Authentication in a Reconfigurable Byzantine Fault Tolerant System

by

Kathryn Chen

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

August 2004 [September 2004]

Author .....
Department of Electrical Engineering and Computer Science
July 23, 2004

Certified by .....
Barbara Liskov
Ford Professor of Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Authentication in a Reconfigurable Byzantine Fault Tolerant System

by

Kathryn Chen

Submitted to the Department of Electrical Engineering and Computer Science
on July 23, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Byzantine (i.e. arbitrary) faults occur as a result of software errors and malicious attacks; they are increasingly a problem as people come to depend more and more on online services. Systems that provide critical services must behave correctly in the face of Byzantine faults. Correct service in the presence of failures is achieved through replication: the service runs at a number of replica servers and as more than a third of the replicas are non-faulty, the group as a whole continues to behave correctly. We would like the service to be able to authenticate data. Authenticated data is data that more than a third of the service is willing to sign.

If a long-lived replicated service can tolerate f failures, then we do not want the adversary to have the lifetime of the system to compromise more than f replicas. One way to limit the amount of time an adversary has to compromise more than f replicas is to reconfigure the system, moving the responsibility for the service from one group of servers to a new group of servers. Reconfiguration allows faulty servers to be removed from service and replaced with newly introduced correct servers. Reconfiguration is also desirable because the servers can become targets for malicious attacks, and moving the service thwarts such attacks.

In a replicated service, we would like the service to be able to authenticate data. Authenticated data is data that more than a third of the service is willing to sign. Any party that knows a public key can verify the signature. Such a scheme is a threshold signature scheme. The signers in a threshold signature scheme each know some part of a secret. Because we would like to reconfigure the system, we need to transfer the knowledge of the secret to the new servers and we want to disable the old servers from signing in the future. Such a scheme is called secret refreshing.

This thesis describes TSPSS, a *threshold signing* and *proactive secret sharing* protocol. TSPSS can be used by asynchronous reconfigurable Byzantine fault tolerant service replicas to perform threshold signing and secret refreshing. TSPSS uses combinatorial secret sharing, which involves an exponential number of shares in f. We implement TSPSS to evaluate how well it scales and whether it performs well enough to be used in practice. We find that TSPSS performs well enough to be used for

$f = 1$, is arguably good enough for $f = 2$, and is impractical for $f = 3$. Thus, a better solution to this problem is needed.

Thesis Supervisor: Barbara Liskov
Title: Ford Professor of Engineering

Acknowledgments

I would like to thank my advisor, Barbara Liskov, for her guidance, patience, and support. Our discussions helped me to refine the design presented in this thesis, and her careful and critical comments significantly improved the content and presentation of this thesis. I also want to thank Rodrigo Rodrigues for all of the guidance he provided. This thesis would not have been possible without Barbara and Rodrigo's supervision.

I would also like to thank all the people at the Programming Methodology Group who were very supportive and contributed to a great work environment. Special thanks to Sameer Ajmani and Ben Leong for their day-to-day help with development tools and cheering me up on the rough days.

I have been very fortunate to have made great friends at MIT. I could not have survived my undergraduate and MEng years without them. From lending an ear for me to vent to, to helping me to convert the graphs in this thesis to the proper format, they have been very supportive and they have enriched my life at MIT. I would especially like to thank Justin Mazzola Paluska and Jeremy Wong.

Lastly, I would like to thank my mother, Kuei Chen. I feel truly blessed to have her love and support. She is such an extraordinary woman and she inspires me to always strive to do better.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Byzantine (i.e. arbitrary) faults occur as a result of software errors and malicious attacks; they are increasingly a problem as people come to depend more and more on on-line services. Systems that provide critical services must behave correctly in the face of Byzantine faults. Correct service in the presence of failures is achieved through replication: the service runs at a number of replica servers and as long as enough replicas are non-faulty, the group as a whole continues to behave correctly.

In [2], Castro and Liskov propose a replication algorithm for Byzantine fault-tolerance in asynchronous systems that offers good performance and strong correctness guarantees provided that no more than 1/3 of the servers fail. The amount of time an adversary has to compromise more than 1/3 of the servers is called the *window of vulnerability*. If no mechanism is used to reduce the window of vulnerability, the window is the lifetime of the system. To reduce the window of vulnerability, Rodrigues and Liskov[12] propose to reconfigure the system, moving the responsibility for the service from one group of servers to a new group of servers. Reconfiguration allows faulty servers to be removed from service and replaced with newly introduced correct servers. Reconfiguration is also desirable because the servers can become targets for malicious attacks, and moving the service thwarts such attacks.

Secure reconfiguration requires a way for the current set of servers to be able to prove to other processors in the system (e.g., processes running code that uses the service) that it truly is the current server set. Otherwise, faulty servers such as those

previously removed from the system can pretend to be the current server set and cause the system to behave incorrectly.

To achieve secure reconfiguration, the current server set can generate a signed proof that allows anyone who knows a particular public key to verify the signature. The corresponding private key is a secret and each of the current servers has a share of the secret. The signature can be generated if and only if more than one third of the current replica servers sign with their shares of the private key.

This thesis describes TSPSS, a *threshold signing* and *proactive secret sharing* protocol that addresses the authentication needs of a reconfigurable Byzantine fault tolerant service that operates in an asynchronous environment. This thesis describes how the current server set generates a signature and how the old server set transfers the secret to the new server set.

Our secret refreshing protocol is based on the APSS protocol described by Zhou et al in [15]. We improve on that work in two aspects. The first is that our refresh protocol works across 2 sets of servers, with one set of servers transferring their collective knowledge of the secret to another set of servers that previously did not know about the secret. The second difference is that where agreement is needed, we use the Castro and Liskov's replication algorithm to achieve agreement. Castro and Liskov's replication algorithm [2] is a practical algorithm for state machine replication [13, 9] that tolerates Byzantine faults. For the rest of this thesis, we will refer to the Castro and Liskov replication algorithm as BFT.

1.1 TSPSS Overview

TSPSS is intended for use in a Byzantine fault tolerant system. In Byzantine fault tolerant systems, the number of servers, n, running a service is usually $3f + 1$, where f is the number of faulty servers that the system can tolerate within a window of vulnerability. This is the same as saying that less than $1/3$ of the servers running the service can be faulty. It has been proven that n must be greater than $3f$ to reach Byzantine agreement. Although TSPSS will work for $n > 3f + 1$, we assume

$n = 3f + 1$ for the rest of this thesis.

To tolerate f faulty servers, we want f or fewer servers to be unable to generate a valid signature. Thus, at least one good server must sign in order for a valid signature to be generated. We want a group of correct servers to be able to generate a valid signature. In the cryptography literature, this type of signing scheme is an $(n, k+1)$ *threshold signature* scheme [6], with $n = 3f + 1$ and $k = f$. In an (n, k+1) threshold signature scheme there are n shares of a private key and one corresponding public key. A message signed by any $k + 1$ of the private keys can be verified by the public key. A message signed by k or fewer of the private keys will not be verified by the public key.

We want to combine a threshold signature scheme with a proactive secret sharing scheme, with the threshold scheme's private key, in its entirety, being the secret of an $(n, k + 1)$ proactive secret sharing, also with $n = 3f + 1$ and $k = f$.

An $(n, k + 1)$ *secret sharing* [14] for a secret s is a set of n random shares such that (i) s can be recovered with knowledge of $k + 1$ shares, and (ii) no information about s can be derived from k or fewer shares. *Share refreshing* [6, 8] is where servers periodically create a new and *independent* set of secret shares for the same secret, replacing the old shares with the new shares. Secret sharing with share refreshing is called *proactive secret sharing.*

Property (ii) of secret sharing ensures that f or fewer servers cannot reconstruct the private key. Since our secret is a private key, we never want to reconstruct the secret. To generate signatures without reconstructing the secret, each server's threshold signature private key must be derivable from the shares of the secret that the server has.

When a new set of servers become responsible for the service, this new set of servers will need to be able to generate a signature that corresponds to the same public key. And, the old server set should no longer be able to generate a signature that can be verified by the public key. Allowing new servers to sign can be accomplished through secret refreshing from the old server set to the new server set with $n = 3f + 1$ and $k = f$. Once the new servers have the new shares, correct old servers forget any

information that can be used to infer old or new shares. Once the $2f + 1$ correct old servers forget their shares, there are not enough old shares to produce a valid signature.

1.2 Thesis Outline

The remainder of this thesis is organizes as follows. Chapter 2 describes the system model, assumptions, and correctness conditions. In Chapter 3, we describe the threshold signing and proactive secret sharing protocol. Chapter 4 describes how the protocol is implemented. We then evaluate the costs of running TSPSS, in terms of computation time and network bandwidth used, in Chapter 5. We present our conclusions in chapter 6.

Chapter 2

System Model and Assumptions

Consider a system composed of a set of *processors* that communicate through a network. For each window of vulnerability, there are n processors that are responsible for the service during that period of time. While a processor is responsible for the service, we call it a *server*. Once a processor is no longer responsible for the service, it is no longer a server.

