4.3, each LM is implemented by a replicated group of physical machines, some of which may be faulty. Faulty servers may fail to send, receive, and/or disseminate wide-area messages. Existing protocols that use state machine based logical machines (e.g., [46, 48, 52]) overcome this problem by redundantly sending all messages between logical machines. For example, in a system tolerating f faults, each of $f + 1$ servers in the sending LM might send the outgoing message to $f + 1$ servers in the receiving LM. While this overhead may be acceptable in high-bandwidth LANs or systems supporting a small number of faults, the approach (or even one with $O(f)$ overhead) is poorly suited to large-scale wide-area deployments.

Steward [16] avoids sending redundant messages during normal-case operation by choosing one server (the site representative) to send outgoing messages. Steward employs a coarse-grained mechanism to monitor the performance of the representative, using a lack of global progress to signal that the representative *may* be acting faulty and should be replaced. This approach has two undesired consequences: timeouts for detecting faulty behavior can be significantly higher than they need to be, and the communication protocol is (1) not generic and (2) tightly coupled with global and local protocols, making it unusable in our customizable architecture.

In this section we present the *Byzantine Link* protocol (BLink), a new Byzantine fault-tolerant protocol that allows logical machines to efficiently communicate with each other over the wide-area

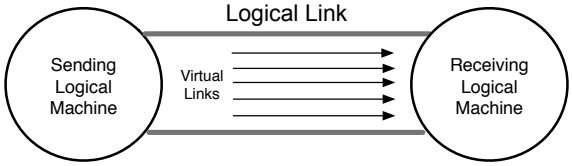

Figure 4.2: A logical link in the (Byzantine, Byzantine) case is constructed from $(3F_A + 1) \cdot (3F_B + 1)$ virtual links. Each virtual link consists of a forwarder and a peer. At any time, one virtual link is used to send messages on the logical link. A virtual link that is diagnosed as potentially faulty is replaced.

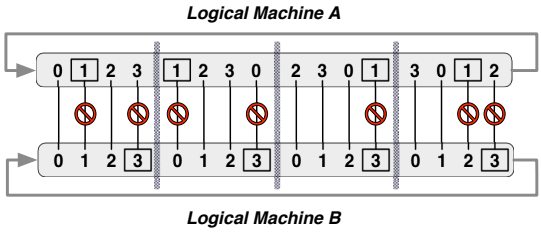

Figure 4.3: An example BLink logical link and selection order, with $F_A = F_B = 1$. Numbers refer to server identifiers. Boxed servers are faulty, and their associated virtual links can be blocked by the adversary. The selection order defines four series, each containing four virtual links. The order repeats after cycling through all four series.

network, regardless of the protocols they are running.[4] BLink consists of several sub-protocols; the sub-protocol deployed between the sending and the receiving logical machines is based on the fault tolerance method employed in each site: (benign, benign), (Byzantine, benign), (benign, Byzantine), and (Byzantine, Byzantine). We first focus on the most challenging case, where each LM runs a Byzantine fault-tolerant protocol. We then describe the other sub-protocols in Section 4.4.2.

4.4.1 (Byzantine, Byzantine) Sub-protocol

BLink establishes a reliable communication link between two LMs using three techniques. The first technique provides a novel way of delegating the responsibility for wide-area communication such that (1) messages are normally sent only once and (2) the adversary is unable to repeatedly block communication between two logical machines. The second technique leverages the power of threshold cryptography and state machine replication to allow the servers in the sending LM to monitor the behavior of the link and take action if it appears to be faulty. The third technique ensures fairness by preventing the adversary from starving any particular link.

Delegating Communication Responsibility: BLink constructs a set of *logical links* from each LM to its neighboring LMs. These logical links are reliable, masking faulty behavior at both the sending and receiving LMs. To support this abstraction, BLink defines a set of *virtual links*, each

[4]The term "link" refers to the logical communication link established between LMs. In particular, BLink operates over UDP.

consisting of one server (the *forwarder*) from the sending LM and one server (the *peer*) from the receiving LM. The servers on a virtual link form a (forwarder, peer) pair. The forwarder sends outgoing wide-area messages to the peer, and the peer disseminates incoming messages to the other servers in the receiving LM. The BLink logical link is shown in Figure 4.2.

For each outgoing logical link, the sending LM delegates communication responsibility to the forwarder of one of its virtual links. This decision is made independently for each outgoing logical link; different servers may act as forwarder on different logical links, and the same server may act as forwarder on multiple logical links. Since either the forwarder or the peer may be faulty, the other servers within the sending LM monitor the performance of the virtual link and move to the next virtual link (electing the next forwarder) if the current forwarder is not performing well enough (we define this notion more precisely below).

The properties of the logical link are based on (1) how one defines the set of virtual links that compose the logical link and (2) the order through which the sending LM proceeds through the virtual links in this set. We consider the logical link between two logical machines, LM_A and LM_B, with $f_A \geq f_B$. In our construction, the set of virtual links in each logical link is simply the set of all $A' \cdot B'$ possible virtual links constructed by choosing a server in LM_A and a server in LM_B. We define a *selection order* for virtual links as an infinite sequence $\langle v_0, \dots \rangle$ of virtual links; the LM cycles through the set of virtual links according to this sequence.

We now describe the selection order used by our logical links. In what follows, $\mathrm{LCM}(x,\ y)$ denotes the least common multiple of x and y, and $\mathrm{GCD}(x,\ y)$ denotes the greatest common divisor of x and y. We define A' *series* of virtual links, each series indexed by $s \in \{0, \dots, A' - 1\}$. Within each series, there are $\mathrm{LCM}(A', B')$ virtual links, with each virtual link in the series indexed by $i \in \{0, \dots, \mathrm{LCM}(A', B') - 1\}$. We denote virtual link i in series s as $L_{s,i}$ and define it to connect the server in LM_A with server id $i + s \bmod A'$ to the server in LM_B with server id $i \bmod B'$. We say that $L_{s,i} = L_{t,j}$ if the two virtual links connect the same pair of servers. Our construction uses the following selection order P:

$$v_{0 \leq i} = L_{\lfloor i / \mathrm{LCM}(A', B') \rfloor, i \bmod \mathrm{LCM}(A', B')}$$

Thus, our protocol selects virtual links by taking series in ascending numerical order modulo A', starting with series 0, and within each series taking the virtual links in ascending numerical order. Figure 4.3 depicts an example with two logical machines, each with four servers, one of which may be faulty. The selection order defines four series, each with four virtual links. Note that the servers in LM_B wrap around modulo 4, while the servers in LM_A "shift" by one position from one series to the next.

We state the following properties regarding the selection order P (proofs are provided in Section 4.4.3):

PROPERTY 4.4.1 $\forall_s \forall_i \forall_{j \neq i}$ *if* $s \in \{0, \ldots, A' - 1\}$ *and* $i, j \in \{0, \ldots, \mathrm{LCM}(A', B') - 1\}$, *then* $L_{s,i} \neq L_{s,j}$. *In other words, each series consists of* $\mathrm{LCM}(A',\ B')$ *distinct virtual links*.

PROPERTY 4.4.2 $\forall_s \forall_{t \neq s} \forall_i \forall_j$, *where* $s, t \in \{0, \ldots, A' - 1\}$ *and* $i, j \in \{0, \ldots, \mathrm{LCM}(A', B') - 1\}$, *if* $s \not\equiv t \pmod{\mathrm{GCD}(A', B')}$ *then* $L_{s,i} \neq L_{t,j}$. *In other words, if the indices of two series* s *and* t *are not congruent modulo* $\mathrm{GCD}(A',\ B')$, *then* s *and* t *contain disjoint sets of virtual links*.

PROPERTY 4.4.3 *For all* s, *the set* $S = \{s \bmod A', \ldots, (s + \mathrm{GCD}(A', B') - 1) \bmod A'\}$ *of series contains* $A' \cdot B'$ *disjoint virtual links. In other words, proceeding through any set of* $\mathrm{GCD}(A',\ B')$ *consecutive series cycles through the set of all virtual links*.

Given Properties 4.4.1 - 4.4.3, we prove the following claim about the ratio of correct virtual links (i.e., virtual links where both forwarder and peer are correct) to faulty links:

Claim 4.4.1 *In the (Byzantine, Byzantine) sub-protocol, the selection order* P *consists of consecutive blocks of* $A' \cdot B'$ *virtual links, and in each block the fraction of correct virtual links is at least* $4/9$.

Proof: In the (Byzantine, Byzantine) case, we have two logical machines, LM_A and LM_B, where LM_A has $A' = 3f_A + 1$ servers, and LM_B has $B' = 3f_B + 1$ servers. By construction, each block consists of $\mathrm{GCD}(A', B')$ consecutive series modulo A' and hence Property 4.4.3 applies to each

block. Consider any block. By Property 4.4.3, all $A' \cdot B'$ possible distinct virtual links are used exactly once in the block.

Assume that f_A servers in LM_A are faulty and f_B servers in LM_B are faulty. There are $f_A(3f_B + 1)$ virtual links that have a faulty server from LM_A. There are $f_B(3f_A + 1)$ virtual links that have a faulty server from LM_B. There are $f_A f_B$ virtual links that have both a faulty server from LM_A and a faulty server from LM_B. Taking into account the virtual links with two faulty servers, there are $f_A(3f_B + 1) + f_B(3f_A + 1) - f_A f_B$ virtual links with at least one faulty server. Let b be the fraction of virtual links with at least one faulty server. Then:

$$\begin{aligned} b &= \frac{f_A(3f_B + 1) + f_B(3f_A + 1) - f_A f_B}{(3f_A + 1)(3f_B + 1)} \\ &= \frac{5f_A f_B + f_A + f_B}{9f_A f_B + 3f_A + 3f_B + 1} \\ &\leq 5/9 \end{aligned}$$

This completes the proof. □

In addition to the ratio of correct virtual links to faulty virtual links, we are also interested in the maximum number of consecutive faulty links through which the LM must cycle before reaching a correct virtual link. We refer to this value as VL_{max}. In Section 4.4.3, we show that VL_{max} is bounded at $2f_A$.

Reliability and Monitoring: BLink uses threshold-signed, cumulative acknowledgements to ensure reliability. Each message sent on an outgoing logical link is assigned a link-specific sequence number. Assigning these sequence numbers consistently is simple, since outgoing messages are generated in response to events totally-ordered by the LM and can be sequenced using this total order. Each LM periodically generates a threshold-signed acknowledgement message, which contains, for each logical link, the sequence number through which the LM has received all previous messages. The generation of the acknowledgement is triggered by executing an LM timeout, as described in Section 4.3. Servers could also piggyback acknowledgements on regular outgoing messages for more timely, fine-grained feedback. The peer server for each incoming logical link

sends the acknowledgement to its corresponding forwarder, which presents the acknowledgement to the servers in the sending LM.

The acknowledgement serves two purposes. First, it is used to determine which messages need to be retransmitted over the link to achieve reliability. This reliability is guaranteed even if the current forwarder is replaced, since the next forwarder knows exactly which messages remain unacknowledged and should be resent. Second, the servers in the sending LM use the acknowledgement to evaluate the performance of the current forwarder. Each server in the sending LM maintains a queue of the unacknowledged messages on each logical link, placing an LM timeout on the acknowledgement of the first message in the queue. If, before the timeout expires, the forwarder presents an acknowledgement indicating the message was successfully received by the receiving LM, the timeout is canceled and a new timeout is set on the next message in the queue. However, if the timeout expires before such an acknowledgement is received, the servers suspect that the virtual link is faulty and elect the next forwarder. This mechanism can be augmented to enforce a higher throughput of acknowledged messages by placing a timeout on a batch of messages. Of course, BLink does not guarantee delivery when a site at one or both ends of the logical link is Byzantine.

