A secure bulletin board

Peters, R.A.

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

A SECURE BULLETIN BOARD

by
R.A. Peters

Supervisor: Dr. Ir. L.A.M. Schoenmakers

Eindhoven, June 2005

Abstract

In this thesis we present the design of a secure bulletin board, which can be viewed as a public broadcast channel with memory. Such a bulletin board is implemented by a distributed protocol executed between several parties. Users can post messages to this board, and once they have received a signed acknowledgement, they have the assurance that their message will never be deleted, will never be changed, and will be available to every other user. Also, no authorized user can be denied access to posting and reading messages. These properties hold even when up to one-third of the parties comprising the bulletin board are corrupted.

Contents

1 Introduction

Real-world election systems have attracted much attention over the last years. The current voting systems have many problems ranging from the results of elections, and therefore democracy, depending on the honesty of the select few people that construct voting machines, to outright fraud with pre-filled ballot containers. Concrete problems include voting results that were only available many weeks after the closing of the election in India, and misleading voting ballots in the US. In many countries, the integrity of the election is questioned by the opposition. Much effort is put into making election systems to be better verifiable.

Such election systems are still quite vulnerable, though. It is very difficult to prevent the corruption of any party. The undetected corruption of a single party has great consequences. The outcome of the election can be influenced, and the privacy of the voters can be violated. What we need is an election scheme, where the corruption of a single or a few parties do not endanger the security aspects of election schemes.

With the combination of the internet and the availability of well-studied cryptography, we have both an infrastructure and the methods to design an election scheme that solves many of the problems that exist with classical voting systems. The goal is to present an election scheme that not only produces correct results even when voters try to disturb the process, but even tolerates malicious talliers. An example of such an election scheme is presented in [CGS97]. This scheme provides privacy and universal verifiability, even when a number of talliers are corrupted. This scheme, however, assumes the existence of a public broadcast channel with memory, a bulletin board. This thesis describes the design and implementation of such a bulletin board, and discusses various aspects ranging from theoretical issues such as resilience to various attacks, to practical issues such as efficiency and implementation considerations.

A secure bulletin board has similar applications in other domains, as well. For example, secure auction schemes could benefit from a secure bulletin board. In fact, many cryptographic protocols, such as Pedersen's verifiable threshold secret sharing scheme, assume the existence of a broadcast channel. The bulletin board presented in this thesis can be used in these protocols.

In this thesis, we study various protocols that are suitable for use in a secure bulletin board, but we explicitly note that the proofs and even descriptions of those protocols are not the focus of this thesis. In examining the protocols, we do try to give a general idea how the protocol works, but the primary intention is to use an existing protocol as the basis in our bulletin board.

2 Case Study: A Multi-Authority Election Scheme

Since the motivation of the construction of the bulletin board presented in this thesis is to complete the scheme presented in [CGS97], we will first examine the properties of that scheme, without going into too much detail. Other applications in which a bulletin board is used probably need similar functionality.

The scheme of [CGS97] works in the model set forth by Benaloh *et al.* [CF85, BY86, Ben87]. In this model, the active parties are divided into l voters $V_1, \ldots, V_l$ and n talliers $A_1, \ldots, A_n$. The election basically consists of two phases: In the first phase, each voter casts his vote by encrypting it and posting the vote to the bulletin board. In the second phase, the talliers execute a multi-party protocol to jointly compute the results of the election, and publish the result to the bulletin board.

This scheme tolerates the benign and malign failure of up to a threshold t talliers, $1 \leq t \leq n$. This means that even when t talliers cooperate, they cannot change the result of the election. This is a very important property, because the outcome of the election does not depend on each tallier being honest, unlike in classical elections, where for example a single pre-filled ballot container can influence the results.

It is assumed that the talliers have jointly generated a public key and shared private keys for a threshold cryptosystem. The encryption key is assumed to be distributed to the voters in a secure manner, and voters will use this key to encrypt their ballots. Since a threshold cryptosystem is used, only sufficiently large groups of talliers will be able to decrypt a ballot. An honest tallier will not participate in the decryption of a single voter's ballot, since that would compromise the voter's privacy. If sufficient talliers are honest, the privacy of voters can be assured.

To cast his vote, the voter encrypts his ballot and posts it to the bulletin board. In order to guarantee privacy, the contents of this encrypted vote may not be linked to the voter. There are two ways to provide this: the first is to use homomorphic encryption schemes to obtain the encryption of the sum of all votes, and only decrypt this sum using a threshold decryption protocol. This requires the voter to present a proof that his encrypted ballot does indeed contain a valid vote. This approach is proposed in [CGS97]. Another method to provide ballot secrecy is with the method of *Mixing*. This method works by taking the set of all encrypted votes, and give it to te first *Mixer*. This mixer permutes the order of the votes. The votes are also blinded, so that the permutation used is not visible, and a proof is presented that the input set consists of the same votes as the output set. The output of the first mixer is the input to the second mixer, who also mixes the votes. This process is repeated by each mixer. As long as there are at least one honest mixer that does not reveal the permutation used, no one will be able to link a vote to its voter. After this mixing, each vote is decrypted using a threshold decryption protocol, and the election result can easily be calculated from these decrypted votes.

3 Specifications of the Bulletin Board

We are going to specify and build a prototype of a bulletin board, which is implemented by a distributed protocol executed between several parties. This bulletin board can be used as the primitive needed in the election scheme, or it can serve other purposes, like a secure broadcast channel needed in protocols such as Pedersen's verifiable secret sharing scheme.

3.1 Functionality

The problem the bulletin board actually has to solve is a *consensus* or *Byzantine agreement* problem. Byzantine agreement is a classical problem introduced in [PSL80] and [LSP82]. All parties must reach consensus on the contents of a particular message, while some parties may be corrupted by an adversary. The protocol used must be resilient to *Byzantine failures*: a corrupt party can behave in any arbitrary way, even conspire together with other corrupt parties in an attempt to make the protocol work incorrectly. The identity of corrupt parties is unknown, reflecting the fact that faults can happen unpredictably.

We informally state the functionality provided by the bulletin board in this section.

Availability Each party is able to successfully send messages to the bulletin board.

Broadcast If some party sends a message to the bulletin board, every party receives that message.

Agreement Once a message has been sent successfully to the bulletin board, every party receives the same message.

Memory Once a message has been sent successfully to the bulletin board, the message is not deleted and is available to each party.

Our bulletin board will run as a distributed protocol between several parties. We do not want to assume that every party does exactly what the specification prescribes, because we want to take into account that some adversary controls some of the parties.

3.2 Types of Parties

Each party may follow the rules of our specifications, or it may deviate from them. To categorize the behaviour of the parties, we introduce a few definitions. A party is either *honest* or *corrupt*. Honest parties follow the specifications of the protocol. Corrupt parties may deviate in any way from the protocol, including cooperating with other corrupt parties. This is also known as *Byzantine* failure. In the *static* adversary model, the adversary selects a number of parties and corrupts them, after which exactly these parties are corrupt, and the other parties are honest. In the *dynamic* adversary model, the adversary may observe the protocol as it runs, and select parties to corrupt at will.

In addition to being honest, we introduce definitions relating to the reachability and responsiveness of parties. A party is either *correct* or *faulty*. Correct parties always respond correctly within some timeout, while faulty parties either respond too late, not at all, or incorrectly. If a party has been cut off the network, or has crashed, it is faulty. By definition, corrupt parties are always faulty, and correct parties are always honest.

3.3 Bounds on the Number of Faulty Parties

If every party is corrupt, then we cannot expect our bulletin board to work properly, of course. The bulletin board must however be tolerant of at least a few faulty parties. Each protocol implementing the functionality of a secure bulletin board has some bound on the number of parties it tolerates to be faulty, which is called the *resilience*. For example, Phalanx [MR98] has a resilience of $\lfloor \frac{n-1}{4} \rfloor$. Another bound, which is the resilience of Rampart [Rei94], is $\lfloor \frac{n-1}{3} \rfloor$. A bulletin board can be used to solve the so-called *agreement* problem, and a well-known theoretical result says that every protocol solving the agreement problem has a resilience of at most $\lfloor \frac{n-1}{3} \rfloor$, hence, the bound for Rampart is optimal.

3.4 Typical use of Bulletin Board

Using the election scheme as the example application for our bulletin board, we identify the required functionality. There are four phases.

1. During the first phase of the election, the bulletin board is initialized and its keys are generated and distributed to the voters.
2. During the second phase of the election, voters must be able to post their vote to the bulletin board. After posting, they must have the assurance that their vote is not altered or removed, and that the vote will be counted in the final tally (of course, under the assumption that no more parties are faulty than tolerated by the protocol).
3. During the third phase of the election, the first mixer must be able to contact the bulletin board, and read all the votes cast. Then, that mixer must be able to write back the result of mixing, and the proof that the result is indeed a permutation of the original set of votes. Subsequently, the second mixer reads the results of the first mixer, and writes back the next permutation of the votes. This process is repeated for each mixer.
4. During the fourth phase of the election, the talliers jointly decrypt each vote. This operation consists of reading all votes, and writing a decryption share back to the bulletin board. From these decryption shares, everyone can reconstruct the original vote, and compute the final result.

Distribution of each party's public key is beyond the scope of this project, so we just assume that the distribution in the first phase is done correctly, e.g. with a PKI. The second phase is correct by design, if the condition on the number of correct parties is met. The mixing results of the second phase must be verified. This verification must be done by at least one honest party, which means that at least one third of the parties must verify the correctness, if the protocol tolerates at most one third minus one faulty parties. The results produced in the third phase must be verified in a similar manner.

3.5 Assumptions of the Bulletin Board

We like to assume as little as possible about the information available to the parties when building the bulletin board. Specifically, there should be no trusted third party, a single party on whose honesty the bulletin board depends.

Communication with or between parties should be done securely. Secure channels are easily set up when each party has its own public/private keypair. Since no trusted third party is available to generate this keypair, each party must generate its own. After generation, some method must exist to securely publish the public keys, so that everyone involved in the election has access to each public key. The method used to distribute the public keys is beyond the scope of this project, it is assumed that such a method exists, e.g. a PKI is used.

