# Practical Mobile Proactive Secret Sharing

by

David Dryjanski

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2008

Author ......................
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by ........................................................ [illegible]
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by ........................................ [illegible]
Terry P. Orlando
Chairman, Department Committee on Graduate Students

# Practical Mobile Proactive Secret Sharing

by

David Dryjanski

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

## Abstract

Secret sharing schemes are needed to store and protect secrets in large scale distributed systems. These schemes protect a secret by dividing the it into shares and distributing the shares to multiple shareholders. This way the compromise of a single shareholder does not reveal the secret. Many new secret sharing schemes, such as Proactive Secret Sharing, have been developed to combat the increasing threat from malicious nodes and keep systems secure. However, most of these schemes can be compromised over time, since share transfer and redistribution are static: the set of shareholders is fixed. Mobile Proactive Secret Sharing (MPSS) is a new protocol with dynamic redistribution that can adapt to Byzantine faults and remain secure for the duration of long-lived systems. This thesis describes the simulation, testing, and evaluation of the MPSS protocol to better understand the performance trade-offs and practicality of secret sharing protocols operating in Byzantine faulty environments. The thesis evaluates the original MPSS scheme and the MPSS scheme with verifiable accusations in a distributed setting, finds that both schemes are practical, and explores the performance trade-offs between the two schemes.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

## Acknowledgments

I would like to thank my advisor, Barbara Liskov, for giving me the opportunity and challenge of working on this thesis. I owe David Schultz much of my gratitude for developing the MPSS protocol, and always being helpful and providing effective guidance for all of my research needs. I would also like to thank Ben Vandiver for specifically aiding me in creating shell scripts for a distributed system, and James Cowling for his expertise in Emulab. I would further like to thank the rest of the members of the Programming Methodology Group (PMG) and other colleagues at the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL): Daniel Myers, Daniel Ports, Dorothy Curtis, Evan Jones, Liuba Shrira, Raluca Ada Popa, Winnie Cheng, and Yoyo Zhou for providing a great research environment. Finally, I would like to thank my family and friends for their love and support throughout my entire educational career.

# Contents

# List of Figures

# Chapter 1

# Introduction

Secret sharing schemes are needed to store and protect secrets in large scale distributed systems. These schemes protect a secret by dividing the it into shares and distributing the shares to multiple shareholders. This way the compromise of a single shareholder does not reveal the secret. Many new secret sharing schemes, such as Proactive Secret Sharing, have been developed to combat the increasing threat from malicious nodes and keep systems secure. However, most of these schemes can be compromised over time, since share transfer and redistribution are static: the set of shareholders is fixed. Mobile Proactive Secret Sharing (MPSS) is a new protocol with dynamic redistribution that can adapt to Byzantine faults and remain secure for the duration of long-lived systems. This thesis describes the simulation, testing, and evaluation of the MPSS protocol to better understand the performance trade-offs and practicality of secret sharing protocols operating in Byzantine faulty environments. The thesis evaluates the original MPSS scheme and the MPSS scheme with verifiable accusations in a distributed setting, finds that both schemes are practical, and explores the performance trade-offs between the two schemes.

## 1.1 Motivation for MPSS

This section identifies a motivating scenario for MPSS. The scenario we consider is a company trying to protect a digital secret, namely a private key (e.g. the private key that a certificate authority like Verisign, Inc. uses to sign its certificates). Companies use their private keys to authentically distribute their software, to digitally sign their documents, and to act as the basis for the company's digital identity. Therefore, companies must protect their private keys or else they face disastrous consequences. Furthermore, these companies have disparate data centers around the world which all need access to private keys to perform company tasks. What is the best solution for this company?

Using standard public key cryptography, where you generate a public/private key pair to each key server is one solution. However, this solution presents a serious key management problem: each key server is a single point of failure. Consider the situation when a data center's key server is compromised; now the attacker has full access to all of the company's operations. To recover from such an attack, the company would have to regenerate its public/private key pair and reissue certificates using the new key. This is a time intensive process that would have to occur on a regular basis in order to prevent attacks.

Fortunately, these problems are solved by using MPSS. The secret sharing aspect of MPSS protects a secret by distributing shares of the secret (i.e., the private key of the public/private pair) to several nodes in the system, such that an attacker would have to compromise more than a threshold, $f$, of nodes to gain access to the secret. Instead of distributing the secret to the nodes, only cryptographic shares of the secret are distributed. MPSS is proactive, performing share regeneration periodically, in system-wide intervals called epochs, so that long-lived systems can remain secure indefinitely. The secret is never revealed, provided that no more than $f$ shareholders are corrupted in a given epoch. Finally, MPSS can change the threshold for obtaining the secret and change the shareholder membership of the secret. MPSS's mobility enables it to adapt to security vulnerabilities by modifying shareholder membership and increasing the system's threshold. MPSS provides a more resilient, secure, and sustainable system for protecting secrets than previous solutions.

## 1.2 Contributions

The technique and protocol for MPSS were defined by Schultz and Liskov [Sch07]; the contribution in this thesis is to implement an MPSS system, explore optimizations, and measure performance in order to ascertain the true practicality of MPSS. The thesis makes the following contributions:

1. MPSS Implementation – Implement a system that simulates the protocol and design specifications in [Sch07].

2. MPSS Evaluation – Run experiments on the system to compare the behavior of the system to the original intentions of MPSS.

3. MPSS Scalability – Run performance tests on the system to analyze the scalability and practicality of MPSS.

4. MPSS Trade-offs – Explore the performance trade-offs of the original MPSS scheme and the scheme that utilizes verifiable accusations.

We found that our implemented MPSS system matched expected performance of the original MPSS protocol and was indeed scalable.

The thesis is organized as follows. Chapter 2 describes related work and the assumptions of our system. Chapter 3 details the MPSS protocol. The implementation and evaluation results of the system are discussed in Chapter 4. We conclude in Chapter 5.

# Chapter 2

# Background and Assumptions

The first part of this chapter discusses the relevant theory and schemes that eventually led to MPSS. The last part of the chapter discusses the assumptions we make about the environment in which MPSS operates.

## 2.1 Background

Secret sharing, originally developed by Adi Shamir in 1975 [Sha79], is an algorithm that provides information theoretical security to protect a given secret. The algorithm creates a set of $N$ shares for a secret that, alone, resemble random bits of information. A threshold, $f$, is defined such that when more than $f$ of the $N$ shares are combined, the algorithm can recover the secret, but with up to $f$ shares, no information is exposed. The algorithm works by generating a random degree-$f$ polynomial, where each share is a point on the polynomial. The secret is the value of the polynomial at zero. When enough shares are combined, the points are interpolated to compute the polynomial, which is then evaluated at zero to determine the secret. Even $f$ shares reveal nothing about the secret since there is a degree of freedom for the last, defining point on the polynomial.

In the original proposal for secret sharing, there is one trusted node, known as the dealer, which runs the secret sharing algorithm to distribute and recombine the shares of the secret; the nodes that receive shares are known as shareholders. Verifiable

Secret Sharing (VSS), as described by Feldman et al. [Fel87], extends secret sharing by allowing a semi-trusted dealer. A semi-trusted dealer cannot reveal the actual secret, but it may trick other nodes by giving false shares. VSS adds extra commitment information to each share, so that receiving nodes can verify the validity of their share.

Although secret sharing adds security to a system by forcing attackers to compromise more than $f$ machines, over long periods of time, an attacker can accumulate enough shares to eventually discover the secret. Proactive Secret Sharing (PSS) was first described by Ostrovsky and Yung in 1991 [OY91], as an adaptation of secret sharing to be able to handle compromises in a system over time. Their model operates under the assumption that nodes in their system can be infected by an adversary at a certain rate, but nodes can also be restored to their correct state at an equal rate. Under the original secret sharing scheme, over time, enough infected nodes will learn all of the shares, allowing the adversary to recover the secret. Ostrovsky and Yung's scheme introduces a refresh protocol with a system wide epoch that recomputes the shares of the secret at the start of each epoch. Therefore, as long as the duration for each epoch is less than the time for an adversary's rate of infection to infect more than t nodes in the system, the secret can be preserved.

Ostrovsky and Yung proved that it was theoretically possible to build a PSS system. However, Ostrovsky and Yung's PSS scheme is inefficient and impractical due to its dependence on secure, multi-party computation, which is expensive, and its assumption of a synchronous network. Herzberg et al. [HJKY95] improved upon the scheme by making an efficient PSS protocol for synchronous networks. Although impractical for real systems, these two works proved that it was possible to define a PSS scheme, which has led to much of the follow-on work for secret sharing schemes.

Mobile Proactive Secret Sharing enhances PSS by preserving secrecy in asynchronous networks even if nodes cannot be fully restored to a correct state. The MPSS scheme gives the system the ability to change the number of shareholders, redefine the set of shareholders to the secret, and adjust the threshold of the secret. Now if the adversary suddenly increases the fraction of infected nodes, the system can increase the threshold (i.e. number of shares of the secret) and increase the number of shareholders to prevent the adversary from accessing the secret. MPSS eliminates the assumption made by PSS that compromised nodes can be recovered back to their correct states. The assumption is faulty due to the fact that it may be impossible to determine if the node can ever be trusted again. Instead, MPSS makes a stronger security guarantee by providing security even if nodes have been compromised. MPSS uses the Byzantine Fault Tolerance (BFT) algorithm [CL02] [CL99], in order to reach system-wide agreement amongst non-faulty nodes. BFT guarantees the liveness and safety of a system with $3f+1$ nodes as long as there are no more than $f$ faulty nodes. Thus, MPSS can guarantee correctness as long as the number of faulty nodes is no larger than the threshold and by allowing the system to change the shareholders of the secret.