Fairness: The third technique used by BLink addresses the dependency between the evaluation of the virtual link forwarder and the performance of the leader of the agreement protocol in the receiving LM. Intuitively, if the leader in the receiving LM could selectively refuse to order certain messages or could delay them too long, then a correct forwarder (in the sending LM) might not be able to collect an acknowledgement in time to convince the other servers that it sent the messages correctly. We would like to settle on a correct virtual link to the extent possible, and thus we augment the agreement protocol with a fairness mechanism.

When a peer receives an incoming message, it disseminates the message within the site; all servers then forward the message to the leader of the agreement protocol and expect it to initiate the message for ordering such that the message can be executed by the LM. To ensure fairness, servers must place a timeout on the leader of the agreement protocol to prevent the selective starvation of a particular incoming logical link. Servers within the LM maintain a queue for each incoming logical link. When the leader receives a message to be ordered, it places the message on the appropriate

queue. The leader then attempts to order messages off of the queues in round-robin fashion. Since incoming link messages have link-based sequence numbers, all servers know which message should be the next one ordered for each link. Thus, upon receiving the next message on a link, a server places a timeout on the message and attempts to replace the leader if the message is not ordered in time. We describe our mechanism for preventing the starvation of any particular client in Section 4.5.

4.4.2 Other BLink Sub-protocols

We now consider the problem of inter-LM communication when one or both of the LMs is implemented using a benign fault-tolerant state machine replication protocol. We first consider the (benign, Byzantine) and (Byzantine, benign) cases. As in the (Byzantine, Byzantine) case, the number of virtual links that compose each logical link is equal to the number of servers in LM_A times the number of servers in LM_B. In the following discussion, we assume that the number of servers in LM_A, A', is greater than or equal to the number of servers, in LM_B, B'. If LM_A runs a Byzantine fault-tolerant protocol and LM_B runs a benign fault-tolerant protocol, then $3f_A + 1 \geq 2f_B + 1$. Otherwise, we have $2f_A + 1 \geq 3f_B + 1$.

We use the same selection order as for the (Byzantine, Byzantine) case, and we use an argument similar to the one found in Section 4.4.1 to obtain the ratio of correct to faulty virtual links. In Section 4.4.3, we show that at least $1/3$ of the virtual links are correct. Further, when LM_A is Byzantine fault-tolerant, the maximum number of consecutive faulty links (VL_{max}) is bounded at $\max(\lfloor 2.5f_A \rfloor,\ 3f_A - 2)$; when LM_A is benign fault-tolerant, VL_{max} is bounded at $\max(2f_A - 1,\ \lfloor \frac{5}{3} f_A \rfloor)$. Intuitively, the difference in the bounds is attributed to the difference in the ratio of faulty servers within LM_B: when LM_B is Byzantine, the ratio is only $1/3$, but when LM_B is benign, the ratio is $1/2$.

In the (benign, benign) case, each logical link consists of $(2f_A+1)\cdot(2f_B+1)$ virtual links. This yields a ratio of $1/4$ correct virtual links. Since no server in either LM is Byzantine, it is possible to use a simple and efficient selection order to cycle through the virtual links. The approach assumes that the correct servers in the sending LM can communicate equally well with the correct servers

Sub-protocol	Correct links	VL_{max} upper bound
(Byz, Byz)	$4/9$	$2F_A$
(Byz, Benign)	$1/3$	$\max(\lfloor 2.5F_A \rfloor, 3F_A - 2)$
(Benign, Byz)	$1/3$	$\max(\lfloor 2.5F_A \rfloor, 3F_A - 2)$
(Benign, Benign)	$1/4$	$F_A + F_B$

Table 4.1: The ratio of correct virtual links and the maximum number of consecutive faulty virtual links for each BLink sub-protocol.

in the receiving LM. This assumption implies that there is no need for the sending LM to replace a correct forwarder. The sending LM thus allows its forwarder to try different peers until it establishes a correct virtual link. The forwarder will need to cycle through at most $f_B + 1$ such peers before finding a correct one. The servers in the sending LM can use a standard ping/Hello protocol to monitor the status of the current forwarder. A server only votes to replace the forwarder if it has not received a response from the forwarder within a timeout period.

When a forwarder detects that a peer is faulty, it locally broadcasts a message indicating that the peer should be skipped by other forwarders. The next forwarder then picks up where the last forwarder left off. In this way, one can think of the logical machine as rotating through a single sequence of peers. Note that subsequent forwarders may eventually send to peers that were previously diagnosed as faulty, because a correct peer may be diagnosed as faulty due to a transient network partition. In Section 4.4.3, we show that VL_{max} is bounded at $f_A + f_B$. We can use a similar strategy in the (benign, Byzantine) case; however, the technique is not applicable to the (Byzantine, benign) case, since the forwarder cannot be trusted to find a correct peer. We summarize our results in Table 4.1.

4.4.3 BLink Protocol Proofs

We now prove several claims about the BLink protocol. In Section 4.4.4, we prove several properties of the selection order used in the BLink protocol. In Section 4.4.5, we bound the maximum number of consecutive virtual links the adversary can block in the (Byzantine, Byzantine) sub-protocol. In Section 4.4.6, we show the ratio of correct to faulty links in the (Byzantine, benign), (benign, Byzantine), and (benign, benign) sub-protocols. In Section 4.4.7 we bound VL_{max} in the (Byzantine, benign), (benign, Byzantine), and (benign, benign) sub-protocols.

4.4.4 Proof of Selection Order Properties

In this section, we first prove Properties 4.4.1-4.4.3 of the BLink protocol, which were stated in Section 4.4. We then prove Lemma 4.4.1, which will be used in subsequent sections.

PROPERTY 5.1 $\forall_s \forall_i \forall_{j \neq i}$ *if* $s \in \{0, \ldots, A' - 1\}$ *and* $i, j \in \{0, \ldots, \mathrm{LCM}(A', B') - 1\}$, *then* $L_{s,i} \neq L_{s,j}$. *In other words, each series consists of* $\mathrm{LCM}(A',\ B')$ *distinct virtual links*.

Proof: Suppose not. $\exists_s \exists_i \exists_{j \neq i}$ such that $L_{s,i} = L_{s,j}$. By the definition of virtual links, $L_{s,i}$ consists of the server in LM_A with server id $i + s \bmod A'$ and the server in LM_B with server id $i \bmod B'$. Similarly, $L_{s,j}$ consists of the server in LM_A with server id $j + s \bmod A'$ and the server in LM_B with server id $i \bmod B'$.

Because $L_{s,i} = L_{s,j}$, we have $i \equiv j \pmod{B'}$ and $i + s \equiv j + s \pmod{A'}$. We can rewrite the latter as $i \equiv j \pmod{A'}$. Because i and j are congruent modulo A' and modulo B', we can conclude that $i \equiv j \pmod{\mathrm{LCM}(A', B')}$. Because $i \neq j$ we can also conclude that $|i - j| >= \mathrm{LCM}(A', B')$.

We now have a contradiction because i and j are each in the interval $[0 \ldots \mathrm{LCM}(A', B') - 1]$, which is of length $\mathrm{LCM}(A', B') - 1$. □

PROPERTY 5.2 $\forall_s \forall_{t \neq s} \forall_i \forall_j$, *where* $s, t \in \{0, \ldots, A' - 1\}$ *and* $i, j \in \{0, \ldots, \mathrm{LCM}(A', B') - 1\}$, *if* $s \not\equiv t \pmod{\mathrm{GCD}(A', B')}$ *then* $L_{s,i} \neq L_{t,j}$. *In other words, if the indices of two series* s *and* t *are not congruent modulo* $\mathrm{GCD}(A',\ B')$, *then* s *and* t *contain disjoint sets of virtual links*.

Proof: Suppose not. $\exists_s \exists_{t \neq s} \exists_i \exists_j$ such that $s \not\equiv t \pmod{\mathrm{GCD}(A', B')}$ and $L_{s,i} = L_{t,j}$. By the definition of virtual links, $L_{s,i}$ consists of the server in LM_A with server id $i + s \bmod A'$ and the server in LM_B with server id $i \bmod B'$. Similarly, $L_{t,j}$ consists of the server in LM_A with server id $j + t \bmod A'$ and the server in LM_B with server id $i \bmod B'$.

Because $L_{s,i} = L_{t,j}$, we have $i \equiv j \pmod{B'}$ and $i + s \equiv j + t \pmod{A'}$. We can rewrite the latter as $i \equiv j + t - s \pmod{A'}$. Because $\mathrm{GCD}(A', B')$ divides both A' and B', we can replace the two congruences with $i \equiv j \pmod{\mathrm{GCD}(A', B')}$ and $i \equiv j + t - s \pmod{\mathrm{GCD}(A', B')}$. By transitivity of congurence modulo $\mathrm{GCD}(A', B')$, we have that $j \equiv j + t - s \pmod{\mathrm{GCD}(A', B')}$.

Adding $s - j$ to both sides, we get $s \equiv t \pmod{\mathrm{GCD}(A', B')}$. This is the contradiction that we were seeking. □

PROPERTY 5.3 *For all s, the set $S = \{s \bmod A', \ldots, (s + \mathrm{GCD}(A', B') - 1) \bmod A'\}$ of series contains $A' \cdot B'$ disjoint virtual links. In other words, proceeding through any set of $\mathrm{GCD}(A',\ B')$ consecutive series cycles through the set of all virtual links.*

Proof: First we show that all of the virtual links in all series in S are disjoint. Consider two virtual links v_1 and v_2. If v_1 and v_2 are in the same series, the property follows from Property 4.4.1. If instead v_1 and v_2 are in different series in S, the property follows from Property 4.4.2.

Now we show that there are $A' \cdot B'$ virtual links. The number of series in S is $\mathrm{GCD}(A', B')$ and the number of virtual links per series is $\mathrm{LCM}(A', B')$. Because the virtual links are disjoint, the total number is the product, which is $\mathrm{GCD}(A', B') \cdot \mathrm{LCM}(A', B') = A' \cdot B'$ □

Lemma 4.4.1 *Any $A' - 1$ successive virtual links in the selection order P will use disjoint servers from LM_A.*

Proof: Recall that by definition each series contains $\mathrm{LCM}(A', B')$ virtual links. By construction of the selection order P and because $A' \leq \mathrm{LCM}(A', B')$, we have that $A' - 1$ successive virtual links are either drawn from one series or from two consecutive series. Consider the two cases.

If the $A' - 1$ virtual links are drawn from one series, the result is immediate because the virtual links within a series take servers from LM_A in increasing consecutive numerical order modulo A'.

Assume instead that the $A' - 1$ virtual links are drawn from two consecutive series. Within the first series, servers are selected from LM_A in increasing consecutive numerical order modulo A'. By the way that P is constructed, exactly one server from LM_A is skipped at the boundary between series. In the second series, servers continue to be selected from LM_A in increasing consecutive numerical order modulo A'. Thus we can take $A' - 1$ servers from LM_A with no repeats. □

4.4.5 Bounding VL_{max} in the (Byzantine, Byzantine) Sub-protocol

We now prove a bound on VL_{max}, the worst-case number of faulty virtual links through which a logical machine must cycle before reaching a correct virtual link, in the (Byantine, Byzantine)

BLink sub-protocol. In what follows, we consider the logical link between two logical machines, LM_A and LM_B, where LM_A has $A' = 3f_A + 1$ servers, and LM_B has $B' = 3f_B + 1$ servers, with $f_A \geq f_B$.