Servers hold the shares of a secret. Each processor is assumed to have an individual public/private key pair. Each server is assumed to know the public key of all other processors in the system. Cryptographic techniques are employed to provide confidentiality and authenticity for messages. It is assumed that the adversary is computationally bound and that the factoring problem and the discrete logarithm problem are hard so that the adversary cannot subvert these cryptographic techniques.

2.1 Attacks and Failures

TSPSS is intended for use in environments like the Internet, where failures and attacks can invalidate assumptions about timing. Thus, we assume an *asynchronous system* where there is no bound on message delay or processor execution speed. We assume that the network through which the processors are connected is composed of fair links. A *fair link* is a communication channel between processors that does not necessarily

deliver all messages sent, but if a processor sends sufficiently many message to a single destination, then one of those message is correctly delivered. Messages in transit may be read or altered by the adversary.

As we mentioned earlier, less than 1/3 of the servers in a window of vulnerability can be faulty. A server is either *correct* or *faulty*. A faulty server can stop executing the protocol, deviate from its specified protocol in an arbitrary manner, and/or corrupt or disclose locally stored information. A correct server follows the protocol and does not corrupt or disclose locally stored information. A server is considered correct in a time interval τ if and only if it is correct throughout interval τ. Otherwise it is considered faulty in time τ.

With the system model and assumptions described above, an adversary is allowed to do any or all of the following in order to cause the most damage to the replicated service:

- compromise and coordinate up to f faulty servers within any *window of vulnerability*
- delay messages or correct servers by arbitrarily finite amounts,
- launch eavesdropping, message insertion, corruption, deletion, and replay attacks

2.2 Window of Vulnerability Definition

The *window of vulnerability* in TSPSS is defined in terms of events rather than the passing of time. To give a precise definition for the window of vulnerability, we assign version numbers $(v_0, v_1, v_2, ...)$ to shares, secret sharings, runs, and participating service replicas. Each execution, or *run*, of TSPSS generates a new secret sharing. A sharing is composed of the set of random shares which form a secret s:

- Servers initially store shares with version number v_0.
- If a run is executed with shares having version number $v_{old} = v_i$, then this run and the resulting new shares have version number $v_{new} = v_{i+1}$.

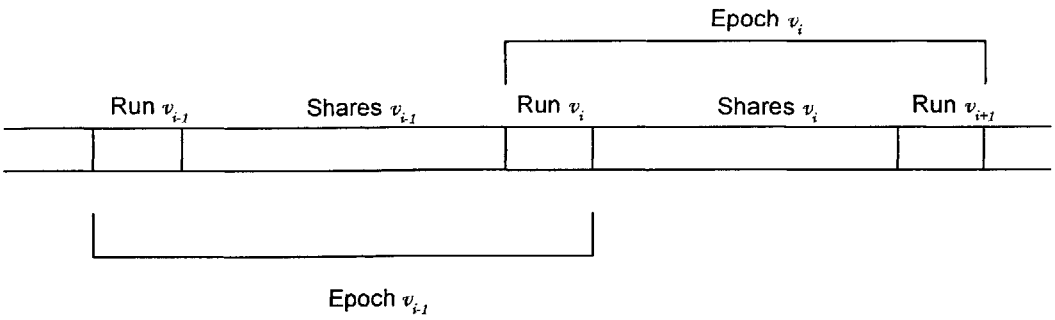


Figure 2-1: Relationship Between Runs, Shares, and Epochs

- A secret sharing is assigned the same version number as its shares.

Run v_{new} starts when some correct server initiates run v_{new} locally. Run v_{new} terminates locally on a correct server once the server has forgotten all information pertinent to v_{old} shares. Run v_{new} terminates globally at the earliest time t that the run has terminated locally on each correct server that participated in run v_{new}.

An *epoch* v_i is defined to be the interval from the start of the run v_i to the global termination of run v_{i+1}. The relations between old and new runs, shares, and epochs, and their surrounding runs, shares and epochs is illustrated in Figure 2-1. In a successful attack, an adversary must collect $f + 1$ or more shares all with the same version number. Since $f + 1$ or more servers must be compromised during the same epoch in order to collect these shares, we define the window of vulnerability to be an epoch.

Chapter 3

The TSPSS Protocol

In this chapter, we describe the TSPSS protocol. We first describe how to split up the secret into shares. Then, we describe how to generate a signature using these shares and how to refresh the shares so that new nodes learn new shares of the same secret, which are independent from the old shares.

In describing the protocol, we assume that communications channels are secure because authenticity and confidentiality can be achieved through signing and encrypting the messages. We will use *nodes* interchangeably with *processors* so that we can use n_i to stand for a node, and s_i to stand for a secret share.

3.1 Combinatorial Secret Sharing

The way we're going to share a secret and achieve a threshold access structure is to use the *combinatorial secret sharing* [7] technique described by Ito et al. We split the secret up into l shares, where $l = \binom{3f+1}{f}$. A share is given a label from 1 to l. We find all possible combinations of f nodes from our $3f + 1$ nodes, and we label each group of f nodes with a number from 1 to l. Then, a node n_i gets a share s_i with label t if node n_i is not in a group with label t.

We give an example for $f = 1$. For $f = 1$, $l = 4$ so we have four shares: s_1, s_2, s_3, s_4. We generate all possible groups of $f = 1$ servers and label them with numbers 1 to l (i.e., $g_1 = \{n_1\}$, $g_2 = \{n_2\}$, $g_3 = \{n_3\}$, $g_4 = \{n_4\}$). Then we give share s_1 to all

index i	shares s_i	groups g_i	nodes given share s_i
1	s_1	$g_1 = \{n_1\}$	n_2, n_3, n_4
2	s_2	$g_2 = \{n_2\}$	n_1, n_3, n_4
3	s_3	$g_3 = \{n_3\}$	n_1, n_2, n_4
4	s_4	$g_4 = \{n_4\}$	n_1, n_2, n_3

Table 3.1: Combinatorial Secret Sharing for $f = 1$

nodes *not* in group g_1, (i.e., nodes n_2, n_3, n_4). Similarly, we give share s_2 to all nodes not in group g_2, s_3 to all nodes not in g_3, and s_4 to all nodes not in g_4. Table 3.1 summarizes the results for $f = 1$.

Now, by construction,

- Each share is held by 2f+1 nodes.
- No set of less than f+1 nodes have all of the shares
- Any set of f+1 or more nodes will have all of the shares[1]

3.2 Threshold Signing

With combinatorial secret sharing, we can use the RSA cryptosystem to generate threshold signatures. Normally, RSA has one key pair: a private key and a corresponding public key. The public key is denoted as (N, e), where $N = pq$ and p, q are large primes of roughly the same size. Define $\phi \equiv (p-1)(q-1)$. The private key is d where $ed \equiv 1 \bmod \phi$ and e is the public RSA exponent. To sign a message m, compute signed message $c = m^d \bmod N$. To verify the message, compute $m' = c^e \bmod N$. Since $m' = c^e \bmod N = m^{ed} \bmod N$, and $ed \equiv 1 \bmod \phi$, the signature is *valid* if and only if m' equals m.

The private key d is the secret we share combinatorially. As such, we must split it into l shares: d_1, d_2,..., d_l. We call a signature generated by one share of the secret key a *partial signature*. The nice thing about RSA is that if the l shares sum to

[1]For any given share, only f nodes do not have the share, every other node has it. Then, for each share, given a group of any f+1 of the 3f+1 nodes, at least one of the nodes has the share

d, then the product of all the partial signatures (i.e., $c_i = m^{d_i} \bmod N$), is the RSA signature that we expect in the normal scheme described above. I.e., if

$$d = \sum_{i=1}^{l} d_i \bmod N \tag{3.1}$$

then

$$c = m^d \bmod N = \prod_{i=1}^{l} m^{d_i} \bmod N \tag{3.2}$$

.

Thus, when starting the system, the trusted dealer generates the shares satisfying the constraints posed by Equation 3.1 by picking random integers that sum to the secret:

- For $2 \leq i \leq l$, choose random integers $d_i \in [-lN^2..lN^2]$
- Compute $d_1 = d - \sum_{i=2}^{l} d_i$

After generating the shares, give them to the nodes following the method described in Section 3.1. We use d_i's as the s_i's discussed in Section 3.1.

Notice that each node has more than one share and that each share must be used exactly once when generating a valid signature. We generate a valid signature as follows. When asked to sign something, each node first determines whether the message should be signed[2]. If the message should be signed, the node generates one partial signature per share of the secret it holds. Then, the node sends all of the partial signatures to the requester. Recall that each share is held by $2f + 1$ nodes of which at most f are faulty. Thus, the requester determines a partial signature to be correct when it receives the same partial signature from $f + 1$ nodes. The requester can compute the complete signature by multiplying together the l correct partial signatures.