A variant of MPSS, that is also explored in the thesis, is the scheme that uses Verifiable Accusations. MPSS with Verifiable Accusations (MPSSVA) allows nodes to verifiably accuse malicious nodes when detecting invalid data. A node receiving an accusation can then verify cryptographically whether or not the accusation is justified. The ability to verify optimizes certain parts of the protocol, when compared to MPSS. The trade-off when using MPSSVA is the cryptographic overhead for producing and analyzing Verifiable Accusations, and the extra storage space needed to append to protocol messages.

## 2.2 Assumptions

### 2.2.1 Network Assumptions

MPSS operates under realistic network conditions, in which adversaries may exist. This translates to the familiar asynchronous network security model, where messages are sent peer to peer and can be delayed, reordered, or lost. We assume an adversary can have complete control over the network and can decide to reorder the messages, modify messages, and even create new messages. Under these conditions our protocol will be correct, but in order to terminate and ensure liveness, our protocol requires the network to have the property of strong eventual delivery. Strong eventual delivery ensures that the maximum delay in message delivery for messages repeatedly sent from uncorrupted nodes is bounded (with some unknown bound), and while that bound can change over time, it does not increase exponentially indefinitely.

### 2.2.2 Cryptographic Assumptions

We use cryptography to both hide secret information and also to ensure that messages come from specific nodes. We require the following cryptographic assumptions: We require collision-resistant hash functions such that it is infeasible for an adversary to find a $Y$ where $hash(Y) = hash(X)$. SHA-256 is an example of a hash function that is widely believed to be collision resistant and this is what our implementation uses. Public Key Encryption and Decryption algorithms, such as RSA, make it infeasible for an adversary to decrypt information without the proper private key pairs. Verifiable Secret Sharing is computationally secure under the Discrete Logarithm Assumption. MPSS uses forward-secure encryption and forward-secure signing, which is computationally secure under the Bilinear Diffe-Hellman Assumption [BF01].

### 2.2.3 Adversarial Assumptions

We assume there exists a powerful adversary that is computationally bounded in polynomial time, and can monitor all network traffic and corrupt nodes at a reasonable rate. Once a node is corrupted, that node is corrupted forever and the adversary automatically learns all of the information from that node. A corrupted node is completely controlled by the adversary and can deviate from the protocol, send fake messages, or even act like an honest node.

### 2.2.4 Epoch Assumptions

MPSS progresses in a series of epochs, such that a share during an epoch $e$ is only valid for that epoch. At the end of each epoch all share information is completely erased before moving to the next epoch. We assume that an adversary can corrupt no more than $f$ nodes in a given epoch.

# Chapter 3

# MPSS Protocol

This chapter describes the original MPSS scheme and the MPSS scheme with Verifiable Accusations [Sch07]. For full descriptions of the protocols, we refer the reader to [Sch07], as this chapter describes the protocols only in enough detail for the purposes of this thesis. In the protocol, all messages are signed by the sender and encrypted for the recipient using a forward-secure signature and forward-secure encryption scheme respectively. We define a Participant to be a node in the protocol that owns a share of the polynomial for the current epoch. We define the Primary to be the Participant that coordinates the protocol. In the text, we write each message type of the protocol in **bold** font with a **MSG** prefix.

## 3.1 Original MPSS Scheme

The MPSS protocol is composed of three main stages: proposal selection, agreement, and share transfer. Proposal selection is the process by which all of the current shareholders generate and select a new secret sharing polynomial for the next epoch. Once a selection has been made, the agreement stage of the protocol commences, and completes when a majority of non-faulty shareholders reach agreement on the selected polynomial. The final stage of the protocol is the share transfer of the polynomial from the group of existing shareholders to the new group of shareholders for the next epoch. This section continues with a description of these stages in more detail.

### 3.1.1 Proposal Selection

During proposal selection, each node in the system proposes a group of $3f + 2$ polynomials of degree $f$, where $f$ is the current threshold, by broadcasting a **MSGPROPOSAL** containing its proposals. These messages contain Feldman commitments to the proposed polynomials; their details can be found in [Sch07]. Upon receiving $2f + 1$ valid **MSGPROPOSAL**'s, the Primary combines these proposals into a proposal set, and broadcasts the **MSGPROPOSALSET** to all of the participants. Each node inspects the **MSGPROPOSALSET** and sends a **MSGPROPOSALRESPONSE** to the Primary with a list of the proposals it finds to be invalid (i.e. corrupt). The Primary runs the online Proposal Selection Algorithm (PSA), shown in Figure 3-1, as each **MSGPROPOSALRESPONSE** arrives until it satisfies the PSA stop condition.

The PSA deterministically selects a subset of the original proposal set that is guaranteed to contain at least one honest proposal. It accomplishes this by removing any proposal from a node that has been accused and also a proposal from an accuser node (since it cannot tell which node is bad). Each **MSGPROPOSALRESPONSE** can only remove at most two proposals (the accuser and the accusee). Since at most $f$ nodes can be malicious, they can remove at most $2f$ proposals, and thus, there will be at least one honest proposal left in the set.

At this point, the Primary has selected a final proposal set that it can use to create a new polynomial to share the secret. The polynomial is created by linearly combining the selected proposed polynomials; and has the property that each node sees only a part of this polynomial and learns nothing about the rest of the polynomial (more information can be found in [Sch07]).

### 3.1.2 Agreement

Once proposals have been selected, the Primary initiates a BFT [CL02] agreement for its selected proposals. The value to be agreed upon by the participants is the resultant set of selected proposals. Thus, the Primary sends its initial proposal set and the

Figure 3-1: Proposal Selection Algorithm (PSA) Pseudocode

```
1.  $d \leftarrow 0$,   $\textbf{satisfied} \leftarrow \emptyset$,   $\textbf{rejected} \leftarrow \emptyset$
2.  $\textbf{props} \leftarrow$ set of all proposals in $\textbf{MsgProposalSet}$
3.  $\textbf{foreach MsgProposalResponse}$ $R$ from distinct node $i$
4.    $\textbf{if}$ $i \in \textbf{rejected}$
5.      $\textbf{continue}$
6.    $\textbf{if}$ $\exists j \in R.\textbf{BadSet}$ such that $j \in \textbf{props}$
7.      $\textbf{props} \leftarrow \textbf{props} - \{i, j\}$,   $\textbf{rejected} \leftarrow \textbf{rejected} \cup \{i, j\}$
8.      $\textbf{satisfied} \leftarrow \textbf{satisfied} - \{j\}$
9.      $d \leftarrow d + 1$
10.   $\textbf{else}$
11.     $\textbf{satisfied} \leftarrow \textbf{satisfied} \cup \{i\}$
12.   $\textbf{if}$ $|\textbf{satisfied}| = 2f + 1 - d$
13.     $\textbf{stop}$
```

in-order set of **MSGPROPOSALRESPONSE**'s it has received to participants. A node votes by running the PSA on these inputs and determining if the output equates to the Primary's selection. The BFT protocol operates with the standard **MSGBFTPREPREPARE**, **MSGBFTPREPARE**, and **MSGBFTCOMMIT** messages. A node can participate in the BFT phase even if it did not agree with the selected proposals. If agreement is reached the protocol moves into the share transfer stage. Otherwise, the Primary is faulty and a view change occurs forcing a restart of the protocol with a new Primary.

### 3.1.3 Share Transfer

Upon reaching agreement, the Primary and all Participants send **MSGNEWPOLY** messages based on the new polynomial to all Participants of the next epoch. When a Participant from the current epoch receives at least $f + 1$ **MSGNEWPOLY** acknowledgments, it deletes all of its current secret sharing information and progresses into the new epoch, since at least one honest node has received the new polynomial. A new Participant computes its own share when it has received $f + 1$ valid **MSGNEWPOLY** from old Participants; it holds on to these messages for the duration of the next epoch in case some other new Participant never received the message.

### 3.1.4 Protocol Objects and Messages

Now we describe the different objects needed to run the protocol. There are underlying secret sharing information objects such as the proposal polynomial and commitments, and the protocol message objects. Here we describe the functionality and role of these objects in the system.

#### Protocol Objects

**Proposal** - A sender node creates a proposal object for every Participant in the protocol. For each Participant's proposal there is a set of $3f + 2$ points, the Participant's point on each of the $3f+2$ polynomials the sender node generated. Each proposal is encrypted for its intended Participant. The object is $O(f)$ in size.

**Commitment** - A commitment object provides the information necessary for a node to verify that the proposal was generated properly and unmaliciously. Commitments are essential for the protocol to generate secure shares and a trusted polynomial. There is one commitment object for each of the $3f+2$ polynomials generated by the sender. Each commitment is composed of $f + 1$ points per proposal point for a total of $(3f + 2)(f + 1) = 3f^2 + 5f + 3$ points.

**AgreementValue** - This object is used during the BFT agreement phase of the protocol as the value to be agreed upon by all of the participants. The AgreementValue consists of a hash of the proposal set and the ordered set of responses that the Primary used as input to the PSA to select the final set of proposals. A shareholder can then rerun the PSA with these inputs to verify the Primary reached the same set of proposals. The size of an AgreementValue is $O(f^2)$.

**Protocol Messages**

**MSGPROPOSAL** - This message contains the proposal for the share generation polynomial for every node in the system along with the commitments to the polynomials. The size of this message is $O(f^2)$.

**MSGPROPOSALSET** - This message sent by the Primary contains the hashes of $2f + 1$ **MSGPROPOSAL**'s the Primary received along with the identifiers of their senders. The size of a proposal set message is $O(f)$.