While it is possible to use a simple threshold signature scheme using only the public key of each party, a speedup in performance can be gained by using a better threshold signature scheme, as will be explained in a later chapter. Such a scheme needs a separate threshold key generation protocol, though, and the public key produced by the protocol must be published as well. We therefore assume that a threshold key generation protocol is included in the bulletin board. This protocol will probably be run some time before the actual election is held, so that the public key can be distributed to the voters.

4 Candidate Bulletin Board Protocols

Now that we have a clear idea of what our bulletin board must be capable of, we can study various existing protocols and decide what protocol we will use as the core of our bulletin board. The protocols that we examine are Rampart [Rei94], a protocol described by Kursawe and Shoup [KS01], Phalanx [MR98], and another version of Phalanx [MR97] which we will denote by PhalanxII. The Rampart and Phalanx protocols will be adapted so that they use threshold signatures in order to gain performance. Therefore, we make a comparison of six different protocols: Rampart, Kursawe-Shoup, Phalanx, PhalanxII, Rampart+, and Phalanx+, where the + indicates the adapted version.

An informal description of each protocol discussed in this chapter is given, but this description is just enough to examine the complexity of the protocols. It is not the intention to give a precise description of each protocol, the designs of the protocols are beyond the scope of this project. After comparing the complexity of the various protocols presented in this chapter, we can select one to use in our bulletin board. Once this choice has been made, the selected protocol will be discussed in greater detail.

4.1 The Bulletin Board Used in Elections

We will use the election scheme from our case study to determine the criteria to compare various protocols. In this election scheme, the bulletin board will be used to publish votes from the voters and additional data by the talliers.

We review the phases defined in section 3.4 and examine the requirements on the complexity of read and write operations:

1. The first phase is the setup phase, where the servers jointly generate public key pairs for a threshold signature scheme. The performance of this phase is not very important.

2. The second phase is the voting phase, during which clients can publish their votes on the bulletin board. These votes are small and are placed on the board one at a time.

3. During the third phase, mixers read the whole contents of the board, permute the contents and write back the contents to the board. These read and write operations are performed only a small number of times, but since it covers the whole contents of the bulletin board, the data read and written is very large.

4. After the mixers have mixed the votes sufficiently many times, the talliers jointly decrypt the votes in the fourth phase. These talliers also use large read and write operations to process large amounts of votes at once. After writing the decrypted votes back to the board, the votes can be tallied.

In conclusion, the protocol must support fast writes of small messages, and reads and writes of many small messages at the same time. Reading one small message does not occur often, if at all.

4.2 Comparison Criteria

In order to compare the protocols, several criteria need to be defined:

Resilience The maximal number of parties that may be corrupted. The best that can be achieved is $\lfloor \frac{n-1}{3} \rfloor$. If more parties than the resilience are corrupted, the security of the protocol is not guaranteed.

Round Complexity The number of changes in the direction of communication. This gives a rough indication of the latency of the protocol.

Message Complexity The number of messages sent over the network.

Communication Complexity The total length of the messages. This is a better indication of network traffic than message complexity, as very large messages take more time and bandwidth to be sent than small messages.

Computational Complexity The amount of computation needed. Only public key cryptosystem operations are taken into account, as the modular exponentiations will be the dominating factor of the needed computation time.

For each protocol, two typical runs will be examined, in which a client writes one message, and a client reads many messages. For simplicity, it is assumed that the client reads the first l messages in the read operation. These two operations are denoted with *Write One* and *Read Many*. In these descriptions, n will be the number of parties and l the number of messages read. Alternatively, Read Many may be seen as reading a very large message of size l.

In our comparisons, we only consider the case where every party is honest. Once a protocol is resilient against faulty behaviour of some of the parties, it is likely that no party even tries to disrupt the protocol, unless there are enough corrupt parties to compromise the integrity of the bulletin board, at which point the performance is not important anyway.

4.3 The Protocols

The pseudo-formalism used to describe the protocols is taken from [CKPS01]. Each message has a parameter describing the kind of message sent. If that parameter is an **in** or an **out**, then the message is not sent over the network, but instead passed on to a lower or upper layer in the protocol stack. In that case, the second parameter describes the kind of message sent. Other parameters contain payload. The protocol descriptions in this section are not as formal as the descriptions in later sections, because this section focuses on performance and complexity, not on correctness and completeness.

A protocol is described as a series of events and responses. Once a specific condition is triggered, the appropriate statements are executed, possibly triggering other conditions.

4.3.1 Rampart Protocol Description

The protocol used in Rampart implements a secure broadcast channel, and is composed of several layers: the Atomic multicast protocol, the Reliable multicast protocol and the Echo multicast protocol. The Echo multicast is the core protocol, which is used by honest servers to ensure that all other servers receive a certain message. The Reliable multicast protocol uses the Echo multicast protocol and ensures that the set of received messages is the same on each server, even if a message is sent by

a dishonest server. The Atomic multicast protocol uses the Reliable multicast protocol and assigns an order to the messages.

These three layers are built on a Secure Group Membership protocol, such as the one detailed in [Rei96]. With this protocol, each server maintains a view of parties V^x, which is the set of active and responsive parties in the broadcast channel. Once a party suspects another party of being faulty, it votes to remove the faulty party from the group, and once enough parties choose to remove some faulty party, the faulty party is removed. A new group view V^{x+1} is formed, which then becomes the current view.

Echo Multicast Protocol The interface of the echo multicast protocol consists of two processes: *E-mcast*(x, m) and *E-deliver*(p, x, m), which are used to multicast a message m in view x.

Let V^x denote the set of servers in the active view. When some party $p \in V^x$ wants to echo multicast a message m in view x, it sends (**in**, **e-mcast**, x, m) to the echo multicast protocol. When some party $p' \in V^x$ receives a multicast message m in view x from p, the echo multicast protocol sends (**out**, **e-deliver**, x, m) to a higher-level protocol.

This protocol is explained in greater detail in section 8.4.1, but a summary is given here: When (**in**, **e-mcast**, x, m) is received, party p starts by sending an **init** message containing the cryptographic hash of m, denoted by $f(m)$. Each party maintains an index l_p denoting the number of messages received by this party. The **init** message commits the hash of m to the l_p-th message from p. Each party then responds with an **echo** message, including a signature serving as proof that that party committed m to the l_p-th index. p then accumulates these **echo** messages, until it has $\lceil (2|V^x| + 1)/3 \rceil$ of them, and then sends a **commit** message, containing the signatures and m. Each party can now deliver m, after verifying that the order of delivery of messages from p is correct.

This protocol prevents a corrupt party from making honest parties deliver different messages on the same index. Since a party has to acquire $\lceil (2|V^x| + 1)/3 \rceil$ signatures from different parties, and since each honest party only creates one signature for a specific index and party, a corrupt party cannot acquire two sets of signatures for two different messages on the same index.

upon receiving a message (**in**, **e-mcast**, x, m)
send (**init**, x, $f(m)$) to each $p \in V^x$.

upon receiving a message (**init**, x, d) from $p \in V^x$
let l_p be the number of messages received from p.
send a signed message (**echo**, p, x, l_p, d) to p.

upon receiving messages $S = \{(\textbf{echo}, p, x, l, d) \text{ signed by } p_j\}_{p_j \in V^x}$, where $|P| = \lceil (2|V^x| + 1)/3 \rceil$
send (**commit**, p, x, m, S) to each $p \in V^x$.

upon receiving a message (**commit**, p, x, m, S) where $S = \{(\textbf{echo}, p, x, l, d)_K\}_{p_j \in V^x}$ and $|P| = \lceil (2|V^x| + 1)/3 \rceil$
if no view V^y is received with $y > x$ and $p \notin V^y$
add this message to $commits^x$.

upon adding a message (**commit**, p, …) to $commits^x$
while a message (**commit**, p_i, x, m, $\{(\textbf{echo}, p, x, l, d)_K\}_{p_j \in V^x}$) is previously received such that $c_i^x + 1 = l$ **do**
send (**out**, **e-deliver**, p_i, x, m) and set $c_i^x := c_i^x + 1$;.

Figure 1: Rampart Echo Multicast Protocol

Reliable Multicast Protocol The interface of the reliable multicast protocol consists of two processes: *R-mcast*(m) and *R-deliver*(p_i, m), which are used to multicast a message m. It uses the *E-mcast*(x, m) and *E-deliver*(p_i, x, m) processes of the echo multicast protocol.

This protocol is discussed in section 8.5.1

In the absence of membership changes, the reliable multicast protocol just relays messages between the echo multicast protocol and the atomic multicast protocol. Therefore, the protocol is not detailed here, but it will be in a later section.

Atomic Multicast Protocol The interface of the atomic multicast protocol consists of two processes: *A-mcast*(m) is used to send a message m to every server, and *A-deliver*(p_i, m) is executed on each server when it receives m.

Without going into detail, special messages are broadcast with the reliable broadcast protocol to assign an order to the messages. A single atomic multicast message m results in one reliable multicast message containing m, and one order packet. When n messages are atomically multicast in a short period, only one order packet is needed, resulting in $n + 1$ reliable multicast messages. This extra message does not impact complexity much, therefore it is not detailed here. The atomic multicast protocol is detailed in section **??**

Constructing a Bulletin Board The secure atomic broadcast protocol that Rampart provides can be used as the most important building block of our bulletin board.

When a client wants to write a message on the bulletin board, the server with which the client connected uses the broadcast protocol to broadcast the message and requests threshold signature shares. These shares are combined by the server into a single signature and returns that signature to the client.

To read a message, a client sends a nonce, which is broadcast by the server to construct a signature on the message and the challenge. The message is returned, together with the signature.

If a client wants to read the contents of the whole bulletin board, it only needs to receive the messages from one of the servers, in addition to a single signature of all the messages together. Since an atomic broadcast protocol is used, each server has exactly the same sequence of messages. Therefore, only one signature share needs to be transmitted from each server to the server to which the client is connected, and the bulletin board messages only need to be sent from one server to the client. This is all under the assumption that no writes occur during the read operation.