For example, for $f = 1$, node 1 asks itself and nodes 2, 3, 4 to sign. Then, node 1 receives replies from itself and nodes 2 and 3. Now, node 1 has the following partial

[2]See Section 4.2 for how node determine whether a message should be signed

signatures:

From node i	c_1	c_2	c_3	c_4
1		X	X	X
2	X		X	X
3	X	X		X

Node 1 has received $f+1$ copies of each partial signature, thus knows that the partial signatures it received are correct.

3.2.1 Alternatives and Optimizations

Since many nodes have the same share, if at least $f+1$ of the nodes are behaving correctly, it is unnecessary for them to all generate partial signatures. Generating a partial signature means creating a digest of the message and taking that digest to the power of the subshare (i.e., performing an exponentiation). Performing an exponentiation is an expensive computation whereas adding partial shares together is a cheap computation. Furthermore, sending all of the partial signatures back to the requester takes up more network bandwidth.

Thus, here is an alternative we will call TS2. TS2 is an optimistic approach that reduces the amount of computing and bandwidth used by receivers of sign requests. In TS2, the requester picks a group of $f+1$ nodes. We call this group g_{sign}. The requester asks for a signature from each node in g_{sign}, also telling them which f other nodes are in g_{sign}. Then each node in g_{sign} has a notion of which shares it is *responsible* for when generating this signature. The way a node determines which shares it is responsible for is that each node knows which shares every other node has. Given the f other nodes in g_{sign}, node n_i is responsible for all the shares that n_i has, minus all the shares held by the nodes in g_{sign} with smaller id numbers. A node returns a signature generated by all the shares it is responsible for.

We give an example for $f=1$. Suppose we choose $g_{sign} = \{n_1, n_2\}$. Then, node n_1 is responsible for all the shares it has (i.e., s_2, s_3, s_4) because there are no nodes with a

smaller id number than 1. n_2 is responsible for the shares it has (i.e., s_1, s_3, s_4) minus the shares node n_1 has, leaving n_2 responsible for s_1. n_1 returns ($m^{s_2+s_3+s_4} \bmod N$) to the requester. n_2 returns ($m^{s_1} \bmod N$) to the requester.

When the requester receives the $f+1$ responses, it can multiply the signatures together and check to see if the combined signature is valid. If the signature does not verify, or $f+1$ responses are not received within a set amount of time, the requester can try the same thing again with another group of $f+1$ nodes and keep trying until it gets responses that verify. Since there are at least $2f+1$ correct nodes, the requester is guaranteed to receive responses that verify if it tries enough times.

A hybrid of the full signing protocol described in Section 3.2 and TS2 can also be used. In the hybrid form, the requester first asks for the TS2 form of signing from $f+1$ nodes. If the requester does not receive the $f+1$ responses within a certain amount of time, or the responses do not result in a valid signature, then instead of trying the TS2 protocol again with another set of $f+1$ nodes, the requester initiates the full signing protocol, where each node returns one partial signature per share it has.

3.3 Share Refreshing

For transferring the secret from the old nodes to the new nodes, we describe the protocol for a single run v_{i+1}. We call run v_{i+1} the current run, shares and witnesses with version number v_i old shares and witnesses, nodes with shares of version number v_1 the old nodes, shares and witnesses with version number v_{i+1} the new shares and witnesses, and nodes computing shares with version number v_{i+1} the new nodes.

Secret sharing involves two operations: `split` and `reconstruct`. The split operation generates a set of random shares from a secret s; we call these shares a sharing of s. The `reconstruct` operation recovers s from certain sets of shares. A unique label is associated with each share and sharing of s.

For share refreshing, we want to split each of the l old shares into l subshares. From the subshares, we want to construct l new shares such that i) the secret s can

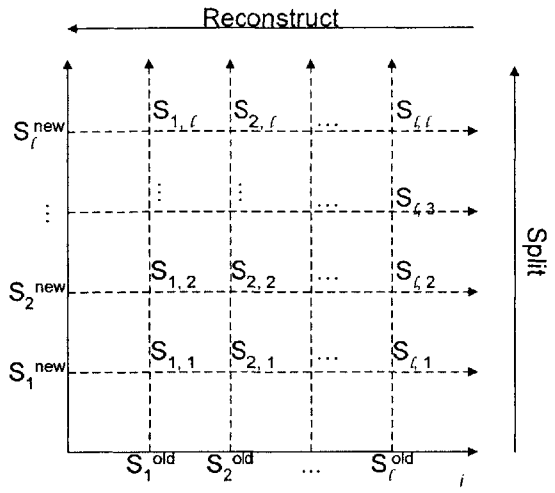


Figure 3-1: Splitting Old Shares and Constructing New Shares

be reconstructed from the l new shares and ii) we can perform threshold signing with the l new shares. The relation between the old shares and the secret is that the old shares sum to the secret. So, the new shares will satisfy (i) and (ii) if the new shares sum to the secret.

We will generate subshares for each share the same way we generated shares for our secret in Section 3.1. For a share s_i, an old node generates subshares $s_{i,j}$'s by choosing random integers $s_{i,j} \in [-lN^2..lN^2]$ for $2 \leq j \leq l$, and setting $s_{i,1} = s_i - \sum_{j=2}^{l} s_{i,j}$. To construct the new share s_i^{new}, a new node computes $s_i^{new} = \sum_{j=1}^{l} s_{j,i}$. Figure 3-1 illustrated the relationship between the old shares and the new shares.

In share refreshing, each old node, for each share s_i that it holds, generates l subshares. The holder then makes a subshare available to only those new nodes that store a share constructed from that subshare. The old nodes are indexed from 1 to n and the new nodes are also indexed from 1 to n. Which shares are held by an old node and which are held by a new node is determined by the combinatorial secret sharing described in Section 3.1 based on the nodes' indices. As an example, for $f = 1$, n_1^{old} has shares s_2^{old}, s_3^{old}, s_4^{old} and n_1^{new} has shares s_2^{new}, s_3^{new}, s_4^{new}

Recall that by construction, each share is held by $2f + 1$ nodes. If all nodes were correct, then for each share of the secret, we can designate 1 holder of the secret to generate subshares and send them to the appropriate new nodes. However, f of the

nodes can be malicious. And, malicious nodes can i) refuse to generate and send new subshares or ii) generate incorrect subshares that do not reconstruct to the correct new shares.

To tolerate faulty nodes that refuse to participate, we require that every node generate new subshares for each of the shares it holds. However, this means that the new nodes will receive more than one set of subshares per old share, and the new nodes have to agree on which of the many sets of subshares for an old share to use when constructing the new shares.

To achieve this agreement, we use the BFT agreement protocol. For each old share, the BFT agreement protocol decides which old node's set of subshares is used to construct the shares for the next epoch.

However, having multiple nodes generate subshares for each share does not help with the problem of a malicious node generating incorrect subshares. We use Feldman's Verifiable Secret Sharing [4] to decide whether a node's subsharing is correct.

We briefly describe Feldman's scheme in relation to our shares and subshares. As in the RSA cryptosystem, p and q are large primes and $N = pq$. Let g be a publically known constant that is an element of high order in Z_N^*. The *order* of g is t, where t is the least positive integer for which $g^t = 1 \bmod N$. The basic idea behind this mechanism is that for each share s_i, there is a *public* value $w_i \equiv g^{s_i} \bmod N$, which we call a *witness*. And, when a node generates subshares for a share, the node also computes and makes available to all new nodes a proof $p_{i,j}$ for every subshare $s_{i,j}$. A proof $p_{i,j}$ is defined to be:

$$p_{i,j} \equiv g^{s_{i,j}} \bmod N \tag{3.3}$$

From the homomorphic properties of the exponentiation function (i.e., $g^a g^b = g^{a+b}$) we know a subsharing for share s_i is correct if the product of all the subshare proofs for s_i is equal to the witness for s_i. i.e.,

$$w_i = \prod_{j=1}^{l} p_{i,j} \bmod N \tag{3.4}$$

In other word, if Equation 3.4 holds, and each proof is computed according to Equa-

tion 3.3, we know the subshares sum to the share because

$$\begin{aligned} w_i &= g^{s_i} \bmod N \\ &= g^{\sum_{j=1}^{l} s_{i,j}} \bmod N \\ &= \prod_{j=1}^{l} g^{s_{i,j}} \bmod N \\ &= \prod_{j=1}^{l} p_{i,j} \bmod N \end{aligned}$$

Thus, we use the proofs and witnesses to verify the validity of a node's subshares. An old node makes all of its proofs available to all of the new nodes. Each new node checks to see that i) for each subshare $s_{i,j}$ that the new node receives, the proof for $s_{i,j}$ was computed according to Equation 3.3 and ii) all the subshare proofs together satisfy Equation 3.4. To ensure that all new nodes see the same proofs, the old nodes write their proofs to the BFT agreement service. The new nodes read their proofs from the BFT agreement service.

Since any group of $f + 1$ nodes have all the shares, any group of $f + 1$ new nodes will receive all of the subshares for an old share. So, if $f + 1$ correct new nodes confirm that an old node's set of subshares is correct, then that old node's subsharing is known to be correct. When the BFT is deciding which old node's set of subshares to use for generating new shares, the BFT only considers those nodes' sets that have been said to be correct by $2f + 1$ of the new nodes. The BFT needs to hear from $2f + 1$ of the new nodes because f of those nodes may be faulty.