We first derive an equation for how to compute the value of VL_{max}. In our selection order P, the servers in LM_B wrap around modulo B' on successive virtual links. Given this property, we consider the strategy taken by the adversary to maximize VL_{max} across a run of successive virtual links in which none of the servers from LM_A is used more than once. Since none of the A' servers from LM_A repeats, the adversary has a pool of at most f_A faulty servers from LM_A that it can use to block a virtual link. In any set of B' successive virtual links from this run, the adversary can block f_B virtual links by using a faulty server from LM_B, and must then consume $2f_B + 1$ out of the f_A faulty servers in its pool from LM_A. There are $\lfloor \frac{f_A}{2f_B+1} \rfloor$ such complete blocks of $3f_B + 1$. In addition, the adversary can then use the f_B servers from LM_B once more, plus any remaining servers from LM_A. Thus, we can use the following equation to represent the maximum number of consecutive virtual links the adversary can block:

$$VL_{max} \quad = \quad \lfloor \frac{f_A}{2f_B + 1} \rfloor (3f_B + 1) + f_B + f_A \bmod (2f_B + 1) \tag{4.1}$$

The following lemma bounds Equation 4.1:

Lemma 4.4.2 *In the (Byzantine, Byzantine) sub-protocol, Equation 4.1 is bounded above by* $2f_A$.

Proof: We consider two cases. In the first case, let $f_A < 2f_B + 1$. Then the first term of Equation 4.1 becomes 0. Since $f_B \leq f_A$, we have:

$$\begin{aligned} VL_{max} &= 0 + f_B + f_A \\ &= f_B + f_A \\ &\leq f_A + f_A \\ &= 2f_A \end{aligned}$$

In the second case, let $f_A \geq 2f_B + 1$. We first rewrite Equation 4.1 by removing the floor from the first term.

$$VL_{max} = \frac{f_A}{2f_B+1}(3f_B+1) - \frac{f_A \bmod (2f_B+1)}{2f_B+1}(3f_B+1) + f_B + f_A \bmod (2f_B+1)$$

We can ignore the terms that are modulo $2f_B+1$, since we subtract at most $1.5(f_A \bmod (2f_B+1))$ and add $f_A \bmod (2f_B+1)$, resulting in a net subtraction. Noting that $f_B \leq \frac{f_A-1}{2} < \frac{f_A}{2}$, we have:

$$\begin{aligned} VL_{max} &< 1.5f_A + f_B \\ &< 1.5f_A + \frac{f_A}{2} \\ &= 2f_A \end{aligned}$$

Since these are the only two possible cases, this completes the proof. □

Claim 4.4.2 *Let LM_A and LM_B be two Byzantine fault-tolerant logical machines, with $f_A \geq f_B$, which use the selection order P. Then VL_{max} is bounded above by $2f_A$.*

Proof: Consider the set of selection orders O in which the servers from LM_B wrap around modulo B'. Lemma 4.4.2 implies that the maximum number of consecutive virtual links the adversary can block in a run in which no server from LM_A is repeated is $2f_A$. This property holds regardless of

how the servers from LM_A are assigned within this run, as long as they do not repeat. The bound holds for any selection order from O in which the servers in LM_A do not repeat within $2f_A$ virtual links. By Lemma 4.4.1, any $A' - 1 = 3f_A$ successive virtual links in the selection order P will use disjoint servers from LM_A. Thus, Lemma 4.4.2 applies to the selection order P, completing the proof. □

4.4.6 Ratio of Correct Links in Other Sub-protocols

We now show the ratio of correct links to faulty links in the (Byzantine, benign), (benign, Byzantine), and (benign, benign) sub-protocols.

In the (Byzantine, benign) and (benign, Byzantine) sub-protocols, there are two cases to consider. When the logical machine with a larger number of servers (i.e., LM_A) is Byzantine fault-tolerant, we have that $3f_A + 1 \geq 2f_B + 1$. In this case, there are $f_A(2f_B + 1)$ virtual links with a faulty server from LM_A, and there are $f_B(3f_A + 1)$ virtual links with a faulty server from LM_B. There are $f_A f_B$ virtual links that have a faulty server from LM_A and a faulty server from LM_B. Thus, there are $f_A(2f_B+1) + f_B(3f_A+1) - f_A f_B$ virtual links with at least one faulty server. Let b be the fraction of virtual links with at least one faulty server. Then:

$$\begin{aligned} b &= \frac{f_A(2f_B + 1) + f_B(3f_A + 1) - f_A f_B}{(3f_A + 1)(2f_B + 1)} \\ &= \frac{4f_A f_B + f_A + f_B}{6f_A f_B + 3f_A + 2f_B + 1} \\ &\leq 2/3 \end{aligned}$$

The second case, when LM_A is benign fault-tolerant, is similar, except that there can be $f_A(3f_B + 1) + f_B(2f_A + 1) - f_A f_B$ virtual links with at least one faulty server. This yields:

$$
\begin{aligned}
b &= \frac{f_A(3f_B+1)+f_B(2f_A+1)-f_Af_B}{(2f_A+1)(3f_B+1)} \\
&= \frac{4f_Af_B+f_A+f_B}{6f_Af_B+2f_A+3f_B+1} \\
&\leq 2/3
\end{aligned}
$$

Thus, in both the (Byzantine, benign) and (benign, Byzantine) sub-protocols, at least $1/3$ of the virtual links are correct.

In the (benign, benign) sub-protocol, there are $f_A(2f_B+1)$ virtual links with a faulty server from LM_A, and there are $f_B(2f_A+1)$ virtual links with a faulty server from LM_B. There are f_Af_B virtual links that have a faulty server from LM_A and a faulty server from LM_B. Thus, there are $f_A(2f_B+1)+f_B(2f_A+1)-f_Af_B$ virtual links with at least one faulty server. Thus:

$$
\begin{aligned}
b &= \frac{f_A(2f_B+1)+f_B(2f_A+1)-f_Af_B}{(2f_A+1)(2f_B+1)} \\
&= \frac{3f_Af_B+f_A+f_B}{4f_Af_B+2f_A+2f_B+1} \\
&\leq 3/4
\end{aligned}
$$

Thus, in the (benign, benign) sub-protocol, at least $1/4$ of the virtual links are correct.

4.4.7 Bounding VL_{max} in the Other Sub-protocols

In the (Byzantine, benign) and (benign, Byzantine) sub-protocols, we bound VL_{max} using a similar technique to the one found in Section 4.4.5. As usual, we let LM_A be the logical machine with a larger number of machines. We consider two cases. First, let LM_A run a benign fault-tolerant local protocol, and let LM_B be Byzantine fault-tolerant. Then $2f_A+1 \geq 3f_B+1$. We obtain the same formula for VL_{max} as in Equation 4.1. We obtain the following result:

Lemma 4.4.3 *In the (Byzantine, benign) and (benign, Byzantine) sub-protocols, when LM_A runs a benign fault-tolerant protocol and LM_B runs a Byzantine fault-tolerant protocol, Equation 4.1 is*

bounded above by $\max(\lfloor 5f_A/3 \rfloor, 2f_A - 1)$.

Proof: We consider two sub-cases. In the first sub-case, let $f_A < 2f_B+1$. Since $2f_A+1 \geq 3f_B+1$, we have that $f_B \leq 2f_A/3$. Using similar algebra as in the first case of Section 4.4.5, we have:

$$\begin{aligned} VL_{max} &= \lfloor \frac{f_A}{2f_B+1} \rfloor (3f_B+1) + f_B + f_A \bmod (2f_B+1) \\ &= f_B + f_A \\ &\leq 2f_A/3 + f_A \\ &= 5f_A/3 \end{aligned}$$

In the second sub-case, let $f_A \geq 2f_B + 1$. We use an identical argument to the second case of Section 4.4.5 to obtain $VL_{max} < 2f_A$. □

We can now prove the following claim:

Claim 4.4.3 *Let LM_A and LM_B be two logical machines, where LM_A runs a benign fault-tolerant protocol and LM_B runs a Byzantine fault-tolerant protocol, with $2f_A+1 \geq 3f_B+1$. If both logical machines use the selection order P, then VL_{max} is bounded above by* $\max(\lfloor 5f_A/3 \rfloor, 2f_A - 1)$.

Proof: The bound established in Lemma 4.4.3 holds for any selection order in which (1) the servers from LM_B wrap around modulo B' and (2) the servers from LM_A do not repeat within $\max(\lfloor 5f_A/3 \rfloor, 2f_A - 1)$ virtual links. By Lemma 4.4.1, any $A' - 1 = 2f_A$ successive virtual links in the selection order P will use disjoint servers from LM_A. Since $2f_A$ is always greater than $\max(\lfloor 5f_A/3 \rfloor, 2f_A - 1)$, Lemma 4.4.3 applies to the selection order P, completing the proof. □

We now consider the second case, when LM_A runs a Byzantine fault-tolerant protocol and LM_B runs a benign fault-tolerant protocol. Then $3f_A + 1 \geq 2f_B + 1$. Our analysis yields a slightly different equation, since within each block of $2f_B + 1$ virtual links, the adversary must consume $f_B + 1$ faulty servers from LM_A. This produces the following modified equation for VL_{max}:

$$VL_{max} = \lfloor \frac{f_A}{f_B+1} \rfloor (2f_B+1) + f_B + f_A \bmod (f_B+1) \quad (4.2)$$

Lemma 4.4.4 *In the (Byzantine, benign) and (benign, Byzantine) sub-protocols, when LM_A runs a Byzantine fault-tolerant protocol and LM_B runs a benign fault-tolerant protocol, Equation 4.2 is bounded above by* $\max(\lfloor 2.5f_A \rfloor, 3f_A - 2)$.

Proof: We consider two sub-cases. In the first sub-case, let $f_A < f_B+1$. Since $3f_A+1 \geq 2f_B+1$, we have that $f_B \leq 1.5f_A$. The floor terms becomes 0, so we have:

$$\begin{aligned} VL_{max} &= 0 + f_B + f_A \bmod (f_B+1) \\ &= f_B + f_A \\ &\leq 1.5f_A + f_A \\ &= 2.5f_A \end{aligned}$$

In the second sub-case, let $f_A \geq f_B + 1$. We first remove the floor from Equation 4.2:

$$VL_{max} = \frac{f_A}{f_B+1}(2f_B+1) - \frac{f_A \bmod (f_B+1)}{f_B+1}(2f_B+1) + f_B + f_A \bmod (f_B+1)$$

We can ignore the terms that are modulo $f_B + 1$, since we subtract at most $2(f_A \bmod (f_B+1))$ and add $f_A \bmod (f_B+1)$, resulting in a net subtraction. Noting that $f_B \leq f_A - 1$, we have:

$$\begin{aligned} VL_{max} &< 2f_A + f_B \\ &\leq 2f_A + f_A - 1 \\ &= 3f_A - 1 \end{aligned}$$

This completes the proof. □

Claim 4.4.4 *Let LM_A and LM_B be two logical machines, where LM_A runs a Byzantine fault-tolerant protocol and LM_B runs a benign fault-tolerant protocol, with $3f_A + 1 \geq 2f_B + 1$. If both logical machines use the selection order P, then VL_{max} is bounded above by* $\max(\lfloor 2.5f_A \rfloor, 3f_A - 2)$.

Proof: The bound established in Lemma 4.4.4 holds for any selection order in which (1) the servers from LM_B wrap around modulo B' and (2) the servers from LM_A do not repeat within $\max(\lfloor 2.5f_A \rfloor, 3f_A - 2)$ virtual links. By Lemma 4.4.1, any $A' - 1 = 3f_A$ successive virtual links in the selection order P will use disjoint servers from LM_A. Since $3f_A$ is always greater than $\max(\lfloor 2.5f_A \rfloor, 3f_A - 2)$, Lemma 4.4.4 applies to the selection order P, completing the proof. □

Claim 4.4.5 *Let LM_A and LM_B be two logical machines running the (Byzantine, benign) or (benign, Byzantine) sub-protocols. If both logical machines use the selection order P, then VL_{max} is bounded above by* $\max(\lfloor 2.5f_A \rfloor, 3f_A - 2)$.