The client protocol:

upon receiving a message (**in**, **write**, m)
    repeat send (**write**, m) to some server $p \in P$
    until ready-write.

upon receiving a message (**write-ack**, m_K) where K is the public key of the bulletin board
    if the signature is correct **then**
        set ready-write.

upon receiving a message (**in**, **read**)
    repeat choose a challenge c and send (**read**, c) to some server $p \in P$
    until ready-read.

upon receiving a message (**read-ack**, m, $\langle m, c \rangle_K$) where K is the public key of the bulletin board
    if the signature is correct **then**
        set ready-read
        send (**out**, **read-ack**, m)

upon receiving a message (**in**, **read-all**, b, e)
    repeat choose a challenge c and send (**read-all**, c, b, e) to some server $p \in P$
    until ready-read-all.

upon receiving a message (**read-all-ack**, $\{m\}_{[b,e)}$, $\langle \{m\}_{[b,e)}, b, e, c \rangle_K$) where K is the public key of the bulletin board
    if the signature is correct **then**
        set ready-read-all
        send (**out**, **read-all-ack**, $\{m\}_{[b,e)}$)

Figure 2: Bulletin Board Protocol for Clients

The server protocol:

upon receiving a message (**write**, m)
    send (**in**, **a-mcast**, $\langle$**write**, m, c $\rangle$) where c is the client

upon receiving a message (**out**, **a-deliver**, $\langle$**write**, m $\rangle$) from p
    record m, c, i
    set $i := i + 1$ send (**write-ack**, m_K) to p where K is the threshold signature key of this party.

upon receiving messages (**write-ack**, m_{K_i}) from enough parties p_i
    combine the threshold signatures into one signature m_K.
    send (**write-ack**, m_K) to c.

upon receiving a message (**read**, c)
    send (**read-request**, c) to each $p \in P$.

upon receiving a message (**read-request**, c)
    send (**read-reply**, $\langle m, c \rangle_K$) to the sender, where K is the threshold signature key of this party, and m is the message recorded as written by client c.

upon receiving messages (**read-reply**, $\langle m, c \rangle_{K_i}$) from enough parties p_i
    combine the threshold signatures into one signature $\langle m, c \rangle_K$.
    send (**read-ack**, m, $\langle m, c \rangle_K$) to c.

upon receiving a message (**read-all**, c, b, e)
    send (**read-all-request**, c, b, e) to each $p \in P$.

upon receiving a message (**read-all-request**, c)
    send (**read-all-reply**, $\langle \{m\}, b, e, c \rangle_K$) to the sender, where K is the threshold signature key of this party, and $\{m\}$ are all messages.

upon receiving messages (**read-all-reply**, $\langle \{m\}, c \rangle_{K_i}$) from enough parties p_i
    combine the threshold signatures into one signature $\langle \{m\}, c \rangle_K$.
    send (**read-all-ack**, $\{m\}$, $\langle \{m\}, c \rangle_K$) to c.

Figure 3: Bulletin Board Protocol for Servers

4.3.2 Rampart Protocol Run

When examining the performance of Rampart, only the echo multicast protocol is of interest, since in a run with only honest parties, the commands in the other layers are performed in a constant amount of time, and the network traffic induced by the atomic and reliable multicast layers is only a fraction of the amount generated by the echo multicast protocol.

When a client c writes a message m to the bulletin board consisting of servers P, the following steps are executed:

1. c sends (**write**, m) to some server $p \in P$.
2. p sends (**init**, x, $f(m)$) to every $p' \in P$.
3. upon receipt, every server sends a signed echo message $(\textbf{echo}, p, x, l, d)_{K_j}$ to p.
4. after receiving the echo messages, p sends $(\textbf{commit}, p, x, m, \{(\textbf{echo}, p, x, l, d)_{K_j}\}_{p \in P' \subseteq P})$ to every process, where $|P'| = \left\lceil \frac{2n+1}{3} \right\rceil$.
5. after server p' receives the order message, it A-delivers m, computes a signature share and sends (**write-ack**, m_K) to p.
6. p combines the shares into a single signature and sends (**write-ack**, m_K) to c.

When a client c reads all l messages, the following steps are executed:

1. c chooses a nonce d and sends (**read-all**, d) to some server $p \in P$.
2. p sends (**read-all-request**, d) to each $p' \in P$.
3. each $p' \in P$ computes a signature share over the l messages and sends (**read-all-reply**, signature) to p.
4. p combines these shares into a single signature and sends (**read-all-ack**, messages, signature) to c.

Note that the broadcast protocol is not used in the read protocol, since the servers do not need to come to an agreement on something. Receiving $\left\lceil \frac{n-1}{3} \right\rceil + 1$ signature shares is enough to construct the threshold signature.

4.3.3 Analysis of Rampart

In this first complexity analysis, we shortly describe how we obtained the numbers. We omit this description in later analyses.

Resilience The resilience is given by the Rampart protocol itself.

$$\left\lfloor \frac{n-1}{3} \right\rfloor$$

Round Complexity In the example protocol run, we count the number of items, since each item represents a communication in a different direction than the previous item.

Write One	Read Many
7	4

Message Complexity Take the protocol run where a client writes a message. We observe that there are 3 protocol steps in which one message is sent, and there are 4 protocol steps in which n messages are sent. We neglect the former, so there are about $4n$ messages sent in the write protocol.

Write One	Read Many
$4n$	$2n$

Communication Complexity Of each protocol step, we note how big the messages are that are being sent, and to how many parties they are being sent. In step 4 of the write protocol, n messages are sent containing $\lceil (2n+1)/3 \rceil$ signatures, so it has a communication complexity of about $2n^2/3$, plus there are three steps in which a small message is sent to n parties.

Write One	Read Many
$3n + \frac{2n^2}{3}$	$2n + l$

Computational Complexity In step 2 of the write protocol, each server creates only one signature which is neglected, but in step 3, each server verifies $\lceil (2n+1)/3 \rceil$ signatures. The entry server also verifies the first $\lceil (2n+1)/3 \rceil$ signatures received in step 3.

	Write One	Read Many
client	1	1
server	$\lceil \frac{2n+1}{3} \rceil$	1
entry server	$\lceil \frac{4n+2}{3} \rceil$	$\lceil \frac{4n+2}{3} \rceil$

4.3.4 Rampart+ Protocol Description

The communication and computational complexity of Rampart is quadratic in the number of parties. This is due to step 4 of the write protocol, where one party sends n messages of length $O(n)$. With a threshold signature scheme, we can 'compress' those $O(n)$ signatures into one signature, thereby reducing the communication complexity to $O(n)$, and reducing the computational complexity of non-entry servers to $O(1)$.

We change the Rampart protocol such that a signature share, rather than a signature, is created when sending the **echo** message, and we change the event handler of the **commit** message to combine the signature shares into a single threshold signature:

upon receiving a message (**in**, **e-mcast**, x, m)
send (**init**, x, $f(m)$) to each $p \in V^x$.

upon receiving a message (**init**, x, d) from $p \in V^x$
let l_p be the number of messages received from p.
send a message (**echo**, p, x, l_p, d) with a signature share on it to p.

upon receiving messages $S = \{$(**echo**, p, x, l, d) signed by $p_j\}_{p_j \in V^x}$, where $|P| = \lceil (2|V^x| + 1)/3 \rceil$
combine the signature shares of the **echo** messages into a single signature s.
send (**commit**, p, x, m, s) to each $p \in V^x$.

upon receiving a message (**commit**, p, x, m, s) where $s =$(**echo**, p, x, l, $d)_K$ where K is the shared private key of the threshold signature scheme and $|P| = \lceil (2|V^x| + 1)/3 \rceil$
if no view V^y is received with $y > x$ and $p \notin V^y$
add this message to $commits^x$.

upon adding a message (**commit**, p, …) to $commits^x$
while a message (**commit**, p_i, x, m, (**echo**, p, x, l, $d)_K$) is previously received such that $c_i^x + 1 = l$ **do**
send (**out**, **e-deliver**, p_i, x, m) and set $c_i^x := c_i^x + 1$;.

Figure 4: Rampart+ Echo Multicast Protocol

4.3.5 Rampart+ Protocol Run

When a client c writes a message m to the bulletin board consisting of servers P, the following steps are executed:

1. c sends (**write**, m) to some server $p \in P$.
2. p sends (**init**, x, $f(m)$) to every $p' \in P$.
3. upon receipt, every server sends a echo message (**echo**, p, x, l, $d)_{K_j}$ with a threshold signature on the message to p.
4. after receiving the echo messages, p combines the threshold signature shares into one signature s and sends (**commit**, p, x, m, s) to every process, $|P'| = \left\lceil \frac{2n+1}{3} \right\rceil$.
5. after server p' receives the order message, it A-delivers m, computes a signature share and sends (**write-ack**, m_K) to p.
6. p combines the shares into a single signature and sends (**write-ack**, m_K) to c.

4.3.6 Analysis of Rampart+

As only the communication complexity and the computational complexity change, the other aspects are not mentioned here.

Communication Complexity

Write One	Read Many
$4n$	$2n + l$

Computational Complexity

	Write One	Read Many
client	1	1
server	2	1
entry server	$\lceil \frac{n-1}{3} \rceil$	$\lceil \frac{4n+2}{3} \rceil$

4.3.7 Kursawe-Shoup Protocol Description

Like the protocol used in Rampart, the protocol of Kursawe-Shoup [KS01] is an atomic broadcast protocol. A big difference between those two protocols, is that Rampart builds on a secure group membership protocol, while Kursawe-Shoup does not. The latter protocol has two phases: the *optimistic phase* and the *recovery procedure*. Instead of group views like Rampart has, it has epochs. Each epoch begins with an optimistic phase, in which messages are sent to the leader of the protocol, which starts broadcasting the message. When enough parties complain because of timeouts or errors, the recovery procedure is started to ensure that messages sent in the current epoch are delivered. After this procedure, the next epoch is started with another optimistic phase.

Because we examine the complexity of the protocol in the absence of faulty parties, we only describe the optimistic phase and examine its properties.

Optimistic Phase First, the optimistic phase is started. The phase ends when the **Recover** procedure is executed. After the **Recover** procedure terminates, this phase is started again.