Note that the BFT is a replicated state machine. All the operations invoked on the BFT is ordered, and the ordering is the same at all of the nodes running a BFT service. Thus, for each share, the BFT chooses the subhshares of the first old node to have that share and receive $2f + 1$ confirmations from new nodes.

Once a correct set of subshares has been chosen for each old share, new shares can be computed, as shown in Figure 3-1. While computing the new shares, the new

nodes also compute the new witnesses.

$$s_i^{new} \equiv \sum_{j=1}^{l} s_{j,i} \tag{3.5}$$

$$w_i^{new} \equiv g^{s_i^{new}} \bmod N \equiv \prod_{j=1}^{l} g^{s_{j,i}} \tag{3.6}$$

Note that the indices for the subshares have been reversed. Also note that the $g^{s_{j,i}}$'s are the subshare proofs that have already been computed.

When a new node has computed the new shares and witnesses, it sends a *computed* message to all the old nodes and the new nodes saying that it has computed the new shares. Once an old node has heard from $2f + 1$ new nodes that the new shares have been computed, it can delete all its shares and subshares. Once a new node has heard from $2f + 1$ new nodes (which can include itself) that the new shares have been computed, the new node can delete all the subshares it received. The reason that old nodes and new nodes wait to hear from $2f + 1$ new nodes is because of the way we constructed our secret shares. By construction, any group of $f + 1$ share holders have all of the shares. Since there may be a maximum of f faulty nodes, $2f + 1$ new nodes guarantees that at least one correct new node has each share.

If a new node did not participate in the protocol for some reason (e.g., it was slow or it was disconnected from the other nodes) and it needs to know about its shares, it can ask other nodes for its shares and the new witnesses. The new node can trust a witness once it receives $f + 1$ copies of the same witness. And a new node can trust its new share by verifying it against the witness. Nodes will only send a share s_i to other nodes that are supposed to have s_i.

The way nodes makes shares and subshares available only to other appropriate nodes is by encrypting those shares and subshares. To guard against replay attacks where the adversary records the encrypted subshare messages, corrupts the receiving node after the epoch has ended, then plays the messages to the now corrupted receiver and learns those subshares, session keys are established and used to encrypt the shares and subshares. Session keys are established at the beginning of each epoch and deleted

at the end of that epoch.

This scheme can tolerate f faulty old nodes and f faulty new nodes. The secret is the sum of all the chosen subshares, as shown in Figure 3-2.

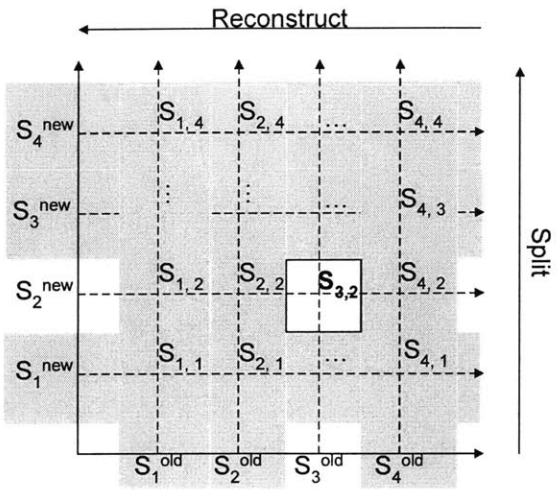


Figure 3-2: Subshares Known to a Group of f Old and f New Nodes

Any group of f faulty old nodes is missing at least one version v_i share of the secret. Any group of f faulty new nodes is missing at least one version v_{i+1} share of the secret. Thus, a group of f faulty old nodes combined with f faulty new nodes knows the secret minus some random number. For the combined group of f faulty old node and f faulty new nodes, finding the missing subshare would be equivalent to trying to guess the secret in the first place[3].

We present the operations provided by the BFT agreement service:

`write_proofs` This operation is invoked by old nodes. The BFT service knows which node invoked this operation and stores the proofs in the space allocated for that node.

`read_proofs(nodeID)` This operation is invoked by new nodes, with `nodeID` specifying which old node's proofs the new node wants to read. The BFT service

[3]The adversary is reduced to trying to find a number that matches the witness, or trying to find the number that decrypts correctly.

returns the proofs if they have been written. Otherwise, the BFT service returns a `NACK` indicating that the proofs are not currently available.

`node_ok(nodeID)` This operation is invoked by a new node, with `nodeID` specifying which old node's subshares the new node is confirming to be correct. The BFT service knows which new node invoked this operation and marks the old node with `nodeID` as being confirmed by the new node that invoked this operation. Once $2f + 1$ new nodes confirm an old node's subshares, BFT service goes through the shares that the old node has. For each share, if no other old node's subshares have already been chosen, this old node's subshares are chosen. Once the subshare set for each share has been chosen, the BFT service calculates the witnesses for the new shares as shown in Equation 3.6.

`read_chosen` This operation is invoked by new nodes. If a subshare set has been chosen for all shares, the BFT service returns which node's subshare set has been chosen for each share. Otherwise, the BFT service returns a `NACK` indicating that the subshare set for at least one share has not been chosen.

`read_new_witnesses` This operation is invoked by new nodes. If a subshare set has been chosen for all shares, return the corresponding witnesses for the new shares. Otherwise, the BFT service returns a `NACK` indicating that the subshare set for at least one share has not been chosen - thus, new witnesses cannot and have not been calculated.

Now, we present the steps of the protocol.

Each old node does the following:

1. Notify the new nodes to begin the refreshment protocol, with a refresh configuration. A refresh configuration consists of the old nodes, the new nodes, and the old witnesses. The refresh configuration is threshold signed.
2. For each share it holds, generate subshares and proofs.
3. Send encrypted subshares to the appropriate new nodes.

4. Write proofs to the BFT agreement service running at the new nodes by invoking `write_proofs`.
5. Wait for `computed` messages from $2f + 1$ of the new nodes. Once received, delete all shares, subshares, witnesses, and the session key for this epoch.

Each new node does the following:

1. Receive the begin message and refresh configuration. Upon verifying the threshold signature, proceed to the next step.
2. Start running the BFT agreement service with the other new nodes.
3. Receive and store the subshares.
4. Read the subshare proofs from the BFT agreement service by invoking `read_proofs`.
5. Check the proofs using Equations 3.3 and 3.4.
6. If the proofs for all of an old nodes' subshares are correct, invoke the `node_ok` operation with this old node's ID.
7. Check with the BFT service to see if there are chosen subshares for every share (`read_chosen`). If so, then compute new shares and read new witnesses (`read_new_witnesses`). Once have new shares and witnesses, then delete all subshares. If not, keep checking.
8. Send `computed` message to all other new and old nodes.
9. Once has received `computed` messages from $2f + 1$ new nodes, delete subshares received and current session key. Create and distribute new session key.

3.3.1 Alternatives and Optimizations

As in the signing protocol, if all nodes are correct, it is unnecessary for them all to generate subshares and corresponding proofs. Only one node needs to generate subshares and proofs for each share. In refreshing, a node has to do a significant amount of work for each share that it is responsible for. Thus, if we optimistically assume that all nodes are correct, we assign each of the $3f + 1$ nodes to be responsible for an equal number of shares and each node only creates and sends subshares and

proofs for the shares they are responsible for. To ensure that faulty nodes do not prevent the refresh protocol from completing, a new node can wait for a set amount of time after it receives the first refresh header to see if it receives and verifies all the subshares it needs to compute its new shares. If a node does not receive and verify all the subshares it needs after a set amount of time, it notifies all the old nodes that it needs more subshares. If an old node receives more than f of these messages, it will generate subshares and proofs for all the shares it has.

Another optimization is that instead of each new node directly telling the BFT agreement service when it has verified the subshares for an old node, each new node sends a signed confirmation back to the old node upon confirming that old node's subshares. Each old node collects $2f + 1$ of these signed confirmations and submits them to the BFT agreement service. If the $2f + 1$ signatures verify, then the BFT agreement service will consider that old node's subshares to be correct. The BFT agreement service then behaves as if it had received $2f + 1$ `node_ok`'s for that old node.

Chapter 4

Implementation

The TSPSS protocol has been implemented (without alternatives and optimizations) in 4799 lines of C++ code. It is meant to be used like a library by a Byzantine fault tolerant service.

The implementation uses the Castro and Rodrigues implementation of the BFT agreement protocol with BASE [1, 3], the SFS[10] crypt library, and the SFS async library. The BFT implementation provides agreement. The SFS crypt library extends the GNU MP library[5], providing an implementation of public-key encryption/decryption and signing/verifying with the Rabin algorithm. The SFS async library provides an infrastructure for event driven programming.