Proof: Follows immediately from Claims 4.4.3 and 4.4.4. □

Finally, we argue that when both LM_A and LM_B run a benign fault-tolerant protocol, VL_{max} is bounded at $f_A + f_B$. Using the optimized strategy presented in Section 4.4.2, the adversary can crash f_A consecutive forwarders from the sending logical machine just before the forwarder reports that it is moving to the next peer in the receiving logical machine. After this occurs, the next forwarder elected will be correct, but this forwarder may need to cycle through f_B faulty peers in the receiving logical machine. Thus, VL_{max} is bounded at $f_A + f_B$.

4.5 Client Updates

Our architecture guarantees that if the system is stable and a client is connected to a stable site, the client will be able to order its update. Since BLink provides efficient communication between logical machines, it is technically possible to treat each client as a non-replicated logical machine and use BLink to provide Byzantine fault-tolerant communication between clients and

logical machines consisting of servers. However, using BLink in this manner requires (1) extra overhead that increases normal-case latency, (2) sending threshold-signed acknowledgements from the LM to the client, and (3) a separate queue for each client. Therefore, our architecture includes a specialized protocol, CLink, which guarantees that clients will be able to efficiently and quickly inject updates into the system. CLink only guarantees that a *logical machine* will order the client's update. Once this occurs, the wide-area protocol running on the logical machines uses techniques similar to BFT to guarantee global ordering.

CLink consists of two sub-protocols, which we refer to as *CLink-Opt* and *CLink-Order*. CLink-Opt is invoked only by servers in non-leader logical machines. Upon receiving an update from a local client, a server in a non-leader LM attempts to optimistically forward the update to the leader LM, without locally ordering the update or performing any cryptographic operation on it. This saves in both computation and latency. CLink-Order can be invoked by any server in the system. The protocol guarantees that the logical machine to which a local client is connected will locally order the client's updates. We now describe the two sub-protocols in more detail.

CLink-Opt: Each client maintains a list of the servers in its local site. At a given time, the client sends its updates to the server at the front of the list and receives replies from this server after the update is globally ordered. After sending an update to its local server, the client sets a timeout. If the client does not receive a reply within the timeout period, it retransmits the update to $f + 1$ servers in the local site, ensuring that at least one correct server will receive the update. The retransmitted update is identical to the original update, except that it has a special *retransmit flag*, indicating that the servers receiving the update should invoke CLink-Order (which we describe below). The client also moves to the next server in its list, wrapping around when it reaches the end. The client will first send to this new server when it sends its next update.

Each server in the non-leader LM maintains a list of the servers in the leader LM. Upon receiving an update from a local client, a server forwards the update to the server in the leader LM at the front of its list and sets a timeout. Upon receiving the update, the server in the leader LM invokes CLink-Order to ensure that the update is locally ordered by the leader LM. Optimistic forwarding may fail because a malicious server in the forwarding path drops the message. If the client's local server does

not globally order and execute the update within the timeout period, then it invokes CLink-Order. The server also moves to the next server in its list, wrapping around when it reaches the end of the list. The server will optimistically forward an update to this new server the next time it receives a new update.

CLink-Order: As described above, CLink-Order is a general mechanism that can be used to ensure that a logical machine will locally order a message. CLink-Order is invoked in three cases:

1. A client is connected to a non-leader LM, and the client's initial choice of local server expires its timeout.

2. A server in a non-leader LM receives an update with the *retransmit_flag* set.

3. A server in the leader LM receives a new update.

In the first two cases, the servers in the non-leader LM use CLink-Order, coupled with BLink, to ensure that the update is propagated to the leader site. In the third case, the servers in the leader LM ensure that the update is locally ordered by the leader.

Upon invoking CLink-Order on an update u, a server r generates an *Ordering_Request* message, $OR(id_r,\ u,\ seq_r)$, signs it, and sends it to the other servers in the site. seq_r is a local sequence number, generated by r, that is incremented each time r sends a new Ordering_Request message. When a server receives an Ordering_Request from server r, it stores the message and forwards it to the leader. Additionally, the server sets a timeout on the message if it has executed all Ordering_Request messages from server r up to and including $seq_r - 1$. The server expects the message to be locally ordered within the timeout period.

The leader attempts to locally order the Ordering_Request messages from each server in round-robin fashion. It decides whether to propose an Ordering_Request from a server s for local ordering using the following algorithm. If the leader does not have an Ordering_Request from server s to propose, then it moves to the next server modulo N, where N is the number of servers in the local site. Otherwise, let *Req(*id_s*,* u*,* seq_s*)* be the Ordering_Request message from server s with the lowest sequence number. The leader proposes Req if seq_s is one greater than the sequence number of the last Ordering_Request executed or proposed from s.

When a server executes an Ordering_Request message from server s, it cancels the timeout associated with the execution of that message, if one is set. If the server has the next Ordering_Request message from s, it sets a timeout on that message. Note that, if server s is Byzantine, some server might have received an Ordering_Request message from s with the same sequence number (seq_s) but a different update. In this case, the server cancels its timeout but has explicit proof that s is corrupt and can broadcast the proof to the rest of the servers. We note that the protocol can be optimized by including only a digest of the update in the Ordering_Request message, reducing the amount of bandwidth consumed when more than one server includes the same update in an Ordering_Request.

4.6 Performance Optimizations

Our composable architecture has significant computational overhead, because each LM must order all events that cause state transitions in the wide-area protocol. This Byzantine fault-tolerant ordering (which in our architecture uses digital signatures) is computationally costly. In addition, each LM threshold signs all outgoing messages, which imposes an even greater computational cost. Consequently, we use Merkle hash trees [70] to amortize the cost of threshold signing, and we improve the performance of LM event processing via well-known aggregation techniques. These optimizations are applied *only* to the local protocols. Thus, there is a one-to-one correspondence between wide-area messages in an optimized, composable protocol and its unoptimized equivalent.

Merkle Tree Based Signatures: Instead of threshold signing every outgoing message, we generate a single threshold signature, based on a Merkle hash tree, that is used to authenticate several messages. Each outgoing message is self-contained, including everything necessary for validation (except the public key). The leaf nodes in a Merkle hash tree contain the hashes of the messages that need to be sent. Each of the internal nodes contains a hash of the concatenation of the two hashes in its children nodes. The signature is generated over the hash contained in the root. When a message is sent, we include the series of hashes that can be used to generate the root hash. The number of included hashes is $\log(N)$, where N is the number of messages that were signed with the single signature.

Protocol Rounds			
Protocol	Wide Area	Local Area	Total
Steward	2	4	6
Paxos/Paxos	2	6	8
BFT/Paxos	3	8	11
Paxos/BFT	2	11	13
BFT/BFT	3	15	18

Table 4.2: Normal-case protocol rounds.

Protocol Computational Costs		
Protocol	Threshold RSA Sign	RSA Sign
Steward	1	3
Paxos/Paxos	0	$2 + (S - 1)$
BFT/Paxos	0	$3 + 2(S - 1)$
Paxos/BFT	1	$3 + 2(S - 1)$
BFT/BFT	2	$4 + 4(S - 1)$

Table 4.3: Number of expensive cryptographic operations that each server at the leader site does per update during normal-case operation.

Logical Machine Event Processing: We use the aggregation technique described in [53] to increase the throughput of local event processing by the LM. The LM orders several events at once, allowing the LM to order thousands of events per second over LANs while providing Byzantine fault tolerance. With this performance, it is likely that the incoming wide-area bandwidth will limit throughput.

4.7 Performance Evaluation

To evaluate the performance of our composable architecture, we implemented our protocols, including all necessary communication and cryptographic functionality.

Test Bed and Network Setup: We used a network topology consisting of 5 wide-area sites, each containing 16 physical machines, to quantify the performance of our system. In order to facilitate comparisons with Steward, we chose to use the same topology and numbers of machines used in [16]. If BFT is run within a site, then the site can tolerate up to 5 Byzantine servers. If Paxos is run within a site, then the site can tolerate 7 benign server failures. If BFT is run on the wide area, then the system can tolerate one Byzantine site compromise. If Paxos is run on the wide area, then the system remains available if no more than two sites are disconnected from the others.

Our experimental test bed 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=6, 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 3.4 ms. The leader site was fully deployed on 16 machines, and the other 4 sites were emulated by one computer each.

Each emulating computer performed the role of a representative of a complete 16 server site. Thus, our test bed is 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 also modeled the aggregation used by our composable architecture. We determined busy-wait times for each type of packet by benchmarking the different types of ordering protocols on a fully deployed, 16 server site. The Spines [65, 72] messaging system was used to emulate latency and throughput constraints on the wide-area links. Wide-area links were limited to 10 Mbps in all tests.

We compared the performance results of five protocols, four of which use our composable architecture:
Paxos/Paxos, BFT/Paxos, Paxos/BFT, BFT/BFT. The fifth is a new implementation of Steward, which includes the option of using the same optimization techniques used in the composable architecture. The updates in our experiments carried a payload of 200 bytes, representative of an SQL statement.

We exclusively use RSA signatures for authentication, both for consistency with our previous work and to provide non-repudiation, which is valuable when identifying malicious servers. The benign fault-tolerant protocols use RSA signatures to protect against external attackers. While it is possible to use more efficient cryptography in the compositions based on Paxos, these changes do not significantly affect performance when our optimizations are used. We also note that BFT can use MACs, which improves its latency and results in much better performance when no aggregation is used. However, this change has a smaller effect on our optimized protocols, because the total update latency is dominated by the wide-area latency.

Protocol Rounds and Cryptographic Costs: Table 4.2 shows the number of normal-case protocol rounds, where view changes do not occur, in Steward and in each of the four combinations of our composable architecture. The protocol rounds are classified as wide-area when the message is sent between sites, and local-area when it is sent between two physical machines within a site. The difference in total rounds ranges from 6 (Steward) to 18 (BFT/BFT). However, it is important to observe that all of the protocols listed have either two or three wide-area rounds.

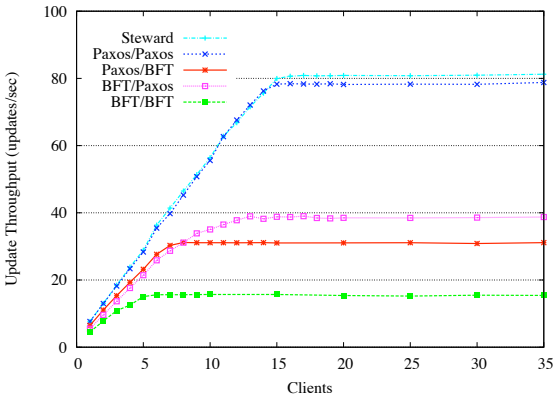

Figure 4.4: Throughput of Unoptimized Protocols, 50 ms Diameter

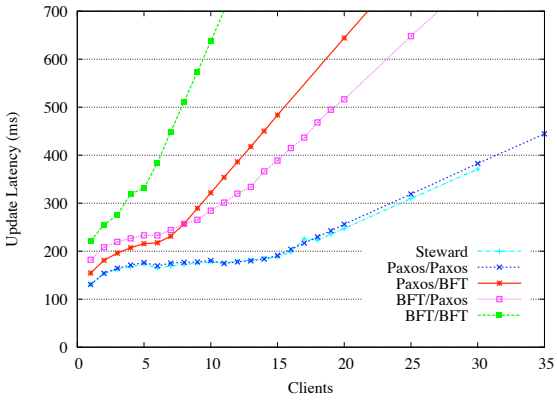

Figure 4.5: Latency of Unoptimized Protocols, 50 ms Diameter

Table 4.3 shows the computationally expensive cryptographic operations required for each update during normal-case operation at the leader site when the optimizations presented in Section 4.6 are not used. The costs are a function of the number of sites, denoted by S. The table shows the number of threshold signatures to which each server in the leader must contribute and the number of RSA signatures that each server in the leader site must compute. In the tests presented in this paper, the unoptimized versions of our algorithm are always limited by computational resources. Consequently, these costs are inversely proportional to the maximum throughput.