**MSGPROPOSALRESPONSE** - This message is the Participant response to a **MSGPROPOSALSET**, where the Participant identifies the proposals it believes are invalid along with a hash of the original **MSGPROPOSALSET**. A node votes on what the Primary sent, so the size of the message is $O(f)$.

**MSGBFTPREPREPARE** - The Primary sends this message when it has selected a set of proposals for the rest of the Participants to agree on. The message consists of the AgreementValue that other nodes must vote on. As noted earlier an AgreementValue has a size of $O(f^2)$, so the size of a **MSGBFTPREPREPARE** is also $O(f^2)$.

**MSGBFTPREPARE** - This message is a vote that contains a hash of the AgreementValue. Size is $O(1)$.

**MSGBFTCOMMIT** - This message commits to the hash of the AgreementValue in the final round of BFT agreement. Size is $O(1)$.

**MSGNEWPOLY** - Much like a **MSGPROPOSAL**, this message contains points on the new polynomial that will be used to generate the new shares for each member in the new group. The polynomial is a linear combination of all the polynomials that were agreed to in the AgreementValue object and commitments, such that recipients of a **MSGNEWPOLY** can only generate their own shares and nothing else. The message size is equivalent to a **MSGPROPOSAL**, $O(f^2)$.

### 3.1.5 State Machine

We can describe the protocol as a finite state machine for a better understanding of MPSS (a diagram is shown in Figure 3-2). We only cover the transitions for the normal case of the protocol and we leave out transitions that would occur from message reordering and faulty nodes.

The Participant node maintains a state for each stage of the protocol in which it participates.

**WAIT** - This is the beginning state, where the Participant waits until the Primary sends a **MSGSTARTPROTOCOL**. The **MSGSTARTPROTOCOL** is a message artifact from our implementation that synchronizes the initiation of the protocol, but the protocol does not necessarily have to start in this manner. The Participant then broadcasts its **MSGPROPOSAL** to initiate the MPSS protocol and transitions to the PROPOSAL_SENT state once it has sent all of the messages.

**PROPOSAL_SENT** - In this state the node is idle until it receives a **MSGPROPOSALSET** from the Primary, in which case it validates the message, sends a **MSGPROPOSALRESPONSE** to the Primary, and moves into the PROPOSAL_RESPONSE_SENT state.

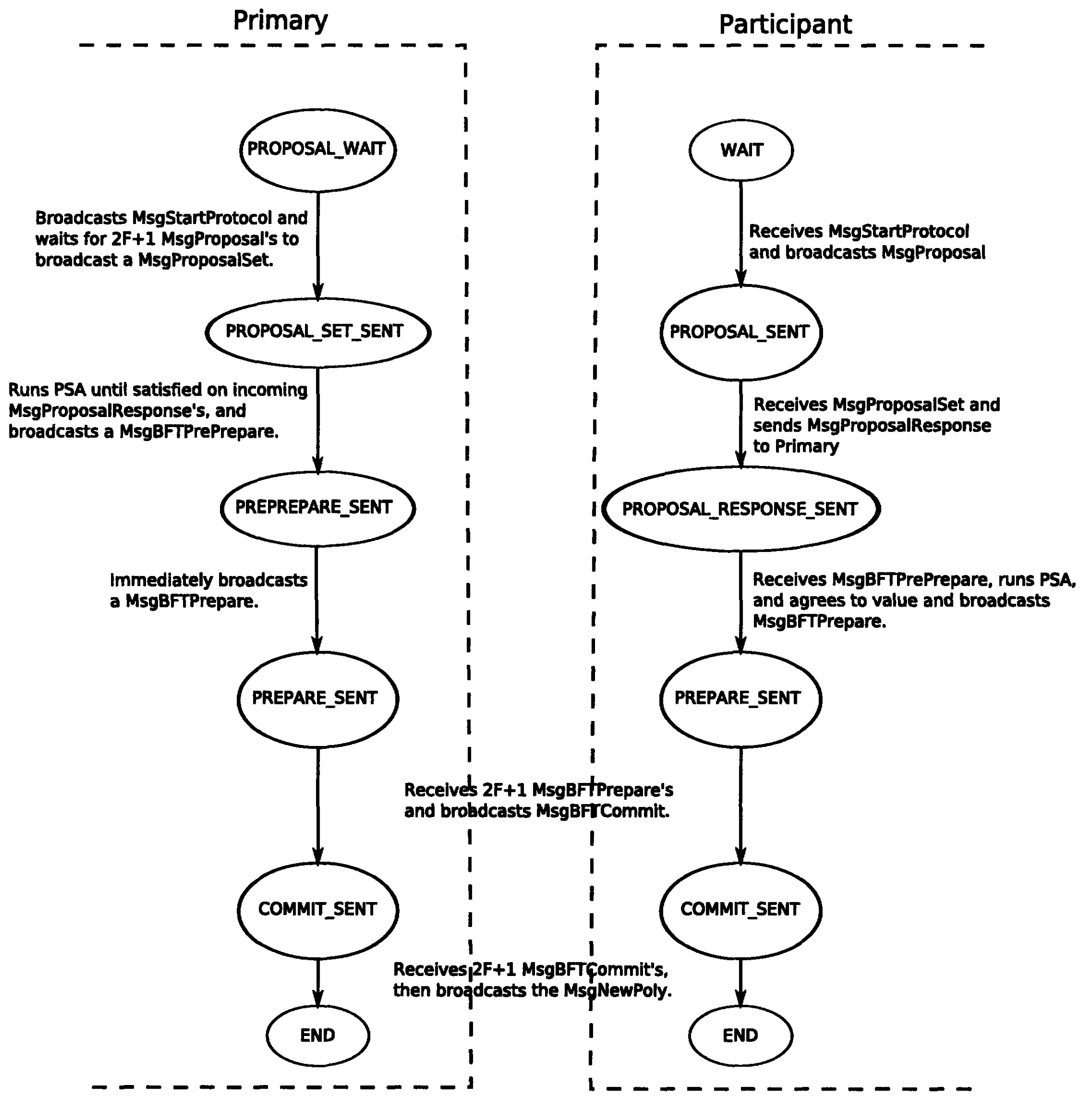
Primary
Participant
PROPOSAL_WAIT
Broadcasts MsgStartProtocol and waits for 2F+1 MsgProposal's to broadcast a MsgProposalSet.
PROPOSAL_SET_SENT
Runs PSA until satisfied on incoming MsgProposalResponse's, and broadcasts a MsgBFTPrePrepare.
PREPREPARE_SENT
Immediately broadcasts a MsgBFTPrepare.
PREPARE_SENT
COMMIT_SENT
END
WAIT
Receives MsgStartProtocol and broadcasts MsgProposal
PROPOSAL_SENT
Receives MsgProposalSet and sends MsgProposalResponse to Primary
PROPOSAL_RESPONSE_SENT
Receives MsgBFTPrePrepare, runs PSA, and agrees to value and broadcasts MsgBFTPrepare.
PREPARE_SENT
COMMIT_SENT
END
Receives 2F+1 MsgBFTPrepare's and broadcasts MsgBFTCommit.
Receives 2F+1 MsgBFTCommit's, then broadcasts the MsgNewPoly.

Figure 3-2: MPSS FSM Diagram

**PROPOSAL_RESPONSE_SENT** - The Participant waits for a **MSGBFTPREPREPARE** to begin the BFT agreement phase of MPSS. Upon receiving the message, the Participant runs the PSA to verify that the polynomial value being agreed on is indeed a valid value. If satisfied, the Participant will broadcast a **MSGBFTPREPARE** to all shareholders and transition to the PREPARE_SENT state. If unsatisfied, or if the node never receives a **MSGBFTPREPREPARE**, the node can still progress to the COMMIT_SENT or END states if it receives $2f+1$ **MSGBFTPREPARE**'s or **MSGBFTCOMMIT**'s respectively.

**PREPARE_SENT** - In this state the Participant waits until it receives $2f+1$, distinct **MSGBFTPREPARE** messages, including its own, so that it can broadcast a **MSGBFTCOMMIT** to all of the Participants and move into the COMMIT_SENT state. Similar to the PROPOSAL_RESPONSE_SENT state, the node can proceed to the END state if it receives $2f+1$ **MSGBFTCOMMIT**'s before it receives $2f+1$ **MSGBFTPREPARE**'s.

**COMMIT_SENT** - Once a node has received $2f+1$ **MSGBFTCOMMIT**'s, the protocol has agreed on a new share distribution polynomial, and the node can broadcast the **MSGNEWPOLY** to the members of the new group. When the node receives $f+1$ acknowledgments, it can transition to the END state.

**END** – The terminating state of the protocol.

The Primary node acts as both a Primary and a Participant throughout the protocol. Here is the Primary State Machine:

**PROPOSAL_WAIT** - This is the initial state of the Primary when the protocol begins. The Primary broadcasts a **MSGSTARTPROTOCOL** to all shareholders to start sending MSGPROPOSAL's. Upon receiving $2f+1$ **MSGPROPOSAL**'s, the Primary collects all the proposals into a proposal set and broadcasts a **MSGPROPOSALSET** message to all of the shareholders, moving into the PROPOSAL_SET_SENT state.

**PROPOSAL_SET_SENT** - The Primary runs the online PSA as it receives **MSGPROPOSALRESPONSE** messages. Once the stop condition is reached, the Primary broadcasts a **MSGBFTPREPREPARE** message to all of the shareholders to begin the BFT agreement phase and moves into the PREPREPARE_SENT state.

**PREPREPARE_SENT** – The Primary immediately sends a **MSGBFTPREPARE** to all participants and moves to the PREPARE_SENT state, whereupon it progresses through the rest of the protocol states like a Participant.