The implementation deviates from the TSPSS protocol in two places. First, during the refreshment protocol, the old nodes do not establish session keys for encryption. They use the public-key the program is initialized with for encryption throughout the lifetime of the program. Second, the threshold signing part of the implementation implements a hybrid scheme similar to the one described in Section 3.2.1 where a node first picks $f + 1$ nodes for signing and if that fails, initiates a full signing protocol. In the implementation, the first $f + 1$ nodes are always the group of $f + 1$ chosen for the first round of signing, which we will call the *quick sign*.

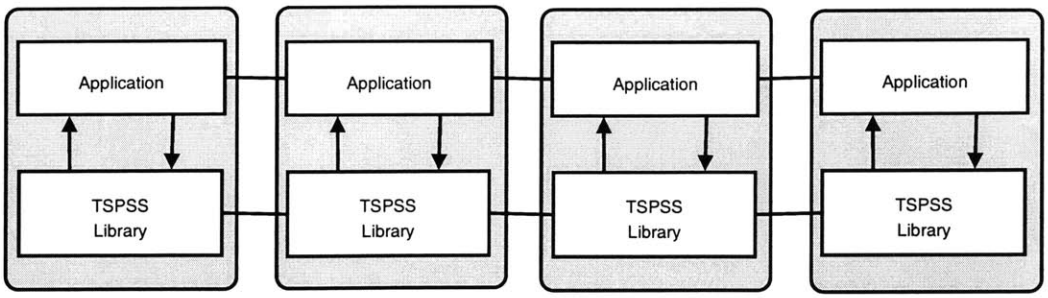


Figure 4-1: Each server in a Byzantine fault tolerant service runs the TSPSS library and it can invoke operations on the library through *local* unix sockets.

4.1 Software Architecture

Since the TSPSS library is designed to be used by a Byzantine fault tolerant service, we will call a service that uses the TSPSS library the TSPSS application. Each of the TSPSS application servers runs the TSPSS library code on a separate process. The application sends requests to and receives replies from the TSPSS library via local sockets. This ensures that the TSPSS library can trust the messages from the TSPSS application. Once the TSPSS application invokes a TSPSS library operation, the TSPSS library carries out the necessary protocol steps (including ones involving other nodes) before returning a reply to the application via another local socket if a reply is specified. Figure 4-1 illustrates the software architecture.

4.2 Threshold Signing Interface

The TSPSS library supports two signing related operations:

`register_message` Register a message with the TSPSS library. When the TSPSS library running at a node n_i receives sign requests from another node n_j, the TSPSS library at n_i will only sign the message if that message has been registered by the TSPSS application at node n_i.

`sign_message` Ask the TSPSS library to initiate the threshold signing protocol for a message m. The TSPSS library will contact other nodes and return a signature

```
struct sign_request_header{
   int request_type;
   int request_ID;
   int length;
}

struct sign_reply{
   int request_ID;
   char signature[SIGNATURE_SIZE];
}
```

Figure 4-2: Signature Request/Reply Headers

to the TSPSS application once it has assembled the signature based on the replies from the other server nodes running the TSPSS library.

A server running a Byzantine fault tolerant service registers the messages that it is willing to sign using `register_message`. It can request a signature using `sign_message`. If at least f other nodes have registered the same message, then the TSPSS library at the `sign_message` invoking node writes the successfully generated signature to another local socket for the application to read. Pseudo code for the reply format is given in Figure 4-2. The `request_type` can be `register_message` or `sign_message`. The `length` is the length of the message to be registered or signed. The `sign_request_header` is followed by the message to be registered or signed.

4.3 Secret Refreshing Interface

The TSPSS library supports two refresh related operations:

`start_refresh` The node at which this operation is invoked has version v_i shares and will start a version v_{i+1} refresh run if one has not already been started.

`get_witnesses` The node at which this operation is invoked currently has version v_i shares and will return version v_i witnesses.

To start a secret refreshing run, each server running the TSPSS application sends the TSPSS library running on the same node a `start_refresh` message followed by

```
struct refresh_header{
   int epoch;
   pss_location old_replicas[NUMBER_OF_REPLICAS];
   pss_location new_replicas[NUMBER_OF_REPLICAS];
   char witness[NUMBER_OF_SHARES][SIGNATURE_SIZE];
   char signature[SIGNATURE_SIZE];
}

struct pss_location{
   char verification_key[VERIFICATION_KEY_SIZE];
   sockaddr_in addr;
}
```

Figure 4-3: Refresh Header

the refresh header via a local unix socket. Thus, only code running on the same machine can ask the TSPSS library to start a refresh. Since the TSPSS application decides which nodes will be the service nodes in epoch v_{i+1}, the new nodes and their verification keys are included in the header. The TSPSS library code at those nodes will in turn send the refresh header to the new service nodes via TSPSS ports to initiate the TSPSS refresh protocol at the new nodes. To keep the interface for threshold signing and refreshing clean, we include everything the new nodes need to know in the refresh header so that it can all be threshold signed and sent to the new nodes. The refresh header needs to be threshold signed so that the new nodes can trust that this is a valid refresh request.

The refresh header includes the epoch started by this run of the refresh protocol, the network location and verification key for each old and new node, and the witnesses for version v_i secret shares. The TSPSS application can request the witnesses from TSPSS library, then register the entire header and request that the header be threshold signed as described in Section 4.2. Pseudo code for the refresh header is given in Figure 4-3.

If a node receives a refresh header with an epoch smaller than its own epoch, the node will not start the refresh protocol. Once a node completes the refresh protocol, it sets its own epoch to the epoch in the refresh header.

Chapter 5

Evaluation

In this section, we evaluate how well TSPSS scales in terms of f. Since we use combinatorial secret sharing, which requires that we split our secret into $l = \binom{3f+1}{f}$ shares, we know that the number of shares is exponential in f. For values of l in relation to f, see Table 5.1. To determine how large f can be before TSPSS becomes impractical, we consider the costs for signing and refreshing. For each, we consider the cost in terms of the network bandwidth used and the speed with which the protocol is completed successfully.

All experiments are run over a local area network(LAN). Experiments are run on two types of machines. Type I machines each have an Intel(R) Pentium(R) 4, 3.06GHz CPU and 2,064MB of RAM. Type II machines each have an Intel Pentium III, 596MHz CPU with 512MB of RAM. We only have four Type I machines and six type II machines.

f	$3f+1$	$l = \binom{3f+1}{f}$	$l' = \binom{3f}{2f}$
1	4	4	3
2	7	21	15
3	10	120	84
4	13	715	495

Table 5.1: f, $3f+1$, l, and l'

For the rest of this chapter, we will refer to the number of shares each node is assigned according to the combinatorial secret sharing scheme as l'. $l' = l - \binom{3f}{f-1}$ because there are l shares and each node is in $\binom{3f}{f-1}$ groups of f nodes.

5.1 Threshold Signing

5.1.1 Network Bandwidth Used

For an idea of the network bandwidth used by the signing protocol, we consider the sizes of the messages and the number of messages sent. For authentication, we use 1024 bit RSA and Rabin signatures. Each sign request (whether it is a quick sign request or a full sign request) consists of some header information, the message to be signed, and the requester's Rabin signature. In the quick sign phase of the protocol, a request is sent to $f+1$ nodes. If the full signing protocol is initiated, a request is sent to each of the $3f+1$ nodes in the system. Each reply has some header information, the partial RSA signature(s) requested (each of the size of the RSA signature), and the replier's Rabin signature. For a quick sign request, only one signature is requested from each of $f+1$ nodes. For a full signing request, l' signatures are requested.

Thus, the requesting node sends:

(number of receivers) $\times$ (header size + message size + Rabin signature size)

$\Rightarrow (f+1) \times$ (24 + message size + 128) bytes *(for quick sign)*

$\Rightarrow (3f+1) \times$ (24 + message size + 128) bytes *(for full signing)*

The replier sends:

header size + (number of RSA sigs $\times$ RSA sig size) + Rabin sig size

$\Rightarrow$ 24 + (1)128 + 128 bytes *(for quick sign)*

$\Rightarrow$ 24 + (l')128 + 128 bytes *(for full signing)*

For a message of 160 bits (20 bytes), we give the amount of data sent by requesters and repliers for quick sign and full sign in Table 5.2. We find that although the bandwidth used by the signing protocol is small for $f \leq 3$, the bandwidth increases quickly and becomes non-trivial $f > 3$

f	Requester (quick)	Requester (full)	Replier (quick)	Replier (full)
1	624	1,248	280	538
2	936	2,184	280	2,072
3	1,248	3,120	280	10,904
4	1,560	4,056	280	63,512

Table 5.2: Number of bytes sent by signature requesters and repliers during the *quick* sign phase of the signing protocol and the *full* sign phase of the signing protocol, based on calculations.

5.1.2 Speed of Protocol

Now we discuss how long it takes to run the protocol. We measure signing times for i) when there are no faulty nodes and ii) when there are f faulty nodes and one of the nodes chosen for a quick sign is faulty. For each f and for both (i) and (ii), we run 3 trials of 100 signatures each. We use one Type II machine for each of the first $2f$ nodes, and one Type I machine for each of the next $f + 1$ nodes. Recall that quick sign chooses the first $f + 1$ nodes to perform a quick sign and in our set up, the first $2f$ machines are Type II machines. The speed of the full signing protocol is determined by the $(2f + 1)^{th}$ fastest node. Thus, the quick sign is carried out by the slower machines and the full sign time is also determined by the speed of the slower machines, making the two times comparable. Since the proportion of fast to slow machines is the same for each f, the signing times for different values of f are comparable.