Architectural Comparison: To evaluate the overhead of our composable architecture compared to that of Steward, we first compare the performance of the five protocols when the optimizations presented in Section 4.6 are not used. Note that that the unoptimized results do not reflect our architecture's actual performance; we specifically removed the optimizations to provide a clear picture of their benefits. We used a symmetric configuration where all sites are connected to each other with 50 ms (emulating crossing the continental US), 10Mbps links. Each client sends an update to a server in its site, waits for proof that the update was ordered, and then immediately injects the next update.

Figure 4.4 shows update throughput as a function of the number of clients. In all of the protocols, throughput initially increases as the number of clients increases. When the load on the CPU reaches 100%, throughput plateaus. This graph shows the performance benefit of Steward's architecture. In Steward, external wide-area accept messages are not ordered before the replicas process them.

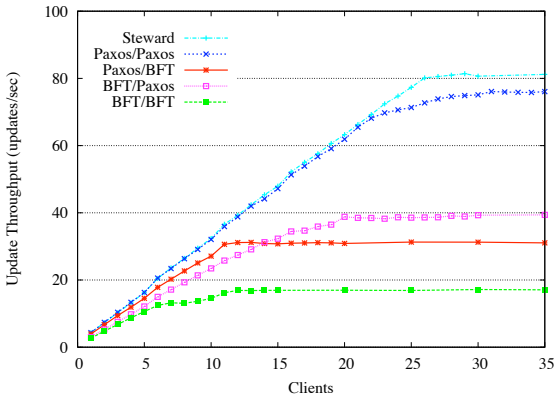

Figure 4.6: Throughput of Unoptimized Protocols, 100 ms Diameter

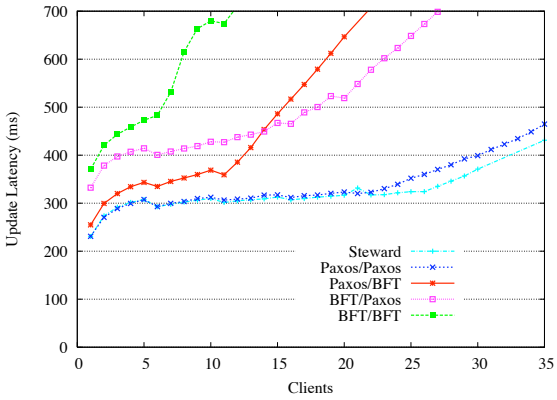

Figure 4.7: Latency of Unoptimized Protocols, 100 ms Diameter

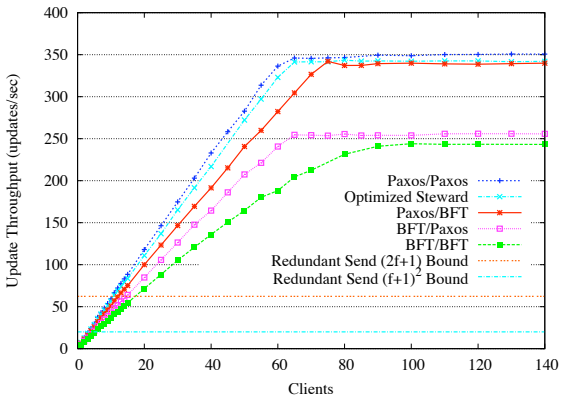

Figure 4.8: Throughput of Optimized Protocols, 50 ms Diameter

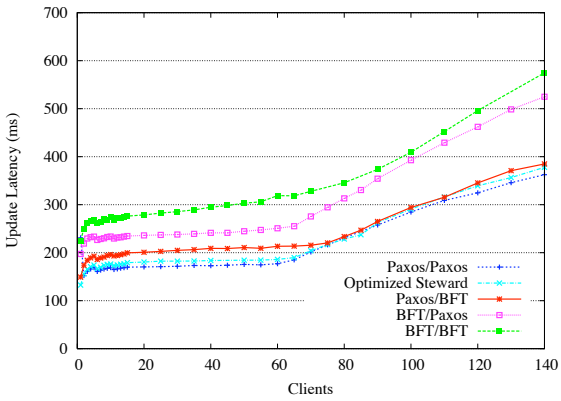

Figure 4.9: Latency of Optimized Protocols, 50 ms Diameter

Steward achieves over twice the performance of Paxos/BFT, its equivalent composition, reflecting the price of clean separation. Steward even outperforms Paxos/Paxos, which has more ordering and RSA signature generation, but does not use threshold signatures. The initial slope of these curves is most dependent on the number of wide-area protocol rounds. The peak performance of each of the protocols is a function of the number of cryptographic operations (see Table 4.3). The Paxos/BFT composition has about twice the throughput of the BFT/BFT composition, and it has approximately half of the cryptographic costs. A similar relationship exists between Paxos/Paxos and BFT/Paxos.

Figure 4.5 shows average update latency measured at the clients as a function of the number of clients. In each of the curves, the update latency remains approximately constant until the CPU is 100% utilized, at which point, latency climbs as the number of clients increases. In our system, we

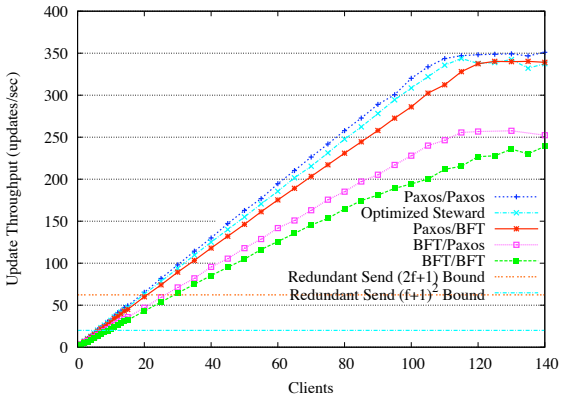

Figure 4.10: Throughput of Optimized Protocols, 100 ms Diameter

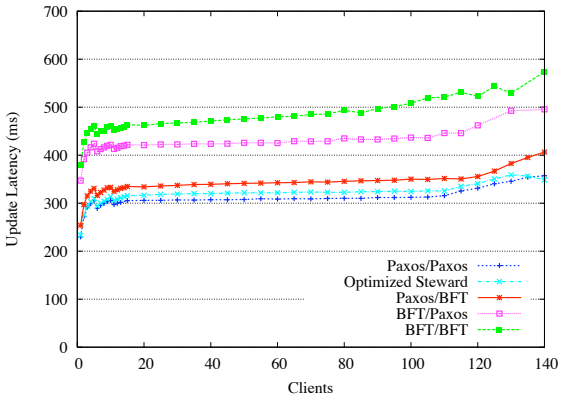

Figure 4.11: Latency of Optimized Protocols, 100 ms Diameter

queue client updates if the system is overburdened and inject these updates in the order in which they were received.

Figures 4.6 and 4.7 show the results for the same tests as above with 100 ms network diameter. We observe the same maximum bandwidth and latency trends. Additional latency on the wide-area links reduces the slope of the lines in Figure 4.6 (update throughput), but has no effect on the maximum throughput that is achieved.

Performance of Optimized Protocols: We now present the performance of the five protocols with the optimizations described in Section 4.6. In these protocols, the cost of the cryptographic operations listed in Table 4.3 are amortized over several updates when CPU load is high. In contrast to the unoptimized protocols, none of our optimized protocols were CPU limited in the following tests. Maximum throughput was always limited by wide-area bandwidth constraints. In all cases, the optimized protocols increased throughput by at least a factor of 4 compared to their unoptimized versions.

In Figures 4.8 and 4.10 (discussed below), we include two theoretical throughput upper bounds of a Paxos/BFT composition in which LMs redundantly send physical messages over the wide area to ensure reliable inter-LM communication. We computed the maximum throughput by assuming that the wide-area Proposal message sent from the leader site contains at least a signed update from the client and an RSA signature from the LM (456 bytes total). We present bounds based on (1) an $(f+1)^2$ protocol where the leader site would need to redundantly send 36 of these messages to

each of the other 4 sites per update and (2) a $(2f + 1)$ peer protocol where the leader site would redundantly send 11 messages to each site per update. The second protocol was included within the original Steward system for use during view changes, but we are unaware of any other systems that use it. The upper bound is the throughput at which the leader site's outgoing link reaches saturation. The difference between the redundant send upper bounds and the performance of Paxos/BFT (with BLink) attests to the importance of the BLink protocol.

Figure 4.8 shows the update throughput as a function of the number of clients. The relative maximum throughput and slopes of the curves are very different from the unoptimized versions. For example, Paxos/Paxos, Steward, and Paxos/BFT have almost the same maximum throughput. This attests to the effectiveness of the optimizations in greatly reducing the performance overhead associated with clean separation. The optimization improves the performance of the compositions more than it improves Steward because the composable architecture uses many more local rounds. In a wide-area environment, local rounds are relatively inexpensive *if* they do not consume too much computational resources. The optimizations eliminate this computational bottleneck. Thus, performance of the optimized version is predominantly dependent on the number of wide-area protocol rounds.

The local-area protocol has a smaller, but significant, effect on performance. The slopes of the curves are different because of the difference in latency contributed by the local-area protocols. BFT and threshold signing contribute the greatest latency. As a result, Steward has a steeper slope than its equivalent composition, Paxos/BFT. Here also, we can see the benefit of Steward, but the performance difference is considerably smaller than in the unoptimized protocols. Paxos contributes very little latency and therefore, Paxos/Paxos's performance slightly exceeds Steward's. Note that Paxos/Paxos benefits slightly more than Steward from the optimizations, because Paxos/Paxos locally orders more messages than Steward (which orders the update locally only once).

Figure 4.9 shows the average update latency in the same experiment. Although aggregation is commonly associated with an increase in latency, the optimized protocols have similar or lower latency compared to the unoptimized variants. An LM locally orders at least two external messages to execute a client's update. Therefore, even with a single client in the system, if the external accept

messages arrive at about the same time, the latency can be lower with aggregation. When there are many clients, the average latency of the optimized protocols is considerably less than that of the unoptimized protocols, because the optimized protocols have much higher maximum throughput. Figures 4.10 and 4.11 show the same trends on a 100 ms diameter network.

Discussion: Our optimized composable architecture achieves practical performance, with throughputs of hundreds of updates per second, even while offering the strong security guarantees of BFT/BFT. The performance of Paxos/BFT represents a factor of 4 improvement compared with the previous state of the art for wide-area Byzantine replication (i.e., unoptimized Steward). The performance of the unoptimized protocols is computationally limited and reflects the cost associated with achieving composability and flexibility. Our results show that the optimizations effectively eliminate this performance bottleneck.

4.8 Safety and Liveness Proof Sketch

Our composable architecture offers flexibility by separating the wide-area protocol, run among the logical machines, from the local-area protocol, run within the logical machine. As a direct consequence, it is possible to use a variety of replication protocols at each level of the hierarchy. Our architecture uses Paxos and BFT, both of which guarantee safety with no synchrony assumptions and liveness under certain synchrony assumptions. We can directly use the known properties of these protocols when proving safety and liveness of our hierarchical architecture.