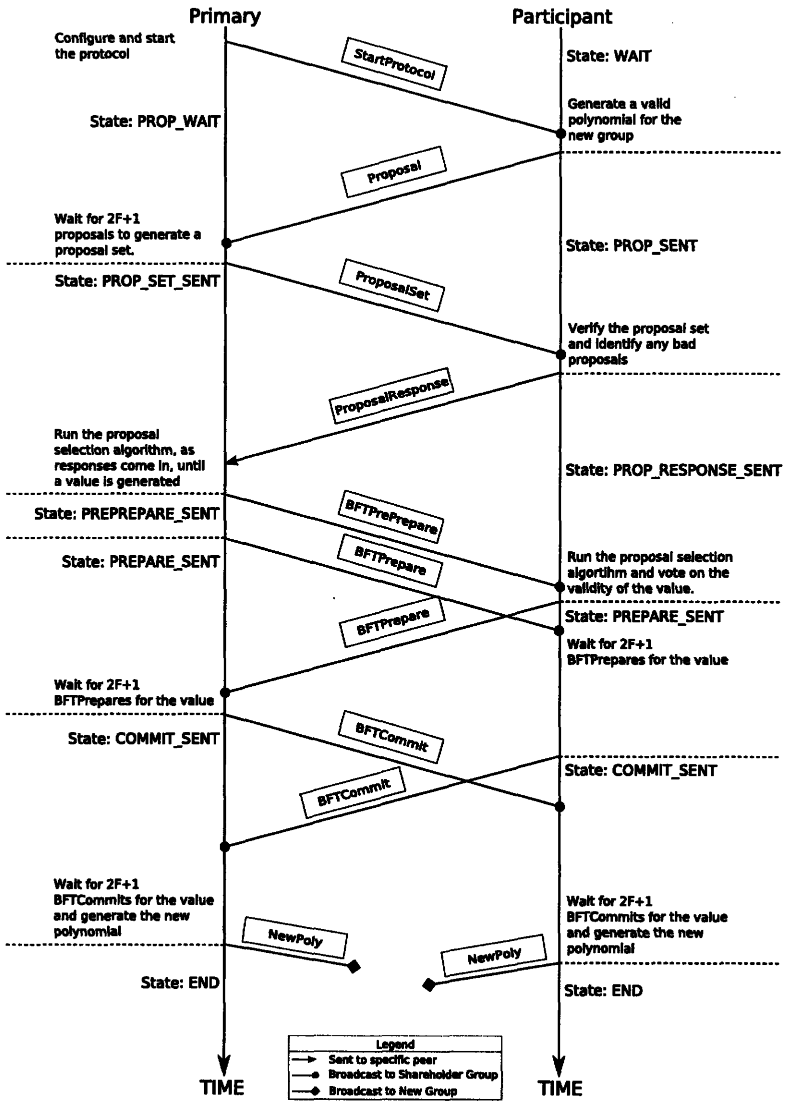
Primary
Participant
Configure and start the protocol
StartProtocol
State: WAIT
State: PROP_WAIT
Generate a valid polynomial for the new group
Proposal
Wait for 2F+1 proposals to generate a proposal set.
State: PROP_SENT
State: PROP_SET_SENT
ProposalSet
Verify the proposal set and identify any bad proposals
ProposalResponse
Run the proposal selection algorithm, as responses come in, until a value is generated
State: PROP_RESPONSE_SENT
State: PREPREPARE_SENT
BFTPrePrepare
State: PREPARE_SENT
BFTPrepare
Run the proposal selection algorithm and vote on the validity of the value.
State: PREPARE_SENT
BFTPrepare
Wait for 2F+1 BFTPrepares for the value
Wait for 2F+1 BFTPrepares for the value
State: COMMIT_SENT
BFTCommit
State: COMMIT_SENT
BFTCommit
Wait for 2F+1 BFTCommits for the value and generate the new polynomial
Wait for 2F+1 BFTCommits for the value and generate the new polynomial
NewPoly
NewPoly
State: END
State: END
Legend
Sent to specific peer
Broadcast to Shareholder Group
Broadcast to New Group
TIME
TIME

Figure 3-3: Timeline of the MPSS protocol

Figure 3-3 illustrates how the Primary and Participant state machines interact to form the underpinnings of the protocol. Although this timeline shows the ideal progression of the protocol, nodes that are behind can still participate in the agreement and share transfer stages.

## 3.2 MPSS with Verifiable Accusations

A variant of MPSS, which is also explored in the thesis, is the scheme that uses Verifiable Accusations.

The goal of the MPSS with Verifiable Accusations (MPSSVA) scheme is to make the proposal selection process more efficient. Under the original MPSS scheme, a node had no way of determining whether incoming **MSGPROPOSALRESPONSE**'s were honest or malicious. Therefore both the accuser and the accused nodes' proposals were removed from the selection process to ensure security. MPSSVA improves upon this process by including verifiable accusations in the **MSGPROPOSAL-RESPONSE**, which allows other nodes to cryptographically verify the claims and accurately identify whether the accuser or the accused node is malicious. In order to implement this scheme, some protocol objects and methods must be modified and augmented to make use of verifiable accusations:

**Proposal** - The proposal object must now use forward-secure, identity-based encryption to encrypt the contained information, the details of which can be found in [Sch07]. The message size increases due to the encryption scheme.

**Accusation** - An accusation object consists of an accusation against a node along with the identity based authentication key of the accuser node.

**AccusationSet** - An accusation set contains all of the accusations from a particular node.

**MSGPROPOSALRESPONSEACCUSE** - This is the **MSGPROPOSALRE-SPONSE** object for MPSSVA with an AccusationSet. The functionality of this message remains the same.

**PROPOSAL_WAIT (Primary State)** - The duration of the time the Primary spends in this state can be shortened with verifiable accusations. The primary only needs to receive $f+1$ **MSGPROPOSAL**'s instead of the original $2f+1$, to create the proposal set object and move into the PROPOSAL_SET_SENT state.

Figure 3-4: PSA for Verifiable Accusations (PSAVA) Pseudocode

```
1.  responses ← 0
2.  props ← set of all proposals in MsgProposalSet
3.  foreach MsgProposalResponse $R$ from distinct node $i$
4.    foreach accusation$_{i,j}$ ∈ $R$.AccusationSet
5.      if accusation$_{i,j}$ is valid
6.        props ← props − $\{j\}$
7.      else
8.        props ← props − $\{i\}$
9.    responses ← responses + 1
10.   if responses $\geq 2f+1$
11.     stop
```

**Proposal Selection Algorithm with Verifiable Accusations (PSAVA)** - The proposal selection algorithm becomes almost trivial, since each valid accusation in a **MSGPROPOSALRESPONSEACCUSE** results in exactly one exclusion, the algorithm acts as a counter for at least $2f+1$ responses to select a final set of proposals to generate the new polynomial. This version of the algorithm may require fewer node responses than the original algorithm. Since exclusions will only remove the faulty nodes, the algorithm may converge on the selection more quickly.

# Chapter 4

# Evaluation

To implement and evaluate the MPSS protocol we built a system that adheres to the protocol specifications outlined in the MPSS [Sch07] design. Then we evaluated the system's performance using values of $f$ in the range of 1 to 7. Throughout the evaluation process we are only interested in the performance for the "normal" (i.e. non-faulty) behavior of the system, since this will be the most common case for the system in practice. We used this approach to evaluate the original MPSS scheme as well as MPSSVA.

## 4.1 Implementation

### 4.1.1 Pragmatic Development

MPSS is a system that intertwines state-of-the-art cryptography, Byzantine fault tolerance, and distributed systems. Due to the complexity of these components, we chose to focus our efforts on the most significant aspects of the protocol and implement the rest through simulation. For example, MPSS relies on several cryptographic tools for which efficient designs have been proposed only recently, such as forward-secure signatures and forward-secure identity-based encryption. To the best of our knowledge, no implementations of these are publicly available. Rather than building prototypes that may not be entirely correct or reflective of the performance of a production implementation, we chose to simulate these methods using time delays proportional to the number of operations (i.e. modular multiplications and exponentiations) required, and incorporated the space requirement of the encryption scheme into the message sizes.

We focused our implementation on the MPSS redistribution protocol, focusing more on the Byzantine fault tolerance and distributed systems aspects, rather than the cryptographic and mathematical details of the system.

### 4.1.2 Development Methodology

The development methodology was an iterative process that used the concepts and protocol designs from Schultz's thesis [Sch07] to build a high level system, then incorporate the lower level details of the protocol, validated the behavior of the protocol, and finally optimized for performance. Unit testing was performed at each of these steps to ensure correctness of the system. The goal of building a modular, skeletal implementation was to to ease the future development of the different MPSS schemes. This methodology led to a better system design by decoupling components and increasing the robustness of the system as a whole.

Our protocol can run over either TCP or UDP. TCP is not the right choice in practice because it has several properties that are unneeded baggage for MPSS (e.g., message ordering and retransmission of messages is not relevant). However, we ran our tests using TCP instead of UDP because our UDP implementation has other issues, e.g., for large broadcasts, it fills the kernel's network buffer space, resulting in dropped packets.

An important aspect of developing the MPSS system was to make it easily configurable to deploy and run on multiple machines under different environments and different settings. The flexibility of the system facilitated diverse performance testing which resulted in a strong evaluation of the system.

Scaling the system involved progressively increasing the number of nodes in the protocol from $f = 1$ to $f = 7$, and from local machines to more distributed and diverse environments, such as Emulab [WLS+02]. We scaled to a maximum of $f = 7$

because previous work [Che04] has shown that it is unlikely that a larger value of $f$ will be needed in practice. Emulab was essential to scale the system to these dimensions, since our local PMG computer lab did not have the resources necessary for large values of $f$.