When there are no failed nodes, the quick sign phase of the hybrid signing scheme finishes successfully. We run 10 trials each with 100 signatures. We give the statistics for the times measured in Table 5.3. We find that the average time to finish one signature if the quick sign phase succeed is less than 200 msec for $f \leq 3$. Since each of the $f+1$ chosen nodes adds its shares together (a negligible calculation in terms of computation time) and performs one exponentiation, the quick sign times are similar for the different values of f.

When there are f failures, with at least one of the faulty nodes being one of the $f+1$ nodes chosen for a quick sign, the quick sign does not complete successfully. We

f	Ave Trial Time	Std Dev	Ave Time per Signature
1	16,053	12	161
2	17,562	349	176
3	17,645	85	176

Table 5.3: Signing Times (in msec) without Failures

f	Ave Trial Time	Std Dev	Ave Time per Signature
1	53,893	1,992	5,389
2	72,903	1,259	7,290
3	131,750	1,697	13,175

Table 5.4: Signing Times (in msec) with f Failures

run 10 trials each with 10 signatures. In our trials, the behavior of a faulty node is that it does not reply to sign requests. Nodes that do not reply cause the maximum delay in completing the signing protocol because the requester has to wait for a set amount of time before starting the full signing protocol. For our trials, the requester waits 5 seconds for quick sign replies, then initiates the full signing protocol if the quick sign did not succeed during those 5 seconds. When full signing is triggered, the amount of time it takes to finish a signing protocol increases significantly with the value of f. We give the statistics for signing times with failures in Table 5.4.

We give a summary of what the signing times should be measuring, keeping in mind that the network round-trip times (RTT's) for our experiments are very small since all of our nodes are in a LAN.

For a quick sign, the amount of time it takes to complete the protocol consists of:

1 RTT to establish TCP connections

\+ 1 RTT to send requests and receive replies

\+ Time for the each node to add together responsible shares and generate 1 signature

\+ Time for requester to put signatures together

f	l'	Time Spent Generating Signatures
1	3	243.12
2	15	1,216
3	84	6,807
4	495	40,114

Table 5.5: Amount of time each replier spends computing partial signatures in a full signing protocol, assuming each partial signature takes 385.75 ms to compute.

For the hybrid signing scheme, the worse case time consists of:

1 RTT to establish TCP connections

\+ Amount of time spent waiting for quick sign replies (that, in the worse case, never came)

\+ 1 RTT to send requests and receive replies

\+ Time for the fastest $2f+1$ correct nodes to each compute one partial signature for each share they have (*)

\+ Time for requester to put signatures together

To determine how much work the repliers in a full signing protocol are doing in the step marked by (*) above, we run some experiments to see how long it takes to produce one partial RSA signature with one subshare. Recall that our subshares are random numbers. Thus, we generate 100 random numbers in the same size range as our subshares. Then, we run 10 trial, each generating a partial RSA signature for each of the 100 random numbers we generated. We find that the average trial time is 8,104 msecs and the standard deviation is 2.51. Thus, the average time per partial signature is 81.04 msec. We show what the computation time for each receiver node is based on the number of shares each node has for the different values of f in Table 5.5. We find that most of the time for a full signing protocol is accounted for by the 5 seconds spent waiting for quick sign replies and the time the receivers spend in calculating partial signatures for the shares they hold.

f	Old Node	New Node
1	19	12
2	309	20
3	11,730	2,880

Table 5.6: Kilobytes sent by TSPSS old nodes and new nodes during the refreshment protocol

5.2 Secret Refreshing

5.2.1 Network Bandwidth Used

For an idea of how much network bandwidth the refresh protocol uses, we need to account for both the data sent by TSPSS nodes and the data sent by the BFT service. For the bandwidth used by TSPSS nodes, we give the total number of bytes TSPSS nodes send via TCP in Table 5.6. These numbers do not include the TCP overhead and they also do not include retransmitted bytes. For the bandwidth used by the BFT service, we use the model that the amount of bandwidth used per operation is $(3f + 1) \times$ (request size) + reply size. This model is reasonable when the request or reply size is much larger than the BFT protocol overhead, which is true in our system. For a more precise evaluation of BFT costs, see [1]. One artifact of the BFT library we are using is that the BFT request and reply sizes are limited to 4KB. Thus, to read or write more than 4KB to the BFT service requires multiple requests. Breaking request and replies into smaller messages does not significantly increase the number of bytes sent by the BFT so we ignore the 4KB limitation for the purposes of determining the network bandwidth used.

The BFT service supports the following operations: `write_proofs`, `node_ok`, `read_proofs`, `read_chosen`, and `read_new_witnesses`. `write_proofs` and `node_ok` are write requests. The reply for each is just a confirmation that the operation has occurred, thus the reply message for each is 8 bytes. The request size of `write_proofs` is the number of proofs (i.e., $l \times l'$) multiplied by the size of each proof (i.e., 128 bytes because that is the size of each RSA signature). The request size of `node_ok` is 12 bytes. The other operations are read requests, each with request sizes of 12

f	Minimum	Maximum
1	24	47
2	1,468	3,413
3	87,851	219,513

Table 5.7: Minimum and maximum amount of bandwidth used(in KB) by the BFT Agreement service, using the model that the number of bytes sent per operation is $(3f+1)\times$ (request size)+ reply size. Note that these numbers represent the bandwidth used by *all* BFT nodes, instead of *per* node

bytes. The reply size of `read_proofs` is the number of proofs multiplied by the size of each proof. The reply size of `read_chosen` is $(3f + 1) \times 4$ bytes. The reply size of `read_new_witnesses` is l multiplied by the size of each witness (i.e., 128 bytes again because that is the size of each RSA signature).

For each of the operations, there is a minimum number of times an operation must be invoked before the protocol can succeed. If all nodes in the system were correct then the maximum number of times that each operation can be invoked is correlated to the number of nodes. A minimum of $f + 1$ old nodes must write their proofs for the refresh protocol to complete. A maximum of $3f + 1$ correct old nodes can write their proofs. A minimum of $2f + 1$ new nodes must read proofs and confirm that a node's subshares are good for a minimum of $f+1$ old nodes. The maximum is $3f+1$ new nodes for $2f + 1$ old nodes. A minimum of $2f + 1$ and a maximum of $3f + 1$ new nodes read which nodes' subshares have been chosen and the witnesses for the new shares. We give the approximate bandwidth used in Table 5.7.

We see from Tables 5.6 and 5.7 that the amount of network bandwidth used for $f = 1$ is very small. The amount of network bandwidth used for $f = 2$ is arguably reasonable. The amount network bandwidth used for $f = 3$ is enormous.

5.2.2 Speed of Protocol

For the refresh protocol, we ran an instance of an old node and a new node on each machine. We use one Type I machine for each of the first $f + 1$ old and new nodes and one Type II machine for each of the next $2f$ old and new nodes. This setup

makes the refresh times as f increases comparable because one condition for finishing the refresh protocol is that at least $f+1$ correct old nodes must have generated and sent subshares and proofs.

We measured local refresh times for $f = 1$ and $f = 2$. We calculate lower bound refresh times for $f = 3$ and $f = 4$. A local refresh time is measured from the time a node initiates run v_{i+1} locally to the time run v_{i+1} terminates at that node. For an old node, a run is initiated when it receives a refresh header from the TSPSS application. The run at that old node terminates when the old node has received *computed* message from $2f + 1$ or more new nodes. A computed message is sent by each new node once it has computed its new shares and witnesses. For a new node, a run is initiated when it has first received and verified a refresh header from an old node. The run terminates locally at that new node when it has received computed messages from $2f$ other new nodes.

To avoid the situation where the last f new nodes must always request their shares from other nodes because they did not have time to finish computing their shares, our implementation lets new nodes finish verifying the chosen subshares that they received and computing their shares of the secret before the node considers the run to be terminated. The advantage of this approach is that the last f new nodes do not have to request shares when they already have the subshares for computing those new shares. The disadvantage is that a few slow nodes can lengthen the window of vulnerability.

One way to prevent a correct but very slow node from unacceptably lengthening the window of vulnerability is for all nodes to set a timer when they receive computed messages from at least $2f + 1$ new nodes. If they do not have all of their new shares by the time the timer expires, then they will calculate the new witnesses and end the run, thereby forgetting all the subshares they received. Then, they request their shares from the $2f+1$ new nodes who sent computed messages. At least $f+1$ of those nodes will be correct nodes, and will send the correct shares, which can be verified against the public witness.

We have not included timers in our implementation. In our experiments, the

absence of a timer that forces slow nodes to end their run can be viewed as a timer that does not expire before the new nodes finish computing their shares.