4.8.1 Safety

Paxos and BFT do not rely on synchrony assumptions for safety. As a result, the safety of a protocol composition follows directly from the safety of these two protocols. The local state machine replication protocol used in the ordering component ensures that all replicas in a logical machine transition through the same states and invoke the signing component on identical outgoing messages. When running BFT in the logical machine, we use threshold cryptography so that malicious servers cannot generate messages that are signed by the logical machine. Thus the logical machine will not exhibit two-faced behavior assuming that it contains at most f malicious servers.

In summary, the fact that logical machine safety is guaranteed under our fault assumptions implies that a logical machine will either faithfully execute the wide-area protocol or halt. Since the safety of the wide-area protocol does not depend on the speed at which the logical machine executes this protocol, the safety of a protocol composition holds. The proof is straightforward because, when considering safety, it does not matter whether a logical machine exhibits the same behavior with respect to time as a physical machine.

4.8.2 Liveness

Although we can leverage the existing liveness properties of both BFT and Paxos, showing liveness of the customizable architecture is more complicated than showing safety. Our strategy is to first identify those timing properties that a logical machine must have in order for the protocol that runs among the logical machines to be live. Intuitively, we wish to show that the properties of our logical machines are sufficiently similar to physical machines so that, if a protocol is live when run among physical machines, it follows that it is also live when run among logical machines. The core requirement for Paxos and BFT liveness is that message delays between the physical machines do not grow faster than the timeout used to detect faulty leaders. Thus, if we can show a similar (or stronger) property for message delays between our logical machines, we can show that the composable architecture is also live.

In the following claim, we formally state the characteristics of the message delay between stable logical machines required for liveness.

Claim 4.8.1 LOGICAL MACHINE MESSAGE DELAY: *Let the system be stable with respect to time T. Then, at some time $T' > T$, for all pairs of stable connected sites comprising logical machines lm and lm', messages sent from lm to lm' are locally ordered and executed by lm' in a bounded time.*

If LOGICAL MACHINE MESSAGE DELAY (Claim 4.8.1) holds, then it follows that if BFT or Paxos are live when run on physical machines, then both protocols are live when run on our logical machines.

In our composable architecture, the maximum message delay between two logical machines is a function of the following delays:

1. Wide-area delay between sites.

2. The latency required by the BLink protocol to send a message from the originating logical machine to the correct servers in the receiving logical machine.

3. The latency required for the intra-site ordering protocol in the receiving logical machine to locally order a message from the originating logical machine.

Wide-area delay, the first of the listed contributors to message delay, also contributes to the message delay between physical machines. The other two sources of delay, due to (1) delays associated with the BLink protocol and (2) delays associated with the local ordering within a site, are specific to the composable architecture. We will argue that if the system is stable with respect to time T, then there exists a time, $T' > T$, after which sources of delay specific to the composable architecture are bounded. Since stability implies that the wide-area message delay is bounded, the total message delay will necessarily be bounded after time T', and thus LOGICAL MACHINE MESSAGE DELAY holds. This claim can be used to show that L1 - GLOBAL LIVENESS (see Section 4.2) holds.

We begin by specifying a property that we will use in the liveness proof. From our stability assumption (see Section 4.2), which is necessary for the composable system's liveness, it follows that the message delay between any two correct servers is bounded. This holds for two servers in the same site and for two servers in different sites. We formally state this property as follows:

DEFINITION 4.8.1 PHYSICAL MACHINE MESSAGE DELAY: *Let the system be stable with respect to time T. Then, at any time after T, for all pairs of stable connected servers, r and s, messages sent from r to s arrive in a bounded time.*

We prove that LOGICAL MACHINE MESSAGE DELAY holds by proving Lemma 4.8.1 which states that the time required to locally order an event will eventually be bounded.

Lemma 4.8.1 *If the system is stable with respect to time T, then in each stable site there is a time T' after which the delay required to locally order a message is bounded.*

Proof: To prove Lemma 4.8.1, we will show that for each stable site, s, there is a time after which the local leader in site s will remain in power forever. In other words, each site *settles* on a leader. It follows directly from the BFT (or Paxos) protocol that if a leader remains in power, it must order one event (if present) from each queue (including the update queue) within the timeout period. Otherwise, the stable servers in site s will elect a new leader.

We prove by contradiction that a stable site will settle on a leader. Assume that there is no leader that will remain in power forever. Then, each time a new leader is elected it must eventually be replaced. The local view change protocols (which elect and install new leaders in a round-robin pattern) used in BFT and Paxos guarantee that the stable servers in site s will rotate through local views together. Thus, during each view, all stable servers will be in the same view for $to - \Delta$, where to is the value of the local timeout and Δ is a constant that accounts for the local message delays.

Each time a new leader is elected, the local timeout increases. From Definition 4.8.1, the message delay between any two stable servers in site s is bounded. From the checkpoint and view change protocols, a leader can install a new view and order an event by exchanging a bounded amount of information with $2f$ stable servers. Since s is stable, it must contain at least $2f + 1$ stable servers. Therefore, there will be a time, T'', after which any stable server that is elected leader will have a sufficient amount of time to complete the view change and order events at a sufficient rate so that it will remain in power.

There are at most f malicious servers (out of a total of $3f + 1$) that, when elected leader, might intentionally fail to complete the view change protocol or order events in a timely manner. Thus, there may be f view changes that occur after T'' before a stable server, c, is elected leader. If view changes continue to occur, then c must fail to order events sufficiently quickly to stay in power. This implies that c is malicious because a correct server will complete the view change protocol and order events quickly enough to remain in power. Thus, we have a contradiction, because c must be both correct (and stable) and malicious. It follows that view changes cannot happen forever and that

there is a time after which the site will settle on a leader. □

We can now prove Claim 4.8.1:

Proof: We use a similar argument to the one used to prove Lemma 4.8.1 to prove Claim 4.8.1. We will show that for each pair of stable sites, r and s, there is a time after which the virtual link (i.e., forwarder-peer pair) used by the BLink protocol to send messages from r to s will remain in power forever. We say that r and s *settle* on a virtual link. It follows directly from the BLink protocol that if a link remains in power, it must pass at least one message from r to s per timeout period. BLink selects the next virtual link in the series unless at least $f + 1$ correct servers receive a threshold signed cumulative acknowledgement (from logical machine s) for one message being sent from r to s per timeout period. This ensures that, in order for a virtual link to remain in power, one message must be sent from r to s and executed by s per timeout period (unless there are no messages that need to be sent from r to s).

If we assume that the BLink protocol running between r and s does not settle on a virtual link, then the BLink protocol must continue rotating through virtual links forever. The timeout for suspecting the virtual link increases each time BLink rotates through all of the virtual links once.

Let $T'' > T$ be a time after which the following conditions hold:

1. The delays required to locally order an update at site r and at site s is bounded and sums to Δ_{lo}.

2. The delays required to propagate a message from r to all correct servers at s, generate a cumulative acknowledgement, ack, at s, and propagate ack from s to the stable servers at r are bounded and sum to Δ_p.

3. $\Delta_{lo} + \Delta_p < to_{BLink}$, where to_{BLink} is the BLink timeout.

It follows from Lemma 4.8.1 that the delays in the first condition will eventually be bounded. The delays in the second condition are bounded because of PHYSICAL MACHINE MESSAGE DELAY (Definition 4.8.1) and the fact that generating a threshold signed acknowledgement consumes

a bounded amount of time. If virtual links continue to rotate (which we assume), then the third condition holds because to_{BLink} continues to increase while Δ_{lo} and Δ_p eventually become bounded.

Thus, any virtual link selected after time T'' has a sufficient amount of time to pass a message from r to s and pass a cumulative acknowledgement back from s to the stable servers in r within a BLink timeout. Since BLink guarantees that at least one virtual link has two stable endpoints, such a link, $l_{correct}$, will eventually be selected after T'. In order for BLink to remove $l_{correct}$ from power, one of the stable servers in either site r or site s must fail to follow the BLink protocol. This is a contradiction because they are both correct servers, which implies that sites r and s eventually settle on a virtual link. It immediately follows that Claim 4.8.1 holds. □

As discussed above, Claim 4.8.1 implies that our logical machines provide message delays sufficient to guarantee liveness of the wide-area protocol. From this, L1 - GLOBAL LIVENESS (Definition 4.2.3) follows. Definition 4.2.3 requires that a stable server that receives an update propagate it to all servers in its site. This is accomplished by the CLink-Order protocol (Section 4.5). We omit a proof because the mechanism used is the same as the one used by BFT [10].

4.9 Discussion

Our composable architecture can be extended by adding new replication protocols for use on the wide area or local area besides Paxos and BFT. Several existing protocols provide desirable properties and are promising candidates for use within our system. As demonstrated in Section 4.7, wide-area protocol rounds are very costly due both to increased latency and increased message complexity. Therefore, if a system requires Byzantine fault tolerance and high performance and can tolerate reduced availability, Martin and Alvisi's two-round Byzantine fault-tolerant replication protocol [12] is well suited for use as our wide-area protocol. We believe that a composition that used this protocol would approach the performance of a composition that used Paxos on the wide area, because both are two-round protocols. The work of Yin et al. on privacy firewalls [11] can also be used effectively within a site, as part of our local-area protocol. Verrisimo's work on hybrid architectures [40,41] is another excellent candidate for use within our architecture. Special trusted hardware that provides stronger guarantees within a site can be used to strengthen the fault tolerance

of our logical machines.

We note that the consistency guarantees of our wide-area protocol can be relaxed for use with systems that do not require state machine replication semantics. For example, a composition could use a state machine replication protocol as the local-area protocol and a benign fault-tolerant anti-entropy protocol [73] on the wide area.

Relaxing the consistency guarantees of intra-site protocols is problematic and requires a fundamental change to the composable architecture. If a full state machine replication protocol is not run within the sites, we must take care to ensure that the logical machines function correctly with respect to the wide-area protocol. Our experience with Steward demonstrates that this can be very complicated. However, future research may yield logical machine abstractions that offer relaxed semantics and improved performance while simultaneously providing services that can be used to more easily construct correctly performing logical machines. Such abstractions might offer services related to both persistent storage and speculative execution, which can improve performance while still offering well-defined guarantees.

4.10 Customizable Replication for Wide-Area Networks Summary

This chapter presented a customizable, scalable replication architecture, tailored to systems that span multiple wide-area sites. Our architecture constructs logical machines (enhanced for use on wide-area networks) out of the physical machines in each site using the state machine approach, enabling free substitution of the fault tolerance method used in each site and in the wide-area replication protocol. We presented BLink, a new Byzantine fault-tolerant communication protocol that provides efficient and reliable wide-area communication between logical machines. BLink was shown to be a critical addition to the logical machine abstraction for wide-area networks, where bandwidth constraints limit performance. An experimental evaluation showed that our optimized architecture achieves a maximum wide-area Byzantine replication throughput at least four times higher than the previous state of the art.

Chapter 5
Building a Survivable Service

In this chapter, we describe how the hierarchical replication architectures presented in this dissertation can be applied to construct large-scale, survivable systems. Our architectures can be used to convert a suitable existing centralized service into a distributed, replicated service. The basic idea behind this transformation is both elegant and relatively straightforward, and it is independent of the fact that our architectures tolerate Byzantine faults. We use examples to explain the process. We also discuss problematic issues that are likely to arise in practice. Many of these issues relate directly to the Byzantine fault tolerance that our systems provide.