### 4.1.3 Development Environment

MPSS was implemented under the following development environments. Within the Programming Methodology Group, we used Fedora Core 6, Linux based machines with processors of at least 600MHz. All machines were connected in a 1Gb/sec Ethernet Local Area Network (LAN). MPSS itself was developed in C++, using the GNU C++ Compiler Version 4.1.1. MPSS used the following external C++ libraries: Botan, the BSD-licensed cryptography library version 1.6.4, and NTL version 5.4.2., a C++ library for number theory developed by Victor Shoupe. Python version 2.4.4 was used to write the scripts necessary run the protocol on remote machines and collect the protocols performance results in an automated fashion. GNU Profiler was used to examine the performance of the system and identify possible optimizations. GNUPlot version 4.0 was used to graph the performance measurements.

## 4.2 Experimental Framework

We evaluated our system using two testing frameworks: a local testbed and an Emulab testbed. We tested locally during development to assess and validate the behavior of the system for small values of $f$, and to be able to rapidly prototype the system. The Emulab testbed was used for real performance testing of the system as we scaled to large values of $f$.

### 4.2.1 Local Testbed

The first set of evaluations was performed under the local PMG environment. The PMG environment consists of 10 Linux based PCs running Fedora Core 6 with a range of 600MHz 2.0GHz of processing power connected to each other on a 1Gb/sec Ethernet LAN. None of these machines were purely dedicated, since they are all shared by members in PMG. Since there were not enough physical machines to test MPSSs scalability up to $f = 7$, getting the necessary number of nodes required using multiple ports per machine. A variety of environment setups were used to gage the correlation of physical nodes to performance. The local testbed was mainly used to get rough estimates on the practicality and performance of the system. Small tweaks and changes to the protocol could be analyzed quickly on a real system, and most of the testing of the protocol was performed on this testbed before scaling to the upper extremes.

### 4.2.2 Emulab Testbed

To evaluate MPSS in a more realistic setting where nodes may be distributed across the world with different bandwidth and network delays, we used the Emulab Total Network Testbed [WLS+02]. Emulab is a service provided by the University of Utah that allows researchers to develop, debug, and test distributed programs in a simulated network environment. The researcher is able to configure the network topology and behavior, such as throughput and latency, in order to test under desired settings. The researcher can also specify the machine types to use in the experiment. We set up the environment with homogeneous 3.0GHz machines with 2GB of RAM. Once an experiment is active on the Emulab network, the researcher has full root access to the deployed machines, and is free to run the program of his or her choosing. Using Emulab, we were able to get performance results for $f = 7$ with dedicated machines and configure the network to resemble a realistic scenario for MPSS. We ran performance tests on two different network topologies for each value of $f$:

**Low Latency** - All machines were simulated to be connected to a high throughput (100Mb/s), low latency LAN. This test resembles a system where all nodes reside in a single, high speed data center. Although unrealistic, this topology allows us to examine the behavior of the protocol under near-optimal conditions.

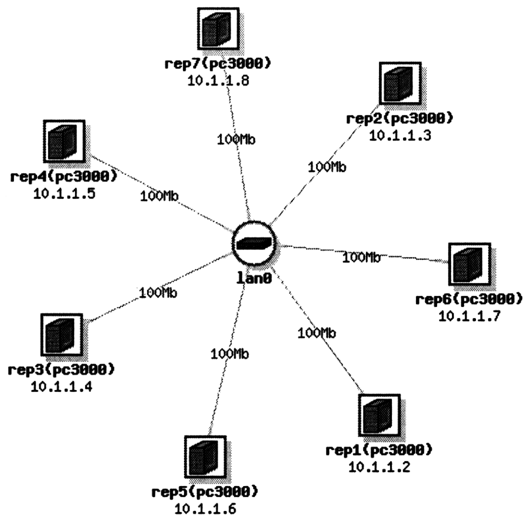

Figure 4-1: Low latency network topology test for $f = 2$.

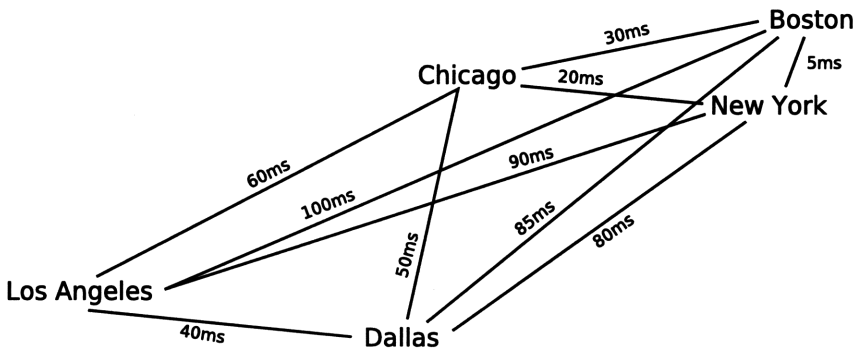

Figure 4-2: A sample geographic distribution of data centers across the U.S. with ping times.

**High Latency** - Nodes are located in data centers with geographic distribution and data independence from each other, such that there is delayed communication. All nodes are operating with high bandwidth (100Mb/s) connections with varying latencies, ranging from 5ms to 100ms delays. This deployment is of most interest since it represents the most realistic scenario for an MPSS system, where nodes are spatially distributed from each other, Figure 4-2 shows an example. We consider the spatial distribution of nodes to be uniform, and that the Primary is located near the geographic center of the network.

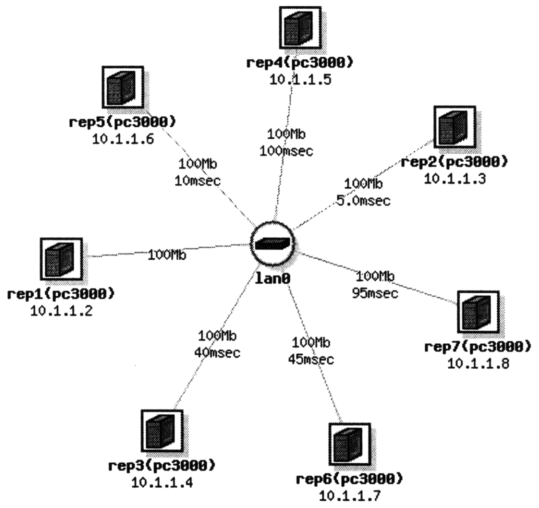

Figure 4-3: High latency network topology for $f = 2$.

## 4.3 Evaluation Results

### 4.3.1 Performance Metrics

To determine the practicality and scalability of MPSS, we identified a set of key performance metrics that the protocol evaluation must reflect: network throughput, protocol duration, and maximum load on each node. Network throughput is important because it is the real cost of running the protocol. Making sure the protocol's duration is minimal is important because a protocol that takes too long is not viable in practice. Additionally, the longer the protocol takes to complete the more feasible it becomes for an adversary to discover the secret within a given epoch. By evaluating each of these metrics while scaling the protocol from $f = 1$ to $f = 7$, we can assess the true practicality of the system under realistic scenarios.

### 4.3.2 Validation Metrics

During the testing of MPSS we created validation metrics to ensure appropriate system behavior. We generated benchmarks to compare the protocol's performance to its theoretical expectations. To do so, we formulated $f$-dependent relationships for the different performance metrics, namely number of messages in the protocol, total data throughput, and protocol duration. Each validation metric is based on ideal system behavior, where we define ideal behavior for the system as operating with the minimum amount of message passing. Our analysis is based on the specifications from [Sch07] for the Primary node and the derivations for the message behavior and bandwidth requirements for MPSS and MPSSVA can be found in Appendix A and Appendix B respectively. Below we show the message sizes with "$.s$" for both schemes, and present our protocol duration analysis.

#### Message Sizes

To compute message size, the space requirement of the cryptographic scheme must be taken into account. Figure 4-4 shows the cryptographic space costs for MPSS and MPSSVA, and Figures 4-5 and 4-6 display the message sizes for each scheme for the different values of $f$.

| Parameter | MPSS | MPSSVA |
|---|---|---|
| Encryption Space | 768 bytes | 1088 bytes |
| Signature Space | 1024 bytes | 768 bytes |

Figure 4-4: MPSS and MPSSVA Cryptographic Space Requirements

| Message | $f = 1$ | $f = 2$ | $f = 3$ | $f = 4$ | $f = 5$ | $f = 6$ | $f = 7$ |
|---|---|---|---|---|---|---|---|
| **MSGSTARTPROTOCOL**.$s$ | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| **MSGPROPOSAL**.$s$ | 6408 | 13128 | 22152 | 33480 | 47112 | 63048 | 81288 |
| **MSGPROPOSALSET**.$s$ | 1160 | 1240 | 1320 | 1400 | 1480 | 1560 | 1640 |
| **MSGPROPOSALRESPONSE**.$s$ | 1072 | 1072 | 1072 | 1072 | 1072 | 1072 | 1072 |
| **MSGBFTPREPREPARE**.$s$ | 4372 | 6596 | 8820 | 11044 | 13268 | 15492 | 17716 |
| **MSGBFTPREPARE**.$s$ | 1036 | 1036 | 1036 | 1036 | 1036 | 1036 | 1036 |
| **MSGBFTCOMMIT**.$s$ | 1036 | 1036 | 1036 | 1036 | 1036 | 1036 | 1036 |
| **MSGNEWPOLY**.$s$ | 3328 | 5824 | 9472 | 14272 | 20224 | 27328 | 35584 |

Figure 4-5: MPSS Messages Sizes (bytes) for $f$ = 1 to 7.

| Message | $f = 1$ | $f = 2$ | $f = 3$ | $f = 4$ | $f = 5$ | $f = 6$ | $f = 7$ |
|---|---|---|---|---|---|---|---|
| **MSGSTARTPROTOCOL**.$s$ | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| **MSGPROPOSAL**.$s$ | 7432 | 15112 | 25096 | 37384 | 51976 | 68872 | 88072 |
| **MSGPROPOSALSET**.$s$ | 864 | 904 | 944 | 984 | 1024 | 1064 | 1104 |
| **MSGPROPOSALRESPONSE**.$s$ | 816 | 816 | 816 | 816 | 816 | 816 | 816 |
| **MSGBFTPREPREPARE**.$s$ | 3052 | 4724 | 6396 | 8068 | 9740 | 11412 | 13084 |
| **MSGBFTPREPARE**.$s$ | 780 | 780 | 780 | 780 | 780 | 780 | 780 |
| **MSGBFTCOMMIT**.$s$ | 780 | 780 | 780 | 780 | 780 | 780 | 780 |
| **MSGNEWPOLY**.$s$ | 3072 | 5568 | 9216 | 14016 | 19968 | 27072 | 35328 |

Figure 4-6: MPSSVA Messages Sizes (bytes) for $f$ = 1 to 7.