When a server wants to broadcast a message, that message is first sent to the leader. The leader multicasts the message to every server with the **0-bind** message. Every server that receives the message in this way, also sends it to every server, with the **1-bind** message. Then, every server that receives this message, it sends ta **2-bind** message to every server, and upon receiving a **2-bind** message, the message is delivered.

When some server does not deliver a broadcast message before some timeout, it sends a complaint and stops responding to **0-bind** messages. When $t+1$ servers block this way, the other servers cannot make progress and eventually the Recovery procedure will be executed.

If the leader is faulty, and only sends the message to $n-t-1$ or less correct servers, then the message will not be broadcast successfully, because a **1-bind** message will never be generated. If the leader sends the message to at least $n-t$ servers which send a **2-bind** in response, then at least $t+1$ of those $n-t$ servers are honest and will send that **2-bind** to every server. Now every server receives the message at least $t+1$ times, so that every server that did not already send a **2-bind**, they will do it now. This results in every correct server sending a **2-bind** message, resulting in every server receiving at least $n-t$ **2-bind** messages, after which every server will deliver the message.

The protocol uses the following variables: e is the epoch number. It starts at zero and is incremented each time the Recovery procedure is executed. D is the set of the messages that have been delivered. I is the initiation queue, the queue of messages that the server has initiated, but not yet delivered. w is the window pointer, denoting the number of requests that have been delivered. $BIND_1$ and $BIND_2$ are the sets of messages received in a **1-bind** resp. **2-bind** message. *acnt* is the number of acknowledgements received for message deliveries. *complained* denotes whether this server has sent a complaint. $it(m)$ holds the value of w for each $m \in I$ at the point in time when m was added to

I. l is the leader index. SR is maintained by the leader and contains the set of messages which have been assigned sequence numbers. $scnt$ is maintained by the leader and contains the value of the next available sequence number.

upon receiving a message (**in**, **a-broadcast**, m)
  verify $m \notin I \cup D \wedge |I| < BufSize$
  Send (**initiate**, e, m) to the leader
  Add m to I
  Set $it(m) := w$

upon receiving a message (**initiate**, e, m)
  verify this process is the leader
  verify $w \leq scnt < w + WinSize \wedge m \notin D \cup SR$
  Send (**0-bind**, e, m, $scnt$) to all parties
  $scnt := scnt + 1$
  Add m to SR

upon receiving a message (**0-bind**, e, m, s)
  verify the sender is the leader
  verify $w \leq s < w + WinSize \wedge s \notin BIND_1 \wedge ((I = \emptyset) \vee (w \leq \min\{it(m) : m \in I\} | Thresh))$
  Send (**1-bind**, e, m, s) to all parties
  Add s to $BIND_1$

upon receiving $n - t$ message of the form (**1-bind**, e, m, s) from distinct parties that agree of s and m
  verify $w \leq s < w + WinSize \wedge s \notin BIND_2$
  Send (**2-bind**, e, m, s) to all parties
  Add s to $BIND_2$

upon receiving $t + 1$ message of the form (**2-bind**, e, m, s) from distinct parties that agree of s and m
  verify $w \leq s < w + WinSize \wedge s \notin BIND_2$
  Send (**2-bind**, e, m, s) to all parties
  Add s to $BIND_2$

upon receiving $n - t$ message of the form (**2-bind**, e, m, s) from distinct parties that agree of s and m
  verify $s = w \wedge acnt \geq |D \wedge m \notin D \wedge s \in BIND_2$
  Send (**out**, **a-deliver**, m)
  Add m to D
  Remove m from I
  stop timer

upon timer is not running and $complained$ is not set and $I \neq \emptyset$ and $acnt \geq |D|$
  start timer

upon timeout
  if $\neg complained$ **then**
    Send (**complain**, e) to all parties
    $complained :=$ **true**

upon receiving $t + 1$ messages of the form (**complain**, e) from distinct parties
  verify $\neg complained$
  Send (**complain**, e) to all parties
  $complained :=$ **true**
  stop timer

upon receiving $n - t$ messages of the form (**complain**, e) from distinct parties
  Execute the Recover procedure.

Figure 5: Optimistic Phase of Kursawe-Shoup

Constructing a Bulletin Board The protocol of Kursawe-Shoup is a broadcast protocol, just like Rampart, so a bulletin board can be constructed on top of the broadcast protocol with the protocol described in section 1.

4.3.8 Kursawe-Shoup Protocol Run

When examining the performance of Kursawe-Shoup, only the optimistic phase is of interest, since in a run with only honest hosts, the recovery procedure will not be executed.

When a client c writes a message m to the bulletin board consisting of servers P, the following steps are executed:

1. c sends (**write**, m) to some server p.
2. p sends (**initiate**, e, m) to the leader.
3. upon receipt, the leader sends (**0-bind**, e, m, $scnt$) to every server.
4. upon receipt, every server sends (**1-bind**, e, m, s) to every server.
5. upon receiving enough **1-bind** messages, every server sends (**2-bind**, e, m, s) to every server.
6. upon receiving enough **2-bind** messages, every server A-delivers m, computes a signature share and sends (**write-ack**, m_K) to p.
7. p combines the shares into a single signature and sends (**write-ack**, m_K) to c.

When a client c reads all l messages, the following steps are executed:

1. c chooses a challenge d and sends (**read-all**, d) to some server $p \in P$.
2. p sends (**read-all-request**, d) to each $p' \in P$.
3. each $p' \in P$ computes a signature share over l messages and sends (**read-all-reply**, signature) to p.
4. p combines these shares into a single signature and sends (**read-all-ack**, messages, signature) to c.

Like in the Rampart protocol, the broadcast protocol is not used in this step, since the servers do not need to come to an agreement on something. Receiving $\left\lceil \frac{n-1}{3} \right\rceil + 1$ signature shares is enough to construct the threshold signature.

4.3.9 Analysis of Kursawe-Shoup

Resilience

$$\left\lfloor \frac{n-1}{3} \right\rfloor$$

Round Complexity

Write One	Read Many
7	4

Message Complexity

Write One	Read Many
$n + 2n^2$	$2n$

Communication Complexity

Write One	Read Many
$n + 2n^2$	$2n + l$

Computational Complexity

	Write One	Read Many
client	1	1
server	1	1
entry server	$\lceil \frac{n-1}{3} \rceil$	$\lceil \frac{n-1}{3} \rceil$

4.3.10 Phalanx Protocol Description

The protocol described in [MR98] does not use the broadcast approach taken by Rampart and Kursawe-Shoup, but instead clients contact a large part of the servers themselves. Each client contacts $\lceil (3n + 1)/4 \rceil$ servers when reading or writing a variable. Each set of $\lceil (3n + 1)/4 \rceil$ is a *Quorum*, so each client contacts a full quorum when reading or writing a variable.

Clients use timestamps tied to the messages they write, which are stored by servers to determine which message is most recently written. Each client c has its separate space of timestamps, T_c, with which a total order on timestamps can be defined without collisions between the timestamps of clients.

Since each quorum overlaps each other quorum in at least $\lceil (2n + 1)/4 \rceil$ other servers, each message written to a full quorum by a client is available for other clients reading a full quorum through at least $\lceil (2n + 1)/4 \rceil$ servers. At most $\lfloor (n - 1)/4 \rfloor$ servers are corrupt, so when at least $\lfloor (n - 1)/4 + 1 \rfloor$ servers agree on a message and timestamp, a client knows that this message is indeed written by another client. From the set of correct messages, a client selects the message with the highest timestamp.

Some servers contacted by a client reading a message might not have the message yet, since the client writing the message did not contact all servers, but only a single quorum. To eventually stabilize the message, after reading a message a client writes that message back to all servers that did not have the message already.

The protocols to read and write a variable by a client:

upon receiving a message (**in**, **write**, v)
    repeat send (**request-timestamp**) to $\lceil (3n+1)/4 \rceil$ servers q
    until ready-timestamp

upon receiving messages (**timestamp**, t_i) from $\lceil (3n+1)/4 \rceil$ different q_i with valid t_i
    set ready-timestamp
    Select timestamp $t \in T_c$ such that $t > t_i$ for all i
    repeat send (**request-signature**, $\langle v, t \rangle$) to $\lceil (3n+1)/4 \rceil$ servers q
    until ready-signature

upon receiving messages (**signature**, $\langle \text{echo}, v, t \rangle_{K_i}$) from $\lceil (3n+1)/4 \rceil$ different q_i with valid signatures
    set ready-signature
    repeat send (**write**)$\{\langle \text{echo}, v, t \rangle_{K_i}\}_i$ to server $q \in Q \in \mathbb{Q}$
    until ready-write

upon receiving messages (**write-ack**) from $\lceil (3n+1)/4 \rceil$ different q_i
    set ready-write

upon receiving a message (**in**, **read**)
    repeat send (**read**) to $\lceil (3n+1)/4 \rceil$ servers q
    until ready-read

upon receiving messages (**read-reply**, $\langle v_i, t_i \rangle\_K_i$) from $\lceil (3n+1)/4 \rceil$ different q_i
    set ready-read
    Choose from the $\langle v_i, t_i \rangle$ value pairs the pair $\langle \text{stored}, v, t \rangle$ that occurs at least $\lceil (n-1)/4 \rceil + 1$ times with the highest timestamp t, and send (**out**, **read-reply**, v). If no such pair exists, send (**out**, **read-reply**, $\perp$).
    repeat send (**writeback**, $\langle \text{stored}, v, t \rangle$ with $\lfloor (n-1)/4 \rfloor + 1$ signatures) to server $q \in Q \in \mathbb{Q}$
    until ready-writeback

upon receiving messages (**writeback-reply**) from $\lceil (3n+1)/4 \rceil$ different q_i
    set ready-writeback

Figure 6: Phalanx Protocol for Clients

The protocol that servers follow:

upon receiving a message (**request-timestamp**)
    choose t as a timestamp valid for the sender, higher than any previous timestamp
    send the message (**timestamp**, t) to the sender

upon receiving a message (**request-signature**, $\langle v, t \rangle$)
    if no pair $\langle v', t \rangle$ is signed before **then**
        send the message (**signature**, $\langle \text{echo}, v, t \rangle_{K_i}$) to the sender

upon receiving a message (**write**, $\langle v, t \rangle$, $\{\langle \text{echo}, v, t \rangle\}_i$)
    if $t > t_x$ **and** the signatures are valid and from $\lceil (3n+1)/4 \rceil$ different servers **then**
        $x := v; t_x := t$
    send (**write-ack**) to the sender.

upon receiving a message (**read**)
    send (**read-reply**, $\langle \text{stored}, x, t_x \rangle_K$) to the sender.

upon receiving a message (**writeback**, $\langle \text{stored}, v, t \rangle$ with $\lfloor (n-1)/4 \rfloor + 1$ signatures)
    if $t > t_x$ **then**
        $x := v; t_x := t$
    send (**writeback-reply**) to the sender.