A non-replicated system that is a suitable candidate for conversion into a replicated system typically consists of two components: a server that provides a service and clients that submit updates and queries to the server. Keep in mind that these parts alone constitute a functional centralized (non-replicated) system. For example, a credit card company could indeed use a single server to verify charge requests. In this case, the clients correspond to the businesses that submit charges to the server.

Although a centralized system contains all of the functionality necessary to provide a credit card verification service, such a system clearly has disadvantages. The central server is a single point of failure. If this server crashes or is partitioned away from the clients, the service will be temporarily unavailable. Perhaps worse, if an adversary gains control of the server, it can control the service for its own gain. In addition, the server may, because of a hardware or software error, deviate from the protocol that implements the service with potentially catastrophic results. For example, a credit card verification system could verify charge requests for overdrawn cards.

The centralized approach also suffers from a performance bottleneck. All queries, including read-only queries that do not change the data on the server, need to be handled by a single server. In systems where read-only queries constitute a significant fraction of the load, such as name servers, multiple replicas distributed across a large area can improve throughput and reduce query latency. In a replicated system, clients can send read requests to servers that are either geographically closest or have the lowest load.

This dissertation focuses on constructing replication systems that offer scalability and intrusion tolerance. The combination of these characteristics is what makes the architectures described here challenging to design. Therefore, in the remainder of this chapter, we assume that the primary goal of the system designer is to bestow a level of survivability on the system that was previously lacking. We use a case study to (1) illustrate how a survivable service can be constructed using the composable architecture and (2) explain an array of important challenges associated with this goal.

5.1 Automated Arbitrage: Case Study

We use a simplified automated arbitrage system as a concrete example. Such a system processes prices of resources originating from different market sources and automatically executes a trade when it determines that a particular resource can be purchased in one market for lower than it can be sold in another market. Executing such a trade guarantees a profit. The model used to decide when to buy and sell is likely to be deterministic. That is, the application's current state is completely determined by its previous state and the event that causes it to transition to its current state. This is important because services that are nondeterministic require a modification that transforms them into deterministic services (see Chapter 4).

Thus, we begin with a complete model that begins in a known state and accepts price information from a variety of markets. It transitions from one state to the next based on each price update. When it identifies a trade that is likely to make money, it sends a request to buy the resource on one market, and then it immediately sells the resource on another market. Figure 5.1 shows a high-level view of a centralized arbitrage system. In this system, the computers that produce the market data can be viewed as clients and each message containing market data can be seen as an update. Note that

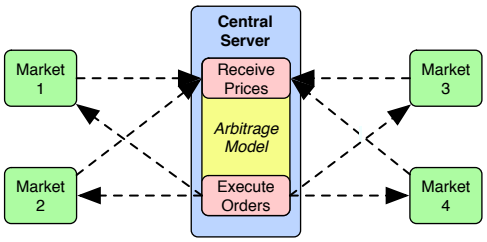

Figure 5.1: A centralized automated arbitrage system. The dotted lines represent messages sent over a wide-area network. The central server receives price data from four markets. Based on this information, it decides to execute buy and sell orders on these markets.

in our example, the clients are not as easy to identify as ATM machines interacting with a central bank, for example.

A non-replicated system would consist of a single server that receives price data for a variety of resources, perhaps stocks or commodities, and sends buy and sell orders to a variety of markets. Suppose that the system owners have already implemented best-practice security measures, and they wish to further harden the system against attackers. It is still clear that the system is vulnerable to a single compromise – that of the central server. Although we are focusing on intrusion tolerance, we note that real systems are also likely to be replicated in order to tolerate benign failures.

Both Steward and the composable architecture provide a state machine replication service. These systems establish an agreed upon order on updates that are submitted by clients and deliver the ordered updates to an application. In this case, our application consists of the arbitrage model that processes price data and executes trades when there is a high probability of financial gain. If the arbitrage model is deterministic, we can created a distributed, replicated service by taking the arbitrage model and making copies of it that will run on each replica.

If one uses the composable architecture, we need to first decide on how many local-area sites should be used, their locations, the fault tolerance (Byzantine or benign) desired in each site, and how many faults should be tolerated in each site. We also need to decide whether to use Byzantine or benign fault tolerance on the wide area. If a complete site compromise is considered likely, a Byzantine wide- area protocol should be used. However, such a protocol has costs, specifically higher update latency if BFT is chosen. We know of no simple recipe that can be used to select the appropriate system fault tolerance and scale. It is a complicated problem that necessarily forces

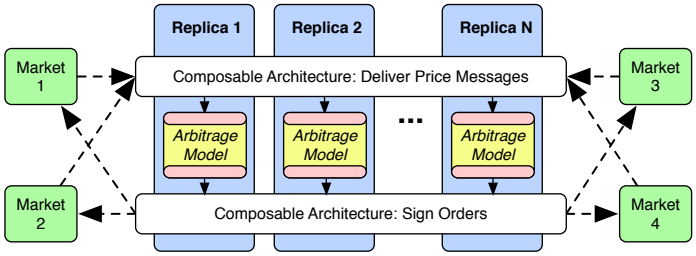

Figure 5.2: A replicated automated arbitrage system. The dotted lines represent messages sent over a wide-area network. The composable architecture receives price messages from the four market computers, establishes an order on these messages, and delivers them to the replicated arbitrage models. Each arbitrage model decides when to issue buy and sell orders and constructs the appropriate message. The composable architecture is responsible for signing these messages and sending them to the appropriate market computer.

designers to make difficult trade offs.

Next, a copy of the arbitrage model is installed in each server. The centralized implementation of the arbitrage model received price messages directly from various market computers. It also sent buy and sell messages directly to market computers. In contrast, the replicated arbitrage model processes an ordered stream of price messages that is delivered by the composable architecture. Figure 5.2 shows a high-level view of our replicated arbitrage system. When the arbitrage model decides to make a trade, it sends a buy or sell message to the composable architecture. If BFT is run on the wide area, the composable architecture is responsible for generating $F + 1$ matching threshold signatures that validate the message and propagating the resulting signed message to the appropriate market computer. A confirmation of the transaction must be received by the composable architecture and ordered before it is delivered to the replicated arbitrage models.

After deciding on the topology of the system and interfacing a copy of the arbitrage model with the composable architecture, the servers must be deployed. The servers are supplied with a site identifier and a server identifier. The tuple consisting of the two identifiers uniquely identifies each server. The servers are given the addresses of all of the other servers via their identifying tuples. Each server is supplied with a private RSA key and a RSA threshold cryptography share for its site. Each server is also supplied with the public RSA keys of all of the servers in its own site and the public RSA key for each site. This requires a secure means to distribute RSA keys.

It should be clear that the composable architecture sits between the replicated arbitrage models

and the external events that cause state transitions in these models. The composable architecture delivers messages to the arbitrage models and constructs valid buy and sell messages on behalf of a sufficient number of correct replicas. Clearly, this is necessary so that a single server does not possess the ability to unilaterally execute a trade, in which case an attacker could exploit a single computer compromise.

This isolation of the arbitrage model does pose a potential problem. The market computers are not likely to be under the control of the owner of the automated arbitrage system. Since the market computers are actually clients, it is easiest to construct the system if the market computers are aware of the replicated system and its topology. For example, if a Byzantine fault-tolerant protocol is run on the wide area, the market computer should execute a trade only if it receives matching threshold signed orders from $F + 1$ sites.

Even at this level, we already see problematic issues. While it is theoretically straightforward to construct a self-contained replicated system with a suitable service (and even to massage limited nondeterminism), problems are apt to arise when one tries to apply state machine replication architectures to real systems. The hierarchical nature of our systems exacerbates these problems. For example, if a buy order must contain a single threshold signature, then the composable system is not currently able to create a wholly satisfactory buy order. A single site can create a signature that represents the site, but there is no way currently to create a single signature that represents the system (i.e., $F + 1$ sites). Solutions to these problems may require new cryptographic protocols (e.g., a hierarchical threshold cryptography protocol).

We now examine our replicated system. Our original goal was to decrease the probability that an attacker would compromise (i.e., assume control of) our automated arbitrage system. Indeed, it is possible that our replicated system is less likely to be compromised. Instead of penetrating the defenses of a single computer, the attacker would need to compromise several computers (depending on our choice of protocols and system scale), which may be significantly less likely than compromising one computer. Note that our replicated system does not improve confidentiality, which would be very important for a real trading system. An attacker that controls a single server can monitor when our system decides to trade. A privacy firewall [11] can be used to protect the

information contained within the arbitrage model.

In order to assess whether we improved the system, we need to estimate the probability that an attacker compromises a server via physical access, a software vulnerability, or social engineering. Even the type of software vulnerability is important. Is it a vulnerability in the replication architecture code, the operating system, the arbitrage model, or another application? This discussion is intended only to emphasize the complexity in estimating the probability of system compromise. Therefore, we give examples to illustrate the scope of the problem, but we do not describe how to estimate probability of server and/or system compromise.

If there are some sites that have a relatively high chance of physical compromise, then we must run a Byzantine fault-tolerant protocol among the sites because there is certainly a high probability that if one server in a site is physically compromised, the attacker will also be able to compromise other servers in the same site, ultimately leading to a site compromise. Unfortunately, there may be correlated weaknesses in multiple sites because of similar security practices. If social engineering techniques are used to penetrate a computer in one site, the same techniques are likely to be effective in another site.

Remote compromises through software vulnerabilities pose an array of additional problems. First of all, the resilience of the replicated system to software vulnerabilities depends on diversity of the replicas. If an attacker is able to effectively use the same software exploit on all of the servers, compromising one server is akin to compromising the entire system. We therefore should use diverse operating systems and diverse software implementations of both the arbitrage model and our replication architecture. N-version programming generates diversity manually [74,75]. Recent systems that automatically create diversity [76,77] have the potential to dramatically reduce the cost of addressing correlated software vulnerabilities. We may also want to consider proactive recovery, where servers are periodically reborn in a state that, hopefully, does not contain the same set of vulnerabilities as their previous state.

We also need to consider the resources available to the attacker. If we make the attacker work harder to compromise the system, and the attacker is resource constrained, then we may make our system more survivable. If we assume that there is a particular cost associated with the compromise

of each server, then an attacker that can afford to compromise one server may clearly not be able to afford to compromise multiple servers. However, an attacker with essentially limitless resources can use multiple teams of hackers working in parallel to compromise our system. In such a case, our replicated system is less effective. It is also important that the owner of the arbitrage system has sufficient resources to create the requisite software diversity and, potentially, security policy variability in different administrative domains. Costs associated with such efforts may be exorbitant.

We believe that a well thought out replicated system offers a significant increase in survivability. However, the key to this statement is *well thought out*. Naively converting a centralized system into a Byzantine fault-tolerant service may in fact make it more likely that it either will be compromised or simply fail because of a software error. Judicious use of Byzantine fault tolerance, on the other hand, may be very effective. It remains to be seen if the types of systems described in this dissertation will be used in practice. This dissertation provides a strong argument that the performance of such systems is satisfactory for many services. However, the costs and benefits associated with an actual deployment remain difficult to judge.

We conclude this example by noting that a replicated system has advantages beyond improved survivability. As we noted early, replication can improve throughput and latency. However, replicated systems offer other potential benefits by making it possible to exploit geographic locality.

In our automated arbitrage system, once a potentially profitable trade has been identified, it is very important to initiate the trade as quickly as possible. If our system has replicas near different market computers, then the closest computer may be able to optimistically initiate a trade based on an update that has not yet been assigned an order, but is likely to trigger the trade when it is ordered. A replica that is close to both the originator of the update that triggers the trade and the market computer that will process the trade is in a prime position to optimistically initiate a trade.