### Protocol Duration

The duration of the protocol is an accumulation of the computation time for each message and the travel time for each message across the network. To generate a formula we first need to define the computations required for each type of message. We have 4 types of computations: encryption($E$), decryption($D$), message signing($S$), and signature verification($V$). Other computations such as generating the proposals, generating commitments, and validating message objects are not taken into account in this model. The table below denotes the computations necessary for sending and receiving each message. It is important to note that the encryption and signature of a message needs to be performed only once before broadcasting that message, however the decryption and verification of a message must be done for each incoming message.

| Message | Sending | Receiving |
|---|---|---|
| **MSGPROPOSAL** | $(3f+1)$E+S | D+V |
| **MSGPROPOSALSET** | S | V |
| **MSGPROPOSALRESPONSE** | S | V |
| **MSGNEWPOLY** | $(3f+1)$E+S | D+V |

Figure 4-7: MPSS Messages Computation Formulas

Using the relationships from message behavior and data throughput, we can find the total computation time for the Primary (we use "*.cs*" and "*.cr*" to denote send and receive computation time, respectively).

$$\begin{aligned}
\text{Comp. Time} &= \text{Computations to Send} + \text{Computations to Receive} \\
&= ((3f+1)\textbf{MSGPROPOSAL}.cs + (1)\textbf{MSGPROPOSALSET}.cs \\
&\quad +(1)\textbf{MSGPROPOSALRESPONSE}.cs + (3f+1)\textbf{MSGNEWPOLY}.cs) \\
&\quad +((2f+1)\textbf{MSGPROPOSAL}.cr + (1)\textbf{MSGPROPOSALSET}.cr \\
&\quad +(2f+1)\textbf{MSGPROPOSALRESPONSE}.cr) \\
&= (((3f+1)E+S)+(S)+(S)+((3f+1)E+S)) \\
&\quad +(((2f+1)(D+V))+(V)+((2f+1)V)) \\
&= 2(3f+1)E+4S+(2f+1)D+(2(2f+1)+1)V \\
&= (6f+2)E+4S+(2f+1)D+(4f+3)V
\end{aligned}$$

The transit time for all the messages is shown below:

$$\text{Transit Time} = \frac{\text{Bytes Sent} + \text{Bytes Received}}{\text{Bandwidth}} = \frac{\text{Data Throughput}}{\text{Bandwidth}} + \text{Latency}$$

Therefore, the total protocol duration can be calculated:

$$\begin{aligned}\text{Protocol Duration} &= \text{Computation Time} + \text{Transit Time}\\ &= (6f+2)E + 4S + (2f+1)D + (4f+3)V + \frac{\text{Data Throughput}}{\text{Bandwidth}}\\ &\quad +\text{Latency}\end{aligned}$$

Using the models above with the appropriate system environment parameters yield reasonable performance benchmarks for our MPSS system. We also use the same approach to generate benchmarks for MPSSVA: The analysis can be found in Appendix B. We ran environment tests to estimate the computation parameters: $E$,$D$,$S$, and $V$ and their respective space requirements. The results are shown in Figure 4-8.

### 4.3.3 Evaluation

Based on the metrics we identified, we tested MPSS's performance by computing the number of messages sent over the network, the total amount of bytes sent and received by nodes in the protocol, and the total duration of the protocol. For each test, we ran the system without malicious nodes in order to discern standard behavior of the system. We ran each of these tests at least 10 times for each value of $f$. We emphasized the performance of the Primary node since it handles the highest load in the system.

We ran the same tests for MPSSVA in order to compare the differences between MPSS and MPSSVA. It should be noted that our implemented MPSS and MPSSVA systems simulated many of the advanced forward-secure encryption, forward-secure signature, and commitment generation techniques. However, since none of the techniques have production implementations, we approximated the overhead of these schemes to the best of our knowledge. Hence, we only claim the performance our systems to be on the same order of magnitude of complete implementations. Figure 4-8 denotes the different system parameters we used for MPSS and MPSSVA.

| Parameter | MPSS | MPSSVA |
|---|---|---|
| $E$ | 12ms | 17ms |
| $D$ | 12ms | 17ms |
| $S$ | 10ms | 10ms |
| $V$ | 2ms | 2ms |

Figure 4-8: MPSS and MPSSVA System Parameters

The legend for each graph is located in its top right corner. The rest of this section describes the results of our analysis.

## Message Behavior

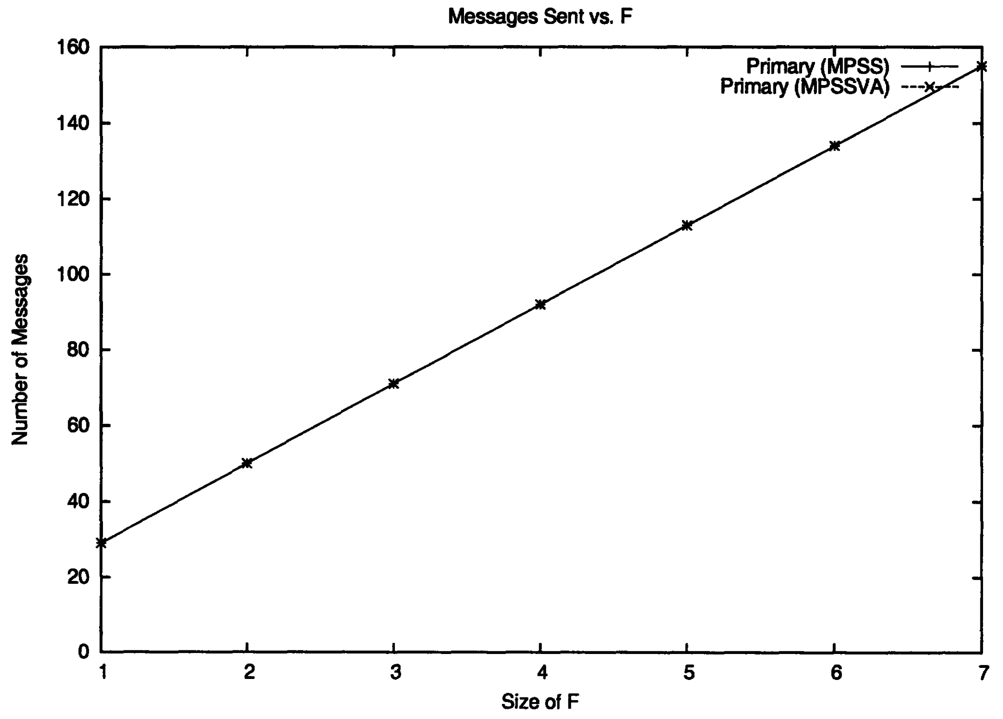

Figure 4-9: MPSS vs. MPSSVA Messages Sent Comparison at the Primary

Figure 4-9 shows the number of messages sent by the Primary in MPSS and in MPSSVA grows linearly with $f$. As expected, the Primary sends the same number of messages in both schemes. Similar results can be found for the number of messages received by the Primary.

### Bandwidth Requirement

Figure 4-10 shows that MPSSVA requires more bandwidth than the original MPSS scheme. This is due to the verifiable accusation cryptographic overhead, which increases message size. The important thing to note is that bandwidth utilization grows on the order of $O(f^2)$, resulting from the quadratic growth of **MSGPROPOSAL** message size. At $f = 7$, the Primary is sending close to 2MB of data in the protocol.