Figure 7: Phalanx Protocol for Servers

This protocol does not offer much protection against writers that write the value to only a part of a quorum. When this happens, a read may or may not return that value, since there exist quorums where not enough servers received that value. Once a read operation returns that value, the read operation writes back the value to a full quorum, so from then on, reads return the correct value. This may be

very inconvenient, as it looks like a vote appearing long after the voting period is closed.

This protocol also does not enable us to optimize the reading of all messages at once. Since each server does not contain the full set of messages, each set of messages must be read off each server. There is not even an order on the messages, so operations like "read the the first 1000 messages" are not possible. The best that can be done is to read all messages stored at one server, and then read all messages of the second server, and repeat this for $\frac{3n}{4}$ servers.

4.3.11 Phalanx Protocol Run

When a client c writes a variable to a quorum of servers Q of size $\lceil (3n+1)/4 \rceil$, the following protocol is executed:

1. c sends (**request-timestamp**) to every server in Q.
2. Every server in Q sends (**reply-timestamp**, t) to c.
3. c sends (**request-signature**, $\langle v, t \rangle$) to every server in Q.
4. Every server in Q sends (**reply-signature**, $\langle v, t \rangle_{K_j}$) to c.
5. c sends (**write**, $\langle v, t \rangle$, $\{\langle v, t \rangle_{K_j}\}$) to every server in Q. The list of signatures in the message is a list of size $\left\lceil \frac{3n+1}{4} \right\rceil$.
6. Every server in Q computes its signature on the pair and sends (**write-ack**) to c.

To read a variable, the following protocol is executed:

1. c sends (**read**) to every server in Q.
2. Every server in Q sends (**read-reply**, $\langle$**stored**, $v, t \rangle_{K_j}$) to c.
3. c sends (**writeback**, $\langle$**stored**, $v, t \rangle$ together with $\lfloor (n-1)/4 \rfloor + 1$ signatures) to some of the servers in Q.
4. The servers that received an update send (**writeback-reply**) to c.

To read a set of variables, the previous protocol is executed for each variable.

4.3.12 Analysis of Phalanx

Resilience

$$\left\lfloor \frac{n-1}{4} \right\rfloor$$

Round Complexity

Write One	Read Many
6	4

Message Complexity

Write One	Read Many
$\frac{9n}{2}$	$l \cdot 3n$

Communication Complexity

Write One	Read Many
$\frac{15n}{4} + \frac{9n^2}{16}$	$l \cdot \left(\frac{3n}{2}[+\frac{n^2}{6} + \frac{n}{2}]\right)$

Note: The communication complexity is very unpredictable, because it depends on the circumstances whether many servers need to be updated. The part between square brackets denotes some value between zero and the indicated value.

Computational Complexity

	Write One	Read Many
client	$\frac{9n}{4}$	$l \cdot \left(\frac{3n}{4}[+\frac{n^2}{6} + \frac{n}{2}]\right)$
server	$3 + \frac{3n}{4}$	$l \cdot \left(\frac{3n}{2}[+\frac{n^2}{6} + \frac{n}{2}]\right)$

4.3.13 Phalanx+ Protocol Run

When we substitute the signature of the $\langle v, t \rangle$ pair by a threshold signature share, we can substitute the list of signatures in the commit packet with a single threshold signature, thereby reducing the communication complexity to $O(n)$. The servers now also do not need to compute their own signature over the value/timestamp pair after they have been written. The amount of time spent on combining the signature shares into one signature is done by the client, which is an advantage in our setting, as clients have plenty of time during voting, since they only need to cast a single vote.

4.3.14 Analysis of Phalanx+

Since only the communication complexity and the computational complexity change with respect to Phalanx, the other criteria have been omitted.

Communication Complexity

Write One	Read Many
$\frac{9n}{2}$	$l \cdot \left(\frac{3n}{2}[+\frac{n^2}{6} + \frac{n}{2}]\right)$

Computational Complexity

	Write One	Read Many
client	$\frac{12n}{4}$	$l \cdot \left(\frac{3n}{2}[+\frac{2n}{2}]\right)$
server	4	$l \cdot (1[+2])$

4.3.15 PhalanxII Protocol Description

Like Phalanx, a client accesses quorums when reading or writing a variable. In the protocol described in [MR97], however, servers also contact each other during writings.

A client writing a message starts the protocol by choosing a timestamp value, and sending an **update** message to a quorum of servers. Each server in the quorum sends an **echo** message containing the message to each other server in the quorum, and upon receiving the **echo** message from each server, it sends a **ready** message to each server in the quorum. If enough of these **ready** messages have arrived, each server sends an **update-ack** message to the client, signalling that the write succeeded. Reading a message is done in exactly the same way as it is done in Phalanx.

Clients follow this protocol to read and write a variable:

upon receiving a message (**in**, **write**, v)
Choose a timestamp $t \in T_c$ greater than any previously used timestamp.
repeat Fix a set Q of servers where $|Q| = \lceil (3n+1)/4 \rceil$ and send (**update**, Q, v, t) to every server $q \in Q$
until ready-update

upon receiving messages (**update-ack**) from q_i such that $\cup_i q_i \supseteq Q$ and $|Q| = \lceil (3n+1)/4 \rceil$
set ready-update

upon receiving a message (**in**, **read**)
repeat send (**read**) to server $q \in Q$ where $|Q| = \lceil (3n+1)/4 \rceil$
until ready-read

upon receiving messages (**read-reply**, $\langle v_i, t_i \rangle\_K_i$) from q_i such that $\cup_i q_i \supseteq Q$ and $|Q| = \lceil (3n+1)/4 \rceil$
set ready-read
Choose from the $\langle v_i, t_i \rangle$ value pairs the pair $\langle v, t \rangle$ that occurs at least $b+1$ times with the highest timestamp t, and send (**out**, **read-reply**, v). If no such pair exists, send (**out**, **read-reply**, $\perp$).

Figure 8: PhalanxII Protocol for Clients

Servers follow this protocol:

upon receiving a message (**update**, Q, v, t) from client c
if $t \in T_c$, and if the server has not previously received from c a message (**update**, Q', v', t') where either $t' = t$ and $v' \neq v$ or $t' > t$, **then**
send (**echo**, Q, v, t) to each $q \in Q$.

upon receiving messages (**echo**, Q, v, t) from every $q \in Q$
send (**ready**, Q, v, t) to each $q \in Q$.

upon receiving messages (**ready**, Q, v, t) from a set B^+ of servers, such that $|B^+| \geq \lfloor (n+1)/4 \rfloor + 1$
send (**ready**, Q, v, t) to each $q \in Q$ if it has not done so already.

upon receiving messages (**ready**, Q, v, t) from a set Q^- of servers, such that $|Q^-| \geq \lceil (2n+1)/4 \rceil$
if $t > t_x$ **then**
$x, t_x := v, t$;
send (**update-ack**) to c, where c is taken from $t \in T_c$.

upon receiving a message (**read**)
send (**read-reply**, $\langle x, t_x \rangle_K$) to the sender.

Figure 9: PhalanxII Protocol for Servers

Since, like the other Phalanx protocol, each variable is not stored at every server, reading large sets of messages has the same disadvantages as in Phalanx.

4.3.16 PhalanxII Protocol Run

To write a variable to a quorum Q, the following protocol is executed:

1. c sends (**update**, Q, v, t) to every server in Q.
2. every server in Q sends (**echo**, Q, v, t) to every server in Q.
3. every server in Q sends (**ready**, Q, v, t) to every server in Q.
4. every server in Q sends (**update-ack**, t)o c.

To read a variable from a quorum Q, the following protocol is executed:

1. c sends (**read**) to every server in Q.
2. Every server in Q sends (**read-reply**, $\langle x, t_x \rangle_{K_i}$) to c.

To read a set of variables, the previous protocol is executed for each variable.

4.3.17 Analysis of PhalanxII

Resilience

$$\left\lfloor \frac{n-1}{4} \right\rfloor$$

Round Complexity

Write One	Read Many
4	2

Message Complexity

Write One	Read Many
$\frac{9n^2}{8} + \frac{3n}{2}$	$l \cdot \frac{3n}{2}$

Communication Complexity

Write One	Read Many
$\frac{9n^2}{8} + \frac{3n}{2}$	$l \cdot \frac{3n}{2}$

Computational Complexity

	Write One	Read Many
client	$\frac{3n+4}{4}$	$l \cdot \frac{3n}{4}$
server	$\frac{3n}{2}$	$l \cdot 1$

4.4 Summary of the Analyses

4.4.1 Resilience

With n servers, Rampart(+) and Kursawe-Shoup have resiliences of $\left\lfloor \frac{n-1}{3} \right\rfloor$, the Phalanx(II)(+) protocols have resiliences of $\left\lfloor \frac{n-1}{4} \right\rfloor$.

4.4.2 Round Complexity

	Rampart	*Rampart+*	*Kursawe − Shoup*	*Phalanx*	*Phalanx+*	*PhalanxII*
Write one	7	7	7	6	6	4
Read many	4	4	4	4	4	2

4.4.3 Message Complexity

Writing one variable to or reading many variables from a system of n servers (where only $\frac{3n}{4}$ servers need to be contacted by the quorum-based protocols) induces the following amount of messages:

	Rampart	*Rampart+*	*Kursawe − Shoup*	*Phalanx*	*Phalanx+*	*PhalanxII*
Write one	$4n$	$4n$	$n + 2n^2$	$\frac{9n}{2}$	$\frac{9n}{2}$	$\frac{9n^2}{8} + \frac{3n}{2}$
Read many	$2n$	$2n$	$2n$	$l \cdot 3n$	$l \cdot 3n$	$l \cdot \frac{3n}{2}$

4.4.4 Communication Complexity

The communication complexity is a more accurate measure of network traffic than message complexity. Between two messages of different size, but which still both fit in a single ethernet frame (1500 bytes), differences in speed will not be very big, but when a single message contains signatures for many servers, the size of the message will grow much larger than an ethernet frame, so the differences in speed will be noticeable.