Our hierarchical systems currently do not deliver an update to the replicated service unless its order has been fixed. However, with relatively minor modifications, our systems can be made to deliver updates earlier. An architecture modified in this manner can provide a rich set of semantics that can be used by a system designer to balance performance and fault tolerance. It is possible that, in practice, survivable replication architectures will provide weaker services in addition to state

machine replication. Such systems have the potential to improve both resilience and performance.

5.2 Identifying and Responding to Faulty Servers

Our architectures are designed to provide safety and liveness. With only one exception, this can be achieved without identifying faulty servers.[1] However, if a server is faulty, it is certainly beneficial to identify and exclude it. If a server engages in an attack and is identified as faulty by all correct servers, then the correct servers can prevent that server from engaging in the same attack again by blacklisting it. Correct servers can ignore messages from blacklisted servers and prevent blacklisted servers from becoming leaders or representatives.

Faulty servers can engage in two types of damaging malicious behavior. First, they can send messages that conflict. For example, a malicious representative at the leader site may send two Pre-Prepares that bind the same sequence number to two different updates. Since our systems use signatures, this type of malicious behavior is easy to identify. Two correctly signed messages from the same server that conflict constitute proof that the server is faulty. The second type of behavior is more difficult to handle; a faulty server can delay sending its messages. Since our architectures do not rely on synchrony assumptions to provide safety, servers that are slow cannot be excluded because they may be correct. In summary, we can remove a slow server from power (which is precisely what our view change protocols do), but we cannot blacklist it.

It is important to note that even when more than f but fewer than $2f + 1$ servers within a site are compromised, the malicious servers may not be able to make the site behave in a Byzantine manner. If the malicious servers are identified and blacklisted by all correct servers, then they will not be able to collect the necessary messages from the correct servers to generate a valid threshold signed wide-area protocol message (when (2f+1,3f+1) threshold signatures are used). Therefore, a practical system should include a rapid blacklisting mechanism. Such a mechanism may require signatures. At the very least, signatures will make it easier to design an effective blacklisting protocol. Considering also that (1) optimizations can be used to distribute the cost of signatures across multiple updates and (2) the computational cost associated with signatures is no longer exorbitant,

[1] The Steward THRESHOLD-SIGN protocol includes a blacklisting mechanism which is necessary for liveness (see Figure 3.10).

we believe that they are the best option for many practical Byzantine replication systems.

5.3 Summary

This chapter is intended to provide the reader with (1) a better understanding of how our architectures might be used to construct a replicated service and (2) an appreciation of the subtle issues that a designer is likely to encounter in practice. Our architectures are research systems. They have served their intended purpose by demonstrating that Byzantine fault-tolerant replication can achieve practical performance on wide-area networks. We leave the task of uncovering an effective methodology for reasoning about the probability of system compromise to future research.

Chapter 6
Conclusions

This dissertation presents the first two Byzantine fault-tolerant replication architectures that scale to wide-area networks. Our systems achieve performance comparable to common benign fault-tolerant protocols, which demonstrates for the first time that Byzantine fault- tolerant state machine replication is practical for wide-area networks. The architectures also contain several important technological contributions that we summarize below.

Our first system, Steward, uses hierarchy to achieve performance that surpasses the previous state of the art by an order of magnitude. Steward comprises a series of Byzantine fault-tolerant protocols that allow the servers within each local-area site to act as a single participant in a benign fault-tolerant, wide-area protocol. This dissertation contains a detailed proof of system safety and liveness. Notable contributions of Steward include a reduction in wide-area message complexity, improved availability, reduced update latency, and the ability to answer read-only queries locally.

Our second system, the composable architecture, improves upon Steward by offering customizability of the fault tolerance approach used within and among the sites. This customizability is derived from cleanly separating the intra and inter-site protocols. Clean separation allows us to leverage existing properties of the flat replication protocols used in the composition to reason about safety and liveness. Notable contributions of the composable architecture include a new Byzantine link protocol that allows efficient communication between logical machines and the use of optimizations that improve performance.

Bibliography

[1] S. Schwankert, "U.s. congressmen accuse china of hacking their computers," *New York Times, IDG News Service*, June 2008.

[2] W. Jackson, "Government, health care web sites attacked: New wave of attacks compromise government and health care web sites," *Government Computer News*, July 2008.

[3] "Ciphertrust's zombie stats," http://www.ciphertrust.com/resources/statistics/zombie.php.

[4] D. Sevastopulo, "Chinese hacked into pentagon," *Financial Times, London*, September 2007.

[5] P. E. Veríssimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*, R. Lemos, C. Gacek, and A. Romanovsky, Eds., 2003, vol. 2677.

[6] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens, "An on-demand secure routing protocol resilient to byzantine failures," in *ACM Workshop on Wireless Security (WiSe)*, Atlanta, Georgia, September 2002. [Online]. Available: citeseer.ist.psu.edu/article/awerbuch02demand.html

[7] R. Curtmola and C. Nita-Rotaru, "Bsmr: Byzantine-resilient secure multicast routing in multihop wireless networks," *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON '07. 4th Annual IEEE Communications Society Conference on*, pp. 263–272, June 2007.

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

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "The SecureRing protocols for securing group communication," in *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, vol. 3, Kona, Hawaii, January 1998, pp. 317–326.

[10] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the 1999 Symposium on 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.

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

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

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

[14] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[15] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

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

[17] ——, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *To appear in IEEE Transactions on Dependable and Secure Computing*.

[18] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.

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

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

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

[22] 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," in *Supplement of the 2002 International Conference on Dependable Systems and Networks*, June 2002.

[23] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders, "Quantifying the cost of providing intrusion tolerance in group communication systems," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, Bethesda, MD, USA, June 2002, pp. 229–238.

[24] P. Pandey, "Reliable delivery and ordering mechanisms for an intrusion-tolerant group communication system," masters Thesis, University of Illinois at Urbana-Champaign, 2001.

[25] H. V. Ramasamy, "A group membership protocol for an intrusion-tolerant group communication system," masters Thesis, University of Illinois at Urbana-Champaign. 2002.

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

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

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

[29] L. Lamport, "Paxos made simple," *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, vol. 32, pp. 18–25, 2001.

[30] A. Doudou, R. Guerraoui, and B. Garbinato, "Abstractions for devising byzantine-resilient state machine replication," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS '00)*. Nurnberg, Germany: IEEE Computer Society, 2000, pp. 144–153.

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

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

[33] R. Kotla, L. Alvisi, M. Dahlin, A. C. t, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Prin ciples (SOSP '07)*, 2007, pp. 45–58.

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

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

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

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

[38] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 59–74.

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

[40] M. Correia, L. C. Lung, N. F. Neves, and P. Veríssimo, "Efficient byzantine-resilient reliable multicast on a hybrid failure model," in *Proceedings of the 21st Symposium on Reliable Distributed Systems (SRDS '02)*, Suita, Japan, Oct. 2002, pp. 2–11.

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

[42] "Survivable spread: Algorithms and assurance argument," The Boeing Company, Tech. Rep. Technical Information Report Number D950-10757-1, July 2003.

[43] 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. [Online]. Available: citeseer.ist.psu.edu/schneider90implementing.html

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

[45] D. Powell, D. Seaton, G. Bonn, P. Veríssimo, and F. Waeselynck, "The Delta-4 approach to dependability in open distributed computing systems," in *Proceedings of the 18th IEEE International Symposium on Fault-Tolerant Computing (FTCS)*, Jun. 1988, pp. 246–251.

[46] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. A. Speirs, and S. Tao, "Implementing fail-silent nodes for distributed systems," *IEEE Transactions on Computers*, vol. 45, no. 11, pp. 1226–1238, 1996. [Online]. Available: citeseer.ist.psu.edu/brasileiro96implementing.html

[47] K. P. Kihlstrom and P. Narasimhan, "The Starfish system: Providing intrusion detection and intrusion tolerance for middleware systems." in *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '03)*. Guadalajara, Mexico: IEEE Computer Society, 2003, pp. 191–199.

[48] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for web-service applications," in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS '05)*. Orlando, FL, USA: IEEE Computer Society, 2005, pp. 131–142.

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

[50] P. Felber, R. Guerraoui, and A. Schiper, "Replication of CORBA objects," in *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*. London, UK: Springer-Verlag, 1999, pp. 254–276.

[51] R. Friedman and E. Hadad, "FTS: A high-performance CORBA fault-tolerance service," in *Proceedings of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS '02)*. San Diego, CA, USA: IEEE Computer Society, 2002, pp. 61–68.

[52] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith, "Providing support for survivable CORBA applications with the Immune system," in *Proceedings of the 19th IEEE*

International Conference on Distributed Computing Systems (ICDCS '99), Austin, TX, USA, 1999, pp. 507–516.

[53] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, 2002.

[54] Y. G. Desmedt and Y. Frankel, "Threshold cryptosystems," in *CRYPTO '89: Proceedings on Advances in cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1989, pp. 307–315.

[55] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[56] V. Shoup, "Practical threshold signatures," *EUROCRYPT 2000, Lecture Notes in Computer Science*, vol. 1807, pp. 207–220, 2000. [Online]. Available: citeseer.ist.psu.edu/shoup99practical.html

[57] P. Feldman, "A Practical Scheme for Non-Interactive Verifiable Secret Sharing," in *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society. Los Angeles, CA, USA: IEEE, October 1987, pp. 427–437.

[58] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Robust threshold dss signatures," *Information and Computation*, vol. 164, no. 1, pp. 54–84, 2001.

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

[60] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant systems to wide area networks," Tech. Rep. CNDS-2005-3, Johns Hopkins University and CSD TR 05-029, Purdue University, www.dsn.jhu.edu, December 2005.

[61] 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.

[62] 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.

[63] 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.

[64] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Byzantine replication under attack," in *Proceedings of the 2008 International Conference on Dependable Systems and Networks (DSN '08)*. Anchorage, AK, USA: IEEE Computer Society, June 2008, p. xxxx.

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

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

[67] 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.

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

[69] R. Rodrigues, M. Castro, and B. Liskov, "BASE: using abstraction to improve fault tolerance," in *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP '01)*. Banff, Alberta, Canada: ACM Press, 2001, pp. 15–28.

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

[71] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," Tech. Rep. CNDS-2006-2, Johns Hopkins University, www.dsn.jhu.edu, November 2006.

[72] Y. Amir and C. Danilov, "Reliable communication in overlay networks," in *Proceedings of the IEEE International Conference on Dependable Systems and Networks, (DSN '03)*, June 2003, pp. 511–520. [Online]. Available: citeseer.ist.psu.edu/article/amir03reliable.html

[73] R. A. Golding and K. Taylor, "Group membership in the epidemic style," University of California, Santa Cruz, CA, Tech. Rep. UCSC-CRL-92-13, Mar. 1992.

[74] A. Avizeinis, "The n-version approach to fault-tolerant software," *IEEE Transactions of Software Engineering*, vol. SE-11, no. 12, pp. 1491–1501, December 1985.

[75] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,' Highlights from Twenty-Five Years'*., pp. 113–, June 1995.

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

[77] 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," in *USENIX-SS '06: Proceedings of the 15th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2006, pp. 8–8.

Vita

John Lane graduated from Cornell University in 1992 with a BA in Biology. He worked in neuroscience research for several years at Johns Hopkins, first doing experiments and then transitioning to designing and constructing computerized data-collection systems. In 1998 he became a programmer at the Krieger Mind/Brain Institute. In 2003, he became a full-time graduate student in the Department of Computer Science at Johns Hopkins University. He was a member of the Distributed Systems and Networks Lab, and his research focused on large-scale, intrusion-tolerant, distributed systems. John Lane joined Live Time Net as a Senior Research Scientist in 2008.