We took the refresh times for each node over five runs. Since the start and termination conditions for new and old nodes differ, we plot them in different graphs. See Figures 5-1 and 5-2. The graphs indicate the maximum, minimum, and median times that nodes take to finish in each run. For the old nodes, some nodes have smaller refresh times because the new nodes will start the refresh protocol as soon as they receive one refresh header with a valid threshold signature. We start the old nodes by ssh-ing into the work station to start the refresh protocol. Thus, some old nodes start later than others and thus have a shorter refresh time. The average refresh time for $f = 1$ is 14 seconds for old nodes and 16 seconds for new nodes. The average refresh time for $f = 2$ is 280 seconds for old nodes and 288 seconds for new nodes.

We give a summary of what the refresh times should be measuring, keeping in mind that the network round trip and transfer times for our experiments are very small since all of our nodes are in a LAN. Recall that for the implementation of the BFT agreement service we are using, each BFT request and reply must be smaller than 4KB. Thus, to read or write more than 4KB to the BFT service requires multiple requests.

The amount of time it takes to finish a run of the refresh protocol consists of:

Time for work done by old nodes (working in parallel):

1 RTT to establish the TCP connections

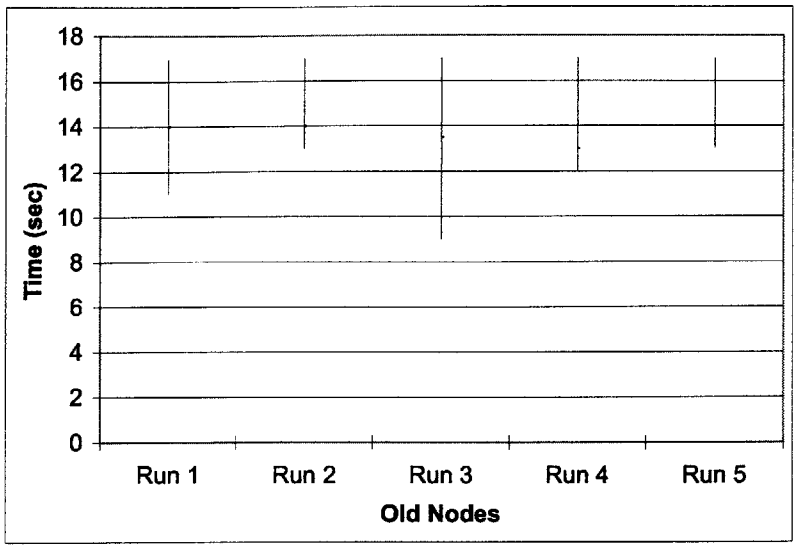
\+ 1 RTT to send the refresh headers

\+ Time it takes a node to compute the proofs

\+ 2 RTT $\times$ number of writes required for writing proofs to BFT

\+ Time it takes a node to encrypt the messages for each of the other nodes

\+ 1 RTT to send the encrypted subshares



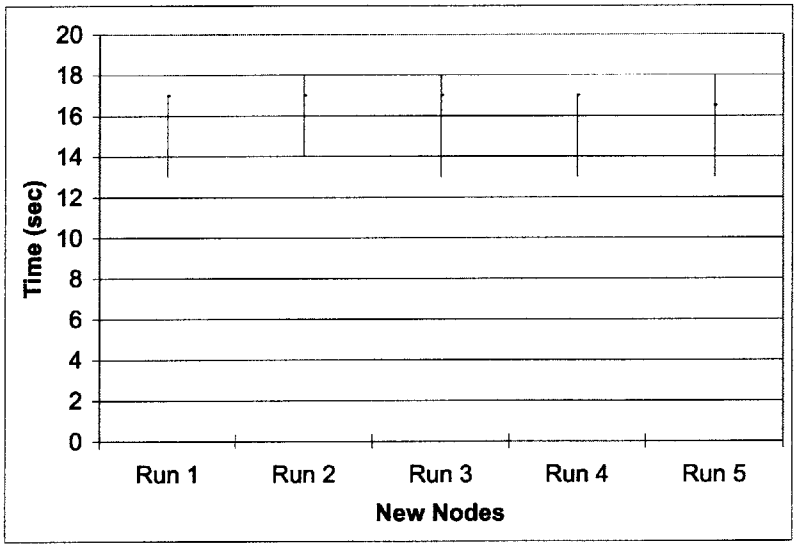
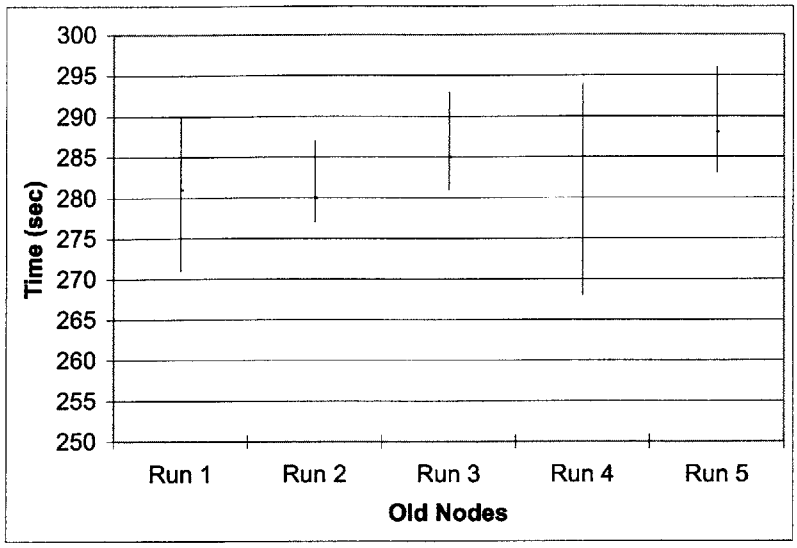


Figure 5-1: Refresh times for 5 runs with $f = 1$. This is a max, min, and median plot.



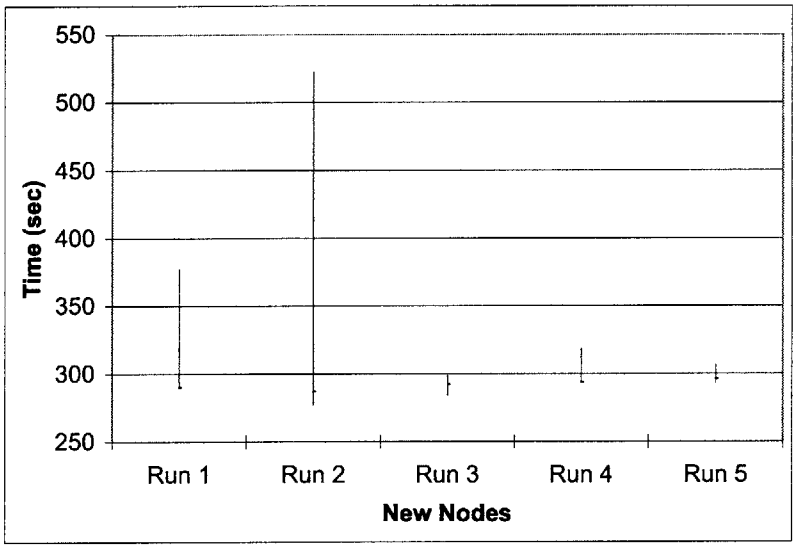


Figure 5-2: Refresh times for 5 runs with $f = 2$

Time for work done by a new node, for each of $f + 1$ old nodes:

1 RTT $\times$ number of reads to get proofs from BFT

$+$ Time to decrypt subshares

$+$ Time to check proofs

$+$ 1 RTT to notify the BFT that the sender's subshares are ok

Time for computing the new shares, once the subshares have been chosen:

1 RTT $\times$ number of reads to get which subshares are chosen and the witnesses for the new shares[1] from the BFT service.

$+$ Time it takes to compute the new shares and witnesses

$+$ 1 RTT to establish TCP connections, then send computed messages

A new node can start its work for an old node only after that old node has finished. The new shares can be computed only after a set of subshares has been chosen for each of the old nodes.

5.2.3 Extrapolated Lower Bounds

It takes too long and is too computationally expensive to run the refresh protocol multiple times for $f = 3$. We ran the refreshment for $f = 3$ once on workstations with some load and it took 3.37 hours to complete. Since we are trying to determine the maximum f for which TSPSS is practical, and the $f = 3$ case takes so long, we give a lower bound for refresh times based on the amount of time that computation alone requires. Specifically, we determine the amount of time required to generate proofs, decrypt the subshares, and check the proofs for the number of subshares the protocol calls for. Since we use the Rabin scheme for asymmetric key encryption and decryption, the encryption computations (i.e., squaring the subshares) take a

[1]The TSPSS code could have stored all the proofs and calculated the new witnesses from some of these proofs. However, since a minority of the proofs generated will ultimately be used to calculate the new witnesses, it makes sense to discard the proofs after checking the subshares and to retrieve the new witnesses from the BFT, which stores all the proofs, once the subshares for the new shares have been chosen.