	Rampart	*Rampart+*	*Kursawe − Shoup*	*Phalanx*	*Phalanx+*	*PhalanxII*
Write one	$3n + \frac{2n^2+n}{3}$	$4n$	$n + 2n^2$	$\frac{9n^2}{16} + \frac{15n}{4}$	$\frac{9n}{2}$	$\frac{9n^2}{8} + \frac{3n}{2}$
Read many	$2n + l$	$2n + l$	$2n + l$	$l \cdot \left(\frac{3n}{2}\left[+\frac{2n^2}{16} + \frac{2n}{4}\right]\right)$	$l \cdot 3n$	$l \cdot \frac{3n}{2}$

4.4.5 Computational Complexity

Because public key cryptography operations will take most of the computation time, this section only deals with the number of such operations. A distinction is made between client and server time, because in our setting, the time that clients spend on calculation does not cost us anything. The client needs to prove its identity in both protocols, but other methods than public key cryptography might be used, so this is not taken into consideration.

The following table lists the number of public key operations performed by an average server. This means that for the Rampart(+) protocols, a non-entry server is taken.

	Rampart	*Rampart+*	*Kursawe − Shoup*	*Phalanx*	*Phalanx+*	*PhalanxII*
Write one	$\left\lceil\frac{2n+1}{3}\right\rceil$	2	1	$3 + \frac{3n}{4}$	4	$\frac{3n}{2}$
Read many	1	1	1	$l \cdot \left(\frac{3n}{2}\left[+\frac{n^2}{6} + \frac{n}{2}\right]\right)$	$l \cdot (1[+2])$	$l \cdot 1$

To give an indication of the latency of a write or read action, the amount of parallelization must be taken into account. If one server performs all signing and verifying operations, latency will be high, but if all servers perform an equal amount of those operations, latency will be lower. The following table lists the maximum number of public key operations performed by any of the servers. For the Rampart(+) protocols, this will be the entry server, for the Phalanx(II)(+) protocols, this will make no difference.

	Rampart	*Rampart+*	*Kursawe − Shoup*	*Phalanx*	*Phalanx+*	*PhalanxII*
Write one	$\left\lceil \frac{2n+1}{3} \right\rceil$	$\left\lceil \frac{4n+2}{3} \right\rceil$	$\left\lceil \frac{n-1}{3} \right\rceil$	$3 + \frac{3n}{4}$	4	$\frac{3n}{2}$
Read many	$\left\lceil \frac{2n+1}{3} \right\rceil$	$\left\lceil \frac{4n+2}{3} \right\rceil$	$\left\lceil \frac{n-1}{3} \right\rceil$	$l \cdot \left(\frac{3n}{2} \left[+ \frac{n^2}{6} + \frac{n}{2} \right] \right)$	$l \cdot (1[+2])$	$l \cdot 1$

Before we make our decision on what protocol we will use in our bulletin board, we first examine threshold signatures and describe the adversary in more detail.

5 Threshold Signatures

Threshold signatures play an important role in the design of the bulletin board. Not only does the client receive a threshold signature from the bulletin board in acknowledgement of her voting, but the speed of the core protocol layer of the bulletin board depends heavily on the speed of the threshold signatures. This chapter describes the requirements on the threshold signature scheme to be used, and discusses several schemes.

5.1 Preliminaries

Before we delve into the details of threshold signature schemes, we first describe the preliminaries necessary to understand the rest of this chapter.

5.1.1 The Discrete Logarithm Assumption

The discrete logarithm assumption is the assumption that in some groups it is hard to calculate the discrete logarithm of a value. This assumption is used in many of the schemes presented in this chapter.

Discrete Logarithm Assumption The discrete logarithm assumption for group G states that it is hard to compute x given generator g and random group element g^x.

Often a subgroup of $\mathbb{Z}/p\mathbb{Z}^*$, or $\mathbb{Z}_p^*$ for short, is used, where p is a prime such that there exists a prime q dividing $p-1$. Then there exists a generator g of order q. When the size of p is around 1024 bits and the size of q is 160 bits, the discrete logarithm problem for the cyclic group generated by g is believed to be hard.

5.1.2 Threshold Secret Sharing Scheme

A secret sharing scheme is used by a dealer to let a number of participants share a secret value, without letting small groups of parties have any information on the secret. Only until a group large enough pool their individual shares, they can recover the secret. In a (t, n)-threshold secret sharing scheme, a dealer distributes shares to the n participants, and only groups of t or more participants can recover the secret. The scheme consists of two algorithms:

Distribution A protocol in which dealer D shares a secret s such that each participant p_i obtains a share s_i, $1 \leq i \leq n$.

Reconstruction A protocol in which secret s is recovered by pooling shares s_i, $i \in A$, $|A| \geq t \wedge A \subseteq \{1, \ldots, n\}$.

A simple and elegant (t, n)-threshold secret sharing scheme is proposed by Shamir. The scheme for sharing a secret $s \in \mathbb{Z}_p$, p a primepower is defined as follows:

Distribution The dealer picks a random polynomial $a(x) \in_R \mathbb{Z}_p[x]$ of degree $< t$ satisfying $a(0) = s$. It sends share $s_i = a(i)$ to participant P_i for $i \in \{1, \ldots, n\}$.

Reconstruction Any set A of t participants may recover secret s from their shares by Lagrange interpolation:

$$s = \sum_{i \in A} s_i \lambda_{A,i}, \text{ with } \lambda_{A,i} = \prod_{j \in A \setminus \{i\}} \frac{j}{j-1}$$

To see why reconstruction works, recall that the Lagrange interpolation formula for the unique polynomial $a(x)$ of degree $< t$ passing through the points (i, s_i), $i \in A$, is given by

$$a(x) = \sum_{i \in A} s_i \prod_{j \in A \setminus \{i\}} \frac{x-j}{j-1}$$

Since we are interested in the constant term $s = a(0)$ only, we may substitute x with 0.

This scheme does not give any protection against malicious parties contributing incorrect shares, or the dealer giving out incorrect shares. Feldman introduced a verifiable secret sharing scheme, based on Shamir's secret sharing scheme. The idea is to let the dealer broadcast commitments of the coefficients of the polynomial $a(x)$, which do not reveal information about the polynomial or the secret under the discrete logarithm assumption, but which enable participants to verify the correctness of their share and the correctness of shares contributed by other parties.

Let $\langle g \rangle$ denote a cyclic group of large prime order n with generator g, and let the discrete logarithm problem in group $\langle g \rangle$ be hard. Feldman's VSS is given by:

Distribution The dealer chooses a random polynomial of the form

$$a(x) = s + \alpha_1 x + \cdots + \alpha_{t-1} x^{t-1},$$

where $\alpha_j \in_R \mathbb{Z}_n$, $1 \leq j < t$. The dealer sends shares $s_i = a(i)$ to participant p_i in private, for $i \in \{1, \ldots, n\}$. In addition, the dealer broadcasts commitments $B_j = g^{\alpha_j}$, $0 \leq j < t$ and $\alpha_0 = s$. Upon receipt of share s_i, participant p_i verifies its validity by evaluating the following equation:

$$g^{s_i} = \prod_{j=0}^{t-1} B_j^{i^j}.$$

Reconstruction Each share s_i contributed by participant p_i is verified using the previous equation. The secret $s = a(0)$ is then recovered as in Shamir's scheme from t valid shares.

5.1.3 Signature Scheme

A signature scheme presents a way to authenticate information. Given some data, only parties with access to a certain private key can compute a valid signature on that data, where everyone with the public key corresponding to the private key can check the validity of the signature.

A signature scheme consists of three algorithms:

Key Generation The algorithm to pick a private and a public key.

Signature Generation Given a private key, this algorithm computes the signature on a block of data.

Signature Validation Given a signature, a public key and a block of data, the validity of the signature on the data can be verified with the signature verification algorithm.

An example of a signature scheme which we will use in this project is Schnorr's signature scheme. Let $\langle g \rangle$ denote a cyclic group of large prime order n with generator g, and let the discrete logarithm problem in group $\langle g \rangle$ be hard. Furthermore, let $H(\cdot)$ denote a hash function such that $H(m)$ denotes the hash value of message m. Schnorr's signature scheme is given by:

Key Generation Let $x \in_R \mathbb{Z}_n$ be the private key, and let $h = g^r$ be the corresponding public key.

Signature Generation On input of a message m and a private key x, choose $u \in_R \mathbb{Z}_n$, set $a = g^u$, $c = H(a, m)$, and $r = u + xc$. The signature on m is the pair (c, r).

Signature Validation On input of a message m, a pair (c, r), and a public key h, accept (c, r) as a signature on m if and only if $c = H(g^r h^{-c}, m)$ holds.

For short, a Schnorr signature on message m in group $\langle g \rangle$ under private key x is the value pair $(H(m, g^r), r + H(m, g^r)x)$ with $r \in_R \mathbb{Z}n$, n the order of $\langle g \rangle$.

Under the discrete logarithm assumption, and in the random oracle model, Schnorr's signature scheme is secure.

5.1.4 Threshold Signature Scheme

Like a normal signature scheme, a threshold signature scheme is used to authenticate data. The difference however, is that out of a group of parties, several parties have to cooperate to generate a valid signature. In a (t, n)-threshold signature scheme, the private key is shared between n parties so that t parties can jointly produce valid signatures. The Signature Generation algorithm is now a distributed protocol, and we can split the signature generation protocol into two steps: signature share generation and signature combination. Therefore, a threshold signature scheme consists of these algorithms:

Key Generation This is a distributed protocol in which all parties obtain the same public key, and each party obtains their own share of the private key.