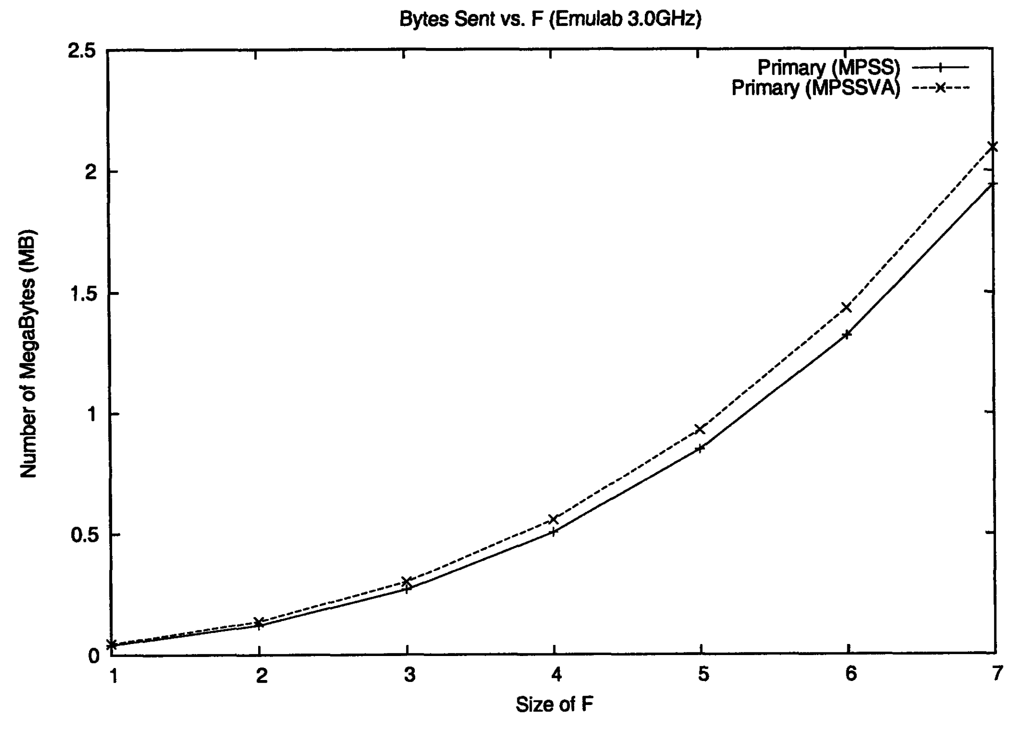

Figure 4-10: MPSS vs. MPSSVA Bytes Sent Comparison at the Primary.

### Protocol Duration

We tested the total time it takes to run the protocol from the Primary's perspective. We measured the duration of the protocol as being the time from when the Primary initiates the protocol to the share transfer of the polynomial to the new group of shareholders. We compared the latency of MPSS and MPSSVA under the two different network topologies. The goal of the tests is to show that the protocol terminates in a reasonable amount of time and determine whether verifiable accusations is useful. MPSSVA might be more expensive because of cryptographic costs and larger message sizes, but it might be faster because the Primary does not have to wait for as many **MSGPROPOSAL**'s.

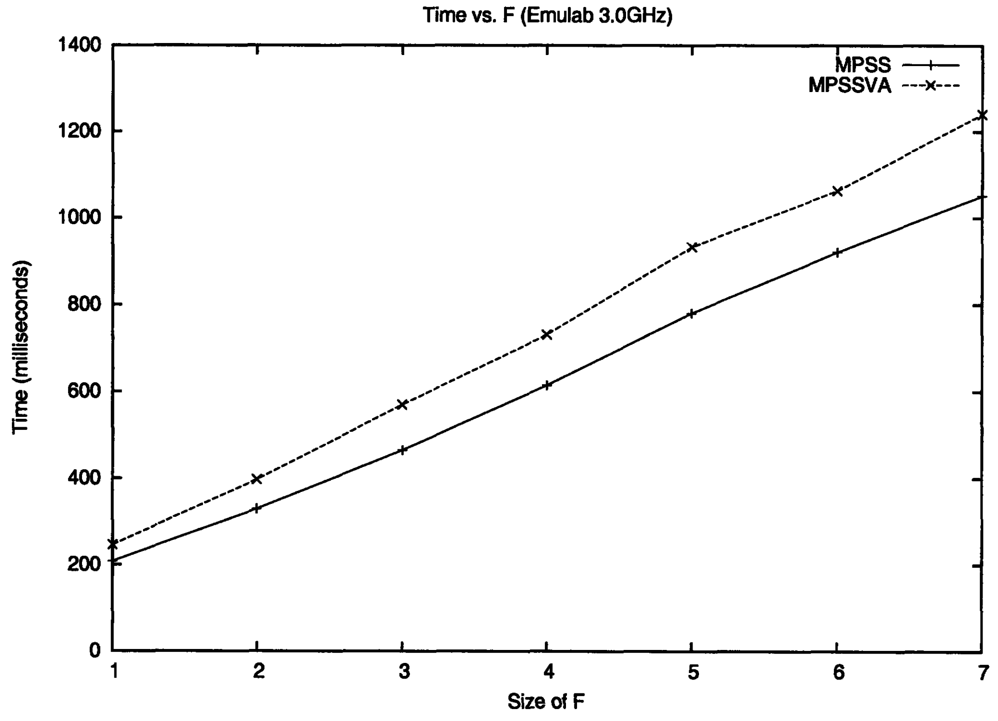

Figure 4-11: MPSS vs. MPSSVA Duration Comparison (low latency)

Figure 4-11 shows the performance of MPSS and MPSSVA in the low latency topology. MPSS clearly outperforms MPSSVA in this scenario. Since there is no latency in the network, the cryptographic overhead is the major difference between the two schemes. MPSSVA has more expensive encryption and decryption costs, resulting in slower performance. As $f$ increases, we see the gap between MPSS and MPSSVA increase as well, due to the linear increase in the number of cryptographic computations.

Based on the system parameters, not all of the overhead shown in Figure 4-11 is due to cryptographic overhead. Some of the overhead is due to the time required to send all of the messages. For instance, for MPSS at $f = 7$, subtracting the cryptographic overhead, 568ms, from the total duration, 1051ms, results in 483ms of bandwidth cost, or 33Mb/s. The effective bandwidth is less than expected, operating between 25Mb/s and 33Mb/s; presumably caused by the Emulab network using shared links between the nodes.

Figure 4-12 shows the results of MPSS and MPSSVA in the high latency topology, a more realistic network. We found that MPSS performs slightly better than MPSSVA when $f$ is small. Verifiable accusations do not seem to have a significant impact on the protocol, so in practice MPSS should suffice.

Due to the prolonged duration of the protocol under this topology, the number of test runs was reduced, accounting for some of the measurement variations in the graph (e.g., there is an anomaly at $f = 4$ that is not fundamental to the protocol).

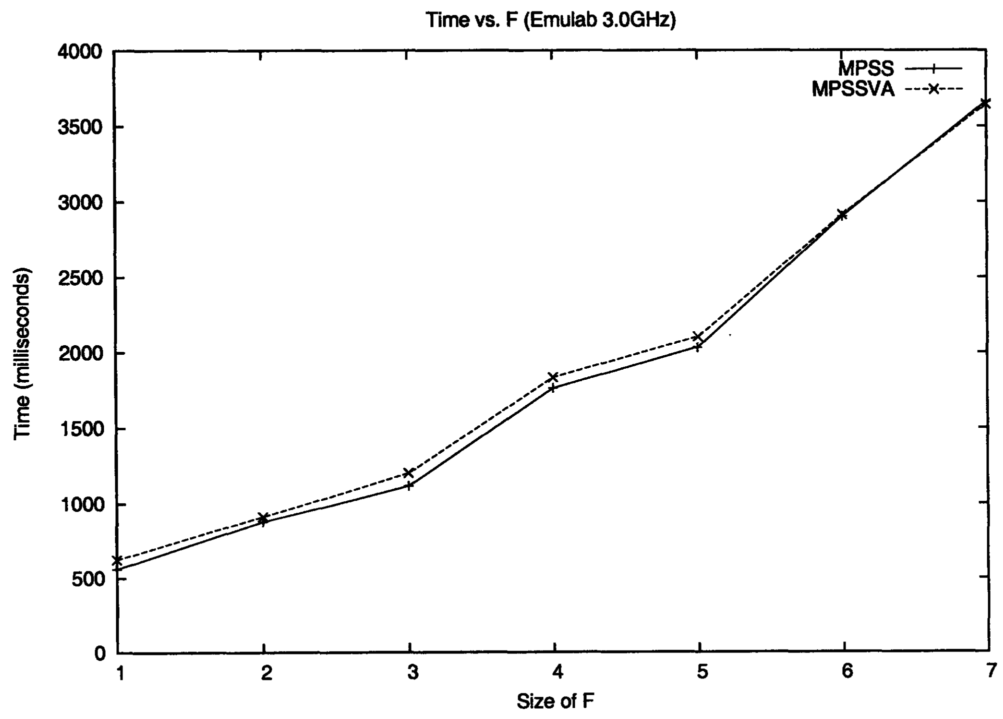

Figure 4-12: MPSS vs. MPSSVA Duration Comparison (high latency)

Within the parameters of our scenario MPSS performs well, running in under 4 seconds for the largest values of $f$. The bandwidth is also reasonable for the protocol, considering how infrequently it is run. MPSS's short protocol duration, reasonable data consumption, and low resource intensity make it a practical system.

# Chapter 5

# Conclusion

## 5.1 Discussion

We have presented implementations of MPSS and MPSSVA and evaluated their performance in both a high-speed LAN and also in a deployment resembling the Internet. Our results show that we were able to scale these systems to the desired values of $f$, in order to ascertain the practicality of these systems. We showed that MPSS performs at least as well as MPSSVA in both latency and bandwidth for every network topology that we examined, but both systems are scalable and would be practical for a real usage scenario.

## 5.2 Future Work

To gain more insight into MPSS and other secret sharing schemes, more implementation and evaluation of these schemes is necessary. For MPSS in particular, we focused on implementing the redistribution protocol and simulated the cryptographic aspects of the system. A logical next step would be to implement the entire MPSS system with all of the cryptographic elements. We only focused on analyzing MPSS performance under normal, non-faulty behavior. Future analysis should focus on MPSS's performance with faulty nodes.

# Appendix A

# MPSS Validation Metrics

Here are the calculations for the MPSS Validation benchmarks.