Calculation Type	Ave Trial Time	Std Dev	Ave Time per Subshare
Decrypt	6,052	6.86	12.10
Proof Gen & Check	10,445	18.75	20.98

Table 5.8: Times for decrypting and proof generating and checking for Type I machines

Calculation Type	Ave Trial Time	Std Dev	Ave Time per Subshare
Decrypting	6,052	6.86	12.10
Proof Chekcing	10,445	18.75	20.98

Table 5.9: Times for decrypting and proof generating and checking for Type II machines

negligible amount of time. Checking a proof involves generating the proof from the received subshare and checking whether it equals the public proof for that subshare.

For Type I machines, we run 10 trials, each performing 500 decryptions and 500 proof generation and checking. For Type II machines, we run 10 trials, each performing 100 decryptions and 100 proof generations. We give the results in Tables 5.8 and 5.9.

The times for Type I and Type II machines are consistent with the machine specifications. Type I machines have CPU's that are five times faster than the Type II machine CPU's. For decryption and proof generation, Type I machines are a little less than five times faster.

We break the computation time down into i) the time taken by old nodes to generate proofs and ii) the time taken by new nodes to decrypt and verify those proofs for $f+1$ old nodes. Recall that $2f+1$ new nodes need to verify $f+1$ old nodes' subshares and that we have $f+1$ Type I machines each running one old node and one new node. We expect that the $f+1$ old nodes running on Type I machines will generate proofs and send subshares first, and some new nodes running on Type II machines are needed to check the subshares. Thus, the computation time for part (i) is the Type I proof generation time per subshare multiplied by l subshares for each of l' shares. The computation time for part (ii) is the Type II decrypt and proof generation time per subshare multiplied by l' subshares for each of l' shares from

f	l	l'	Lower Bound with Type I & II
1	4	3	3
2	21	15	112
3	120	84	4,629 (1.3 hours)
4	715	495	199,172 (2.3 days)

Table 5.10: Extrapolated lower bound refresh times (in sec) based on the time it takes for old nodes to generate proofs and for new nodes to decrypt and check proofs, using $f + 1$ Type I machines and $2f$ Type II machines.

f	Lower Bound with Type I
1	1
2	29
3	1,141 (0.5 hour)
4	47,817 (13.3 hours)

Table 5.11: Extrapolated lower bound refresh times (in sec) based on the time it takes for old nodes to generate proofs and for new nodes to decrypt and check proofs, using all Type I machines.

$f + 1$ old nodes. Part (i) and part (ii) times do not overlap and the estimated total computation time is:

($l \times l' \times$ Type I proof generation time per subshare)

$+ (l' \times l' \times (f + 1) \times$ Type II decryption and proof generation time per subshare).

Table 5.10 gives lower bound refresh times, based only on encryption, proof generation, and proof checking with the encryption and proof generation numbers given in Tables 5.8 and 5.9. These times *are* smaller than the refresh times we measured.

We find that using $f + 1$ Type I machines and $2f$ Type II machines, even the refresh time based only on encryption and proof generation times is unacceptably long for $f > 2$. For an idea of how much better the times would be with all Type I machines, we calculate such a lower bound give them in Table 5.11.

To determine whether a better implementation can complete the protocol within an acceptable amount of time, we analyze the optimistic scenario where we assume all nodes are correct. The optimistic scenario is that all nodes are correct and thus

f	Optimistic Lower Bound with Type I
1	.5
2	12
3	363 (6 min)
4	12,499 (3.5 hours)

Table 5.12: Extrapolated lower bound refresh times (in sec) based on the optimistic assumption that all nodes are correct and the time it takes for old nodes to generate proofs and for new nodes to decrypt and check proofs, using all Type I machines.

redundancy in the system is not needed. If all nodes were correct, we would need $f+1$ old nodes to generate, encrypt, and send subshares, with only one node generating subshares for each of share. Then $2f+1$ new nodes would need to decrypt and check the proofs for all of the shares.

Thus, in the optimistic scenario, the part (i) computation time is the Type I proof generation time per subshare multiplied by l subshares for each of $l/(3f+1)$ shares. The part (ii) computation time is the Type II decrypt and proof generation time per subshare multiplied by l' subshares for each of l shares. I.e., $(l \times l/(3f+1)\times$ Type I proof generation time per subshare) + $(l' \times l\times$ Type II decryption and proof generation time per subshare). We calculate and give the results for the optimistic scenario, using all Type I machines in Table 5.12.

Although the times do improve dramatically if we make optimistic assumptions, the these times are still unacceptably long for $f \geq 3$. We conclude that the exponential (with respect to f) number of times that expensive operations like exponentiation and decryption must be performed prohibits this scheme from being practical even for small values of f like 3.

Chapter 6

Conclusion

This thesis describes TSPSS, a *threshold signing* and *proactive secret sharing* protocol to address the authentication needs of reconfigurable Byzantine fault tolerant services such as Rosebud [12] and Pond [11]. We based the proactive secret sharing part of TSPSS on APSS [15], improving on the APSS work in two aspects: 1) Our refresh protocol works across 2 sets of servers, with one set of nodes transferring their collective knowledge of the secret to another set of servers that previously did not know about the secret. 2) Where agreement is needed in the protocol, our refresh protocol uses the Castro and Liskov BFT replication algorithm to achieve agreement.

Both TSPSS and APSS use combinatorial secret sharing which requires an exponential number of shares in f. The amount of computation time and bandwidth required to carry out the sign and refresh protocols corresponds to the number shares in the system.

The redundancy in the combinatorial secret sharing construction is useful. The property that any $f+1$ nodes have all the shares is useful for recovering shares because if $2f+1$ new nodes have received and verified their subshares, then we know that at least $f+1$ of those nodes are good and at least one correct node has each new share. If there are correct nodes that do not have their new shares, they can recover their shares from the $2f+1$ nodes. Since at least $f+1$ of those nodes are correct, the new node will receive at least one good copy for each of the shares it should have and it can check to see if a share is good using the corresponding witness.

The property that $2f + 1$ nodes hold each share is useful for signing because it allows us to use the full signing protocol instead of having to repeat the quick sign scheme with different groups of $f+1$ nodes until one succeeds. The number of possible groups of $f + 1$ nodes when choosing from $3f + 1$ nodes is exponential in f. Thus, many groups may need to be tried before a successful signature is generated.

However, the price of these useful properties is the exponentially increasing cost of TSPSS. TSPSS performs well for $f = 1$. Its performance is arguable reasonable for $f = 2$. It takes too much computation power and too much network bandwidth for $f \geq 3$, which translates into requiring too much time before the protocol completes. We find that the expensive operations TSPSS requires are the RSA partial signature generation during threshold signing and decryption using the Rabin asymmetric key cryptosystem and the exponentiations required for generating proofs for the subshares during secret refreshing. Inherent to proactive secret sharing using combinatorial secret sharing is that the number of shares is exponential in f. The number of RSA partial signatures that must be generated is directly related to the number of shares in the system. The number of encryptions and the number of proof generations is proportional to the number of shares squared. Thus TSPSS requires and exponentially increasing number of expensive operations to be performed and an exponentially increasing amount of bandwidth as f increase.

Assuming that Moore's Law continues to hold, we can see that for $f \leq 3$, refresh times for TSPSS will decrease to the point of being acceptable in the next ten years. However, a better solution is needed because ten years is a long time and some systems need $f > 3$.

This thesis has described a threshold signing and proactive secret sharing protocol intended for use by Byzantine fault tolerant services. We implemented and evaluated the cost of the protocol described. We identified the expensive operations in the protocol and quantified the practical costs of a secret sharing scheme with an exponential number of shares in f.

Bibliography

[1] Miguel Castro. *Practical Byzantine Fault Tolerance.* Ph.D., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation.* USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[3] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3), August 2003.

[4] P. Feldman. A practical scheme for non-interactive verifiable secret sharing. In *Proc. 28th IEEE Symp. on Foundations of Comp. Science*, pages 427–438. IEEE, 1987.

[5] The gnu mp library. http://www.gnu.org/software/gmp/manual/html_node/index.html#Top.

[6] Amir Herzberg, Stanislaw Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352. Springer-Verlag, 1995.

[7] M. Ito, A. Saito, and T. Nishizeki. Secret sharing scheme realizing general access structure. In *IEEE Globecom*, pages 99–102. IEEE, 1987.

[8] Stanislaw Jarecki. Proactive secret sharing and public key cryptosystems. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, September 1993.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21 (7), pages 558–565, July 1978.

[10] David Mazires. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, pages 261–274, June 2001.

[11] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, 2003.

[12] Rodrigues Rodrigo and Liskov Barbara. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical report, MIT CSAIL, 2003.

[13] Fred Schneider. Implementing fault-tolerant services using the state machin approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[14] Adi Shamir. How to share a secret. *Commun. ACM*, 22 (11):612–613, 1979.

[15] Lidong Zhou, Fred B. Schneider, and Robbert Van Renesse. Apss: Proactive secret sharing in asynchronous systems. Talk at Fanstord University (this is a full UNPUBLISHED entry), October 2002.