Signature Share Generation Given a private key share, this algorithm computes the signature share on a block of data.

Signature Share Combination Given t signature shares obtained from different parties, these shares are combined into one signature.

Signature Validation Given a signature, a public key and a block of data, the validity of the signature on the data can be verified with the signature verification algorithm.

In some threshold signature schemes, the signature share generation may be a distributed protocol, in which each party has to communicate with other parties in order to generate signature shares. These threshold signature schemes are *interactive*.

5.2 Requirements and Assumptions

Each party of the bulletin board generates its own public key pair. The public key of each party must be available to all parties of the bulletin board, while the private keys must remain secret. These public keys are used for protecting the communication channels between the parties, and may be used directly by the threshold signature scheme.

The key generation protocol of the threshold signature scheme must be completely distributed. The goal of this project is to provide a bulletin board where there is no single point of failure, so there cannot be a single trusted party that generates the keys.

The performance of the threshold signature scheme is very important. The echo multicast protocol, the core protocol of the bulletin board, uses a threshold signature on each message that is multicast. Ideally, a party that wishes to create a signature on a message sends that message to all other parties, receives a signature share from each party and combines those shares into a single signature. Such a scheme is non-interactive: parties that create a signature share do not need to communicate with other parties. If a non-interactive threshold signature scheme produces signatures of constant size, this signature scheme will (most likely) give rise to a complexity linear in the number of participants when broadcasting a message: each message is sent n times, generating n threshold signature shares requires n computations, and combining those shares requires a computation linear in n. Finally, confirming the message by sending the threshold signature to each party costs time linear in n. The threshold signature scheme used in [Rei94], is non-interactive, but produces signatures of size linear in n. Therefore, the last step in the protocol takes $O(n^2)$ time. The main purpose of this chapter is to improve on that bound.

5.3 A Trivial Threshold Signature Scheme

First, we will have a look at the threshold signature scheme used in [Rei94].

Key Generation There is no key generation protocol, the existing public keys of each party are sufficient to create a threshold signature.

Signature Share Generation A party generates its threshold signature share on a message by signing that messages with its own private key.

Share Combination To combine the signature shares into one signature, a party waits until it has received exactly the threshold t shares, and creates a list of pairs of the signature share and its signer. This list is the threshold signature.

Signature Validation To verify that the list obtained in the previous step is a valid threshold signature, check that its size is exactly the threshold t, check that no party has submitted two signature shares, and verify each signature share.

Since this scheme does not have a key generation, it meets our requirements on the key generation. The size of a threshold signature, however, is linear in the number of participants. We would like to come up with a smarter scheme in which signature shares are combined into one signature of constant size.

5.4 Pedersen's Threshold Signature Scheme

Pedersen's threshold signature scheme [Ped91] is based on Schnorr signatures [Sch89] and Feldman's verifiable secret sharing scheme (*VSSS*) [Fel87]. See [GJKR03] for security proofs.

Recall that a Schnorr signature on message m with private key x in group $\langle g \rangle$ is $(H(m, g^r), r + H(m, g^r)x)$, where $r \in_R \mathbb{Z}_n$ and $H(\cdot)$ is a hash function. A threshold signature scheme is easily constructed from this signature scheme.

Let each party p_i have a private key share x_i where $\sum_i x_i = x = \sum_{i \in A} x_i \lambda_{A,i}$, where $\lambda_{A,i} = \sum_{j \in A \setminus \{i\}} \frac{j}{j-i}$ for each subset A of size t, and let $y = g^x$ be public to every party. Feldman's verifiable secret sharing scheme is used to generate such x_i: Each party runs Feldman's threshold secret sharing scheme with itself as dealer, and computes its x_i as the sum of all received shares. Since in Feldman's VSS scheme commitments of the coefficients of the polynomial $g^{\alpha_{ij}}$ are broadcast, and since $x_i = \alpha_{i0}$, the public key y is computed by taking the power of all broadcast $g^{\alpha_{i0}}$.

To generate a signature on message m, a distributed secret r is generated to compute the Schnorr signature $(H(m, g^r), r + H(m, g^r)x)$. Denote each party's share of r by r_i. The value of each r_i is generated in the same way as the value of x_i, and after generation each party computes the value of g^r similar to the value of $y = g^x$. Each party computes its signature share as $(c, s_i) = (H(m, g^r), r_i + H(m, g^r)x_i)$. Combining t shares, the signature on message m can be computed as $(c, \sum_{i \in A} s_i \lambda_{A,i})$.

To summarize:

Key Generation Each participant runs Feldman's VSSS as dealer to share a secret. These shares are used to compute each party's secret key.

Signature Share Generation Each party runs Feldman's VSSS as dealer again, to generate a global random value r. Then, each party computes its signature share as $(H(m, g^r), r_i + H(m, g^r)x_i)$.

Signature Combination From t signature shares from parties $i \in A$, the signature can be computed as $(c, \sum_{i \in A} s_i \lambda_{A,i})$.

Signature Validation The signature is a valid Schnorr signature which can be validated by the publicly available value g^x.

The key generation protocol meets our requirements. It is efficient, and fully distributed. The signature share generation, however, is not efficient enough. Though it has the same complexity as the key distribution, our requirements are tighter here. Since each party runs as a dealer in Feldman's VSSS, each party sends messages to all other parties. This gives rise to at least a bit complexity of $\omega(n^2)$. While this is efficient enough for key generation, such a scheme would not give our bulletin board the speed increase we are looking for. The trivial scheme presented in the previous section also has bit complexity of $O(n^2)$.

5.5 Shoup's Threshold Signature Scheme

Pedersen's threshold signature scheme is not efficient enough for our purposes, because it is interactive. In fact, every threshold signature scheme based on discrete logarithms appears to be interactive, since the joint generation of the random value needed for these schemes appears to require

interaction. The RSA setting, however, does not need such a randomization, and Shoup describes a threshold signature schemes based on RSA signatures [Sho00]. The generation of threshold signatures is non-interactive. This scheme requires a modulus where the prime factors are of special form, which presents difficulties in jointly generating the primes. [DK00] proposes modifications to [Sho00], which removes the requirements on the primes, so that [FMY98] can be used as the threshold key generation protocol. The threshold signature scheme proposed in [Sho00] with the modifications of [DK00] meets our requirements: The threshold key generation [FMY98] is fully distributed; the generation of each signature share is done in constant time and does not need interaction; and the combination of enough signature shares results in a signature of constant size in linear time. However, jointly generating an RSA modulus is far from trivial: Although [FMY98] proposes such a protocol, it is quite complicated and the number of protocol rounds is quadratic in the number of bits of the modulus. We would therefore rather have another threshold signatures scheme, where a simpler threshold key generation protocol suffices.

5.6 Adapted Version of Pedersen's Threshold Signature Scheme

In this section, we propose a modified version of Pedersen's threshold signature scheme. This scheme, however, provides linear bit complexity only in the optimistic case. Since we study the performance of the bulletin board in the absence of faulty parties, this is tolerable. It uses the same key generation protocol as used in Pedersen's version of the scheme. Basically, we ask the parties to generate a random value r_i, and return g^{r_i}. We then select t of these values, and compute $g^r = \prod_{i \in A} (g^{r_i})^{\lambda_{A,i}}$. We present the parties in A with g^r and ask them to compute their signature share using this value. If they return a signature share, we can compute the signature as normal.

This protocol fails if a corrupt party does return a g^{r_i}, but afterwards does not return a signature share. After some timeout, we can either select a different set not containing the faulty party, or we can revert to another threshold signature scheme, which is more costly in terms of network traffic.

If no corrupt parties disturb the protocol, we have a protocol with a bit complexity of n, that produces signatures of constant size, and that uses an acceptable key generation protocol.

5.7 Threshold Signatures Based on Gap Diffie-Hellman Groups

Recall the following problems, informally stated:

Computational Diffie-Hellman (CDH) problem Given g^x and g^y, compute g^{xy}.

Decision Diffie-Hellman (DDH) problem Given g^x, g^y, and g^z, determine if $xy = z$.

In the usual modular arithmetic in $\mathbb{Z}_q$, both problems are believed to be hard. There are certain groups, however, that have the property that the CDH problem is believed to be hard, while the DDH problem is actually easy to solve. These are the Gap Diffie-Hellman groups. Such groups give rise to a signature scheme [BLS01]. Let $\langle g \rangle$ be a group, and let $H(\cdot)$ be a hash function that hashes messages onto the group $\langle g \rangle$.

Key Generation Select a private key $x \in_R \langle g \rangle$, and compute public key $y = g^x$.

Signature Generation Given a message m, compute the signature $s = H(m)^x$.

Signature Validation Given a signature s on a message m, validate the signature by checking that g, y, $H(m)$ and s are a valid Diffie-Hellman tuple.

Since no shared random secret is needed in the signature generation, this scheme is easily converted into a threshold signature scheme [Bol02].

Distributed Key Generation As with Pedersen's Threshold Signature Scheme, Each participant runs Feldman's VSSS as dealer to share a secret. This results in each party p_i having a x_i, and a global $y = g^x$ where x is determined by each t out of n parties.

Signature Share Generation Given a message m, compute the signature $s_i = H(m)^{x_i}$.

Signature Share Combination From t valid shares $s_i \in A$, compute the threshold signature $s = \Pi_{i \in A} s_i \lambda_{A,i}$.

Signature Validation Given a signature s on a message m, validate the signature by checking that y, $H(m)$ and s are a valid Diffie-Hellman tuple.

We now have a threshold signature scheme that has a simple key generation, and efficient generation and combination algorithms. That leaves us with only one problem: how do we find a Gap Diffie-Hellman group? Without going into too much detail, we remark that bilinear maps on elliptic curves such as the Weil and the Tate pairing give rise to Gap Diffie-Hellman groups. A bilinear map is a binary function with the following property:

$$P(g^{ax}, g^{by}) = P(g^x, g^{by})^a = P(g^{ax}, g^y)^b = P(g^x, g^y)^{ab}$$

Since $P(g^x, g^y) = P(g, g)^{xy} = P(g, g^{xy})$, we can decide that given a tuple (g^x, g^y, g^z) if $z = xy$ holds by checking if $P(g^x, g^y) = P(g, g^z)$ holds.