## A.1 Message Behavior

The Primary's message behavior can be divided into its message sending behavior and its message receiving behavior. Our analysis assumes the primary sends messages to itself, e.g., it sends itself the start command, and also the proposal. Our implementation works this way, but clearly it could be improved to avoid these messages. Under ideal circumstances the Primary will broadcast to $3f + 1$ nodes, so the message sending behavior can be characterized as follows:

$$\begin{aligned}
\text{Msgs Sent} &= (3f+1)\textbf{MSGSTARTPROTOCOL'}s + (3f+1)\textbf{MSGPROPOSAL'}s \\
&\quad +(1)\textbf{MSGPROPOSALSET} + (1)\textbf{MSGPROPOSALRESPONSE} \\
&\quad +(3f+1)\textbf{MSGBFTPREPREPARE'}s + (3f+1)\textbf{MSGBFTPREPARE'}s \\
&\quad +(3f+1)\textbf{MSGBFTCOMMIT'}s + (3f+1)\textbf{MSGNEWPOLY'}s \\
&= (3f+1) + (3f+1) + 1 + 1 + (3f+1) + (3f+1) + (3f+1) + (3f+1) \\
&= 6 * (3f+1) + 2 \\
&= 18f + 8
\end{aligned}$$

For messages received, we use the same analysis. Since it is operating ideally, only $2f+1$ nodes are required to continue to different stages in the protocol. We get the following relationship for messages received:

$$\begin{aligned}
\text{Msgs Rec} &= (1)\textbf{MSGSTARTPROTOCOL}'s + (2f+1)\textbf{MSGPROPOSAL}'s \\
&\quad +(1)\textbf{MSGPROPOSALSET} + (2f+1)\textbf{MSGPROPOSALRESPONSE}'s \\
&\quad +(1)\textbf{MSGBFTPREPREPARE} + (2f+1)\textbf{MSGBFTPREPARE}'s \\
&\quad +(2f+1)\textbf{MSGBFTCOMMIT}'s \\
&= 1+(2f+1)+1+1+(2f+1)+(2f+1)+(2f+1) \\
&= 4*(2f+1)+3 \\
&= 8f+7
\end{aligned}$$

## A.2 Bandwidth Requirement

Data throughput is calculated in a similar manner to the messages in the previous section. The only difference is that the size of the message is taken into account (we use "$.s$" to denote message size). The actual size of each message is dependent on the value of $f$ and the cryptographic overhead; Figure 4-5 shows the different message sizes for each value of $f$. Therefore, we get the following results for bytes sent and received from the Primary:

$$\begin{aligned}
\text{Bytes Sent} &= (3f+1)\textbf{MSGSTARTPROTOCOL}.s + (3f+1)\textbf{MSGPROPOSAL}.s \\
&\quad +(1)\textbf{MSGPROPOSALSET}.s + (1)\textbf{MSGPROPOSALRESPONSE}.s \\
&\quad +(3f+1)\textbf{MSGBFTPREPREPARE}.s + (3f+1)\textbf{MSGBFTPREPARE}.s \\
&\quad +(3f+1)\textbf{MSGBFTCOMMIT}.s + (3f+1)\textbf{MSGNEWPOLY}.s
\end{aligned}$$

$$\begin{aligned}
\text{Bytes Rec} \quad = \quad & (1)\mathbf{MSGSTARTPROTOCOL}.s + (2f+1)\mathbf{MSGPROPOSAL}.s \\
& +(1)\mathbf{MSGPROPOSALSET}.s + (2f+1)\mathbf{MSGPROPOSALRESPONSE}.s \\
& +(1)\mathbf{MSGBFTPREPREPARE}.s + (2f+1)\mathbf{MSGBFTPREPARE}.s \\
& +(2f+1)\mathbf{MSGBFTCOMMIT}.s
\end{aligned}$$

# Appendix B

# MPSSVA Validation Metrics

Here are the calculations for the MPSSVA Validation benchmarks.

## B.1 Message Behavior

$$\begin{aligned}
\text{Msgs Sent} &= (3F+1)\textbf{MSGSTARTPROTOCOL'}s + (3F+1)\textbf{MSGPROPOSAL'}s \\
&\quad +(1)\textbf{MSGPROPOSALSET} + (1)\textbf{MSGPROPOSALRESPONSEACCUSE} \\
&\quad +(3F+1)\textbf{MSGBFTPREPREPARE'}s + (3F+1)\textbf{MSGBFTPREPARE'}s \\
&\quad +(3F+1)\textbf{MSGBFTCOMMIT'}s + (3F+1)\textbf{MSGNEWPOLY'}s \\
&= (3F+1) + (3F+1) + 1 + 1 + (3F+1) + (3F+1) + (3F+1) + (3F+1) \\
&= 6 * (3F+1) + 2 \\
&= 18F + 8
\end{aligned}$$

$$\begin{aligned}
\text{Msgs Rec} &= (1)\textbf{MSGSTARTPROTOCOL'}s + (F+1)\textbf{MSGPROPOSAL'}s \\
&\quad +(1)\textbf{MSGPROPOSALSET} + (2F+1)\textbf{MSGPROPOSALRESPONSE'}s \\
&\quad +(1)\textbf{MSGBFTPREPREPARE} + (2F+1)\textbf{MSGBFTPREPARE'}s
\end{aligned}$$

$$\begin{aligned}
&+(2F+1)\mathbf{MSGBFTCOMMIT}'s \\
&= 1+(F+1)+1+1+(2F+1)+(2F+1)+(2F+1) \\
&= 3*(2F+1)+(F+1)+3 \\
&= 7F+7
\end{aligned}$$

## B.2 Bandwidth Requirement

$$\begin{aligned}
\text{Bytes Sent} = {} & (3F+1)\mathbf{MSGSTARTPROTOCOL}.s + (3F+1)\mathbf{MSGPROPOSAL}.s \\
& +(1)\mathbf{MSGPROPOSALSET}.s + (1)\mathbf{MSGPROPOSALRESPONSE}.s \\
& +(3F+1)\mathbf{MSGBFTPREPREPARE}.s + (3F+1)\mathbf{MSGBFTPREPARE}.s \\
& +(3F+1)\mathbf{MSGBFTCOMMIT}.s + (3F+1)\mathbf{MSGNEWPOLY}.s
\end{aligned}$$

$$\begin{aligned}
\text{Bytes Rec} = {} & (1)\mathbf{MSGSTARTPROTOCOL}.s + (F+1)\mathbf{MSGPROPOSAL}.s \\
& +(1)\mathbf{MSGPROPOSALSET}.s + (2F+1)\mathbf{MSGPROPOSALRESPONSE}.s \\
& +(1)\mathbf{MSGBFTPREPREPARE}.s + (2F+1)\mathbf{MSGBFTPREPARE}.s \\
& +(2F+1)\mathbf{MSGBFTCOMMIT}.s
\end{aligned}$$

## B.3 Protocol Duration

$$\begin{aligned}
\text{Comp. Time} &= \text{Computations to Send} + \text{Computations to Receive} \\
&= ((3F+1)\mathbf{MSGPROPOSAL}.cs + (1)\mathbf{MSGPROPOSALSET}.cs \\
&\quad +(1)\mathbf{MSGPROPOSALRESPONSE}.cs + (3F+1)\mathbf{MSGNEWPOLY}.cs) \\
&\quad +((F+1)\mathbf{MSGPROPOSAL}.cr + (1)\mathbf{MSGPROPOSALSET}.cr \\
&\quad +(2F+1)\mathbf{MSGPROPOSALRESPONSE}.cr) \\
&= (((3F+1)E+S)+(S)+(S)+((3F+1)E+S))
\end{aligned}$$

$$
\begin{aligned}
& +(((F+1)(D+V))+(V)+((2F+1)V)) \\
= \; & 2(3F+1)E+4S+(F+1)D+((F+1)+(2F+1)+1)V \\
= \; & (6F+6)E+4S+(F+1)D+(3F+2)V
\end{aligned}
$$

$$
\begin{aligned}
\text{Protocol Duration} &= \text{Computation Time} + \text{Travel Time} \\
&= (6F+6)E+4S+(F+1)D+(3F+2)V+\frac{\text{Data Throughput}}{\text{NetworkSpeed}}
\end{aligned}
$$

# Bibliography

[BF01] D. Boneh and M. Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology–CRYPTO 2001*, Lecture Notes in Computer Science, pages 213–229. Springer-Verlag, 19-23 August 2001.

[Che04] Kathryn Chen. Authentication in a reconfigurable byzantine fault tolerant system. Master's thesis, Massachusetts Institute of Technology, July 2004.

[CL99] M. Castro and B. Liskov. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. In *Technical Memo MIT/LCS/TM590.* MIT Laboratory for Computer Science, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. In *ACM Transactions on Computer Systems*, pages 398–461, November 2002.

[Fel87] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 427–437, New York City, 25-27 May 1987.

[HJKY95] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing, or how to cope with perpetual leakage. In Don Coppersmith, editor, *Advances in Cryptology–Crypto '95*, volume 963 of *Lecture Notes in Computer Science*, pages 457–469. Springer-Verlag, 27-31 August 1995.

[OY91] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In *Proceedings of the 10th (ACM) Symposium on the Principles of Distributed Computing*, pages 51–61, 1991.

[RR78] L. Adleman R. Rivest, A. Shamir. A method for obtaining digital signatures and public-key cryptosystems. In *Communications of the ACM*, pages 120–126, 1978.

[Sch07] David Schultz. Mobile proactive secret sharing. Master's thesis, Massachusetts Institute of Technology, January 2007.

[Sha79] A. Shamir. How to share a secret. In *Communications of the (ACM)*, pages 612–613, 1979.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.