5.8 Conclusion

The trivial signature scheme is very easy to implement, but does not give the best performance. Threshold signature schemes based on RSA signatures have a threshold RSA modulus generation protocol that is generally too difficult and inefficient to implement. Elliptic curves give rise to good performance, but implementing elliptic curves may provide difficulties. The adapted version of Pedersen's threshold signature scheme also gives good performance in the optimistic case, and is easily implemented.

Only the trivial signature scheme does not need the bulletin board in the key generation protocol. This signature scheme will therefore be used at least in the setup phase of the bulletin board. After the setup phase, we may switch to a different signature scheme and both the adapted version of Pedersen's threshold signature scheme, and the threshold signature scheme based on elliptic curves meet our requirements. As we will see in a later chapter, the threshold signature scheme based on elliptic curves is much easier to incorporate in our protocols.

6 Attack Models

The protocols work under different assumptions on the adversary. In this chapter, we examine the attack model of the Kursawe-Shoup protocol and the Rampart protocol. Before we do this, however, we outline the specifications imposed by an election scheme.

6.1 The Context

In order to determine what kinds of attack our voting system must be able to withstand, we first examine what a proper election looks like. Once the requirements are known, we can point out ways to disrupt the election, and define different kinds of adversaries.

An election is held on a preselected day. There is a certain period in that day during which people can cast their votes. We assume that every voter can identify himself as a legitimate voter, and that each person that identifies himself as a voter is indeed a legitimate voter. Each voter must have the opportunity to cast his vote, but each voter can vote at most once. Each voter must be convinced that his vote is counted correctly, and that his vote is kept confidential. At the end of the day, when the election period is over, the votes must be tallied. The result of that tally must be that the exact number of votes cast for each candidate is known. The result must also be trusted by being verifiable, and in case of malfunctions, be recoverable. The outcome of the election must be made public in a reasonable amount of time.

6.2 Security Properties

We distinguish two probably, but not necessarily, disjoint groups of adversaries. The first group wants to change the outcome of the election. The second group wants to sabotage the election.

The first group consists of persons that favor one of the candidates, and they may try to prevent certain people from voting, change votes, or interfere with the tallies. This group is very well-funded, and capable of bribing persons. An attack by this group is successful if the outcome of the election is shifted in favor of the attackers. In order to achieve this, the attackers have to corrupt parties, and this will be done before the election starts. We can therefore consider this group of attackers to be a static adversary. This group targets the *Integrity* of the bulletin board.

The second group might consist of terrorists, anarchists or just plain script kiddies. The goal is to disrupt the election by mounting denial of service attacks in order to prevent access to servers. This can be done by icmp flooding, or by impersonating voters and having servers devote their entire cpu time to calculations only to find that these were not necessary. An attack by this group is successful if at any point during the election period, people were unable to cast their vote and receive their receipt in a timely manner. The attackers can choose the parties on which they mount their denial of service attacks. We can therefore consider this group of attackers to be an adaptive adversary, but limited in their attacks. This group targets the *Robustness* of the bulletin board.

Privacy is a major requirement in election schemes. Our bulletin board, however, does nothing to protect the privacy of voters since the election scheme built on top of the bulletin board handles privacy.

6.3 The Adversary in Kursawe-Shoup

The parties participating in the broadcast channel specified by Kursawe and Shoup in [KS01] communicate over an insecure, asynchronous network. The adversary may corrupt at most t of the parties, where $t < \left\lfloor \frac{n-1}{3} \right\rfloor$. Furthermore, no assumptions are made about the network, which is left under the complete control of the adversary: the network is the adversary. Parties send their message to the adversary, and the adversary may choose to deliver messages faithfully, alter messages, or just discard them. When the adversary decides to cut off a party from the network, all messages for that party remain queued until the adversary lets them pass.

When facing an adversary, the adversary succeeds in compromising the integrity or robustness of the bulletin board when they corrupt or affect at most $\left\lfloor \frac{n-1}{3} \right\rfloor$ parties. When one of the parties is being cut off the network, the other parties will queue all messages for that party. When dealing with long lasting outages, this might become a problem.

6.4 The Adversary in Rampart

The Rampart protocol described in [Rei94] builds on a secure membership protocol. Any party that is found to be faulty, is removed from the broadcast channel, whether this party is dishonest or just unresponsive. The protocol allows for parties to be accepted into the broadcast channel while running, so formerly unresponsive parties can re-enter the group. The major difference in attack model between this protocol and the [KS01] protocol, is that in this protocol if the network is the adversary, the adversary can simply remove honest parties by blocking communication, and then ensure that the corrupted parties are the majority of parties in the broadcast channel. Therefore, an application considering this protocol must ensure that the adversary cannot entirely control the network.

As with the previous protocol, at most $\left\lfloor \frac{n-1}{3} \right\rfloor$ parties may be affected or corrupted by the adversary. In contrast with the previous protocol, however, when a party is being cut off the network, messages are not kept in a queue. Rather, when the party re-joins the group, it will only receive messages from then on. This causes some servers to have missed a number of votes, but as each message is delivered atomically, each message may be numbered so that each server exactly knows what windows of votes it is missing. A protocol can be built on top of this protocol to synchronize the missing votes.

6.5 Hackers

There exists another adversary, and this adversary is very different from the adversaries discussed above. Hackers try to gain access to the machines running the bulletin board. Once they have compromised one machine, they can use the same tactic to gain access to the other machines, since they are likely similar and have the same weaknesses. This is different from the adversaries discussed above, because in this case, there are no or very small costs to compromise more machines after the first machine has been compromised. Therefore, it is advisable that the bulletin board is ran on different platforms, so that if a hacker compromised one particular platform, the number of machines compromised stays below the resilience of the bulletin board. For example, if the bulletin board consists of 28 servers, spread evenly over 4 platforms like machines running Windows XP, Linux, SGI IRIX and MacOS, the bulletin board is resilient against the corruption of 9 parties. If a hacker hacks all machines running Windows, and two administrators are bribed, then the bulletin board is still secure.

7 The Bulletin Board Core Protocol

We are now ready to select one of the above described protocols, and use that protocol to implement our bulletin board.

First we note that three of our protocols have a resilience of $\lfloor \frac{n-1}{3} \rfloor$, and the three other protocols have a resilience of $\lfloor \frac{n-1}{4} \rfloor$. This is not significant enough to justify a choice.

Communication and computational complexity have the greatest influence on throughput.

- If we look at the communication complexity, and examine the Write One operation, we notice that Rampart+ and Phalanx+ have $O(n)$ complexity and the other protocols have $O(n^2)$ complexity.
- If we read many variables, say x, then Rampart, Rampart+, and Kursawe-Shoup have a communication complexity of $O(n + l)$, while the Phalanx protocols have a complexity of at least $O(n \cdot x)$.
- The computational complexity of writing one variable is $O(1)$ in the Rampart+, Kursawe-Shoup, and Phalanx+ protocols, and $O(n)$ in the other protocols
- The computational complexity of reading many variables has a complexity of $O(1)$ in the Rampart, Rampart+, and Kursawe-Shoup protocols, while it has complexity of at least $O(l)$ in the Phalanx protocols.

In conclusion, the communication and computational complexity of each operation in Rampart+ is asymptotically never worse than in any other protocol. Each other protocol has an operation that has a worse complexity than in Rampart+.

There are more arguments for choosing Rampart above the other protocols:

- As seen in the previous chapter, the attack model assumed in Rampart more closely matches the attack model of an election than the attack model of Kursawe-Shoup.
- In the Phalanx protocols, messages "eventually stabilize", meaning that if a voter writes his vote to less parties than he should, for example to $\frac{n-1}{4} + 1$ parties instead of $3\frac{n-1}{4}$ parties, the vote remains hidden until a correct reader chooses those $\frac{n-1}{4} + 1$ parties to read the vote from. This means that after closing the election, suddenly new votes may appear. Of course, the parties may choose to synchronize right after closing the election, but that boils down to implementing a broadcast protocol like Rampart or Kursawe-Shoup.

We therefore choose **Rampart+**, the version of Rampart using threshold signatures, as the core protocol of our bulletin board.

8 Formal Specifications of the Protocols

We have selected the Rampart protocol as the basis of our bulletin board. We already described how a multicast protocol can be used to implement a bulletin board, but before we can start coding, we need proper descriptions of the protocols. The bulletin board consists of a secure group membership protocol, on top of which the three layers of Rampart are built, and again on top of which several layers of other protocols are built. Some of these protocols have clear descriptions, but others are described really vaguely, and still others are invented specifically for the bulletin board and have thus not been described at all. To acquire a clear and consistent specification from which the code can be derived, in this chapter we introduce a pseudo code and describe all needed protocols with it.

The protocols also have another purpose: while the specifications of Rampart in [Rei94] and of the secure group membership protocol in [Rei96] are probably well-suited for proofs of correctness, they are not much good as a guide for an implementation. The formal specifications given in this chapter form a bridge between the specifications of [Rei94] and [Rei96] and an implementation, by staying on the specification level but being far more explicit in details. This gives us a set of specifications that we can compare to the original specifications in order to verify the correctness, but these specifications also give a firm lead for the code.

8.1 Notation

In this section, we introduce a formalism used to describe our protocols. The most important thing protocols do is sending messages. Therefore, we start with describing what messages look like in our formalism, and we introduce means to send and receive them. We also describe how timers work. These timers are often needed in protocols to handle unresponsive parties. Finally, we describe how timeouts are specified.

8.1.1 Messages

We refine the notation introduced in section 4.3. Since the protocols discussed in this chapter are described in much greater detail, we need better descriptions of how messages and timers work.

The protocols communicate with each other using messages. The notation used in the syntax of these messages resembles the notation in [CKPS01, sect. 2.1.2]. We distinguish two different types of messages: horizontal messages and vertical messages. Horizontal messages travel from one party running a protocol to another party running the same protocol. Vertical messages travel from one protocol ran by a party to another protocol ran by the same party, i.e. another layer on the protocol stack.

Messages are of the form (**type**, parameters), for example (**init**, *x*, *m*). If the type is not **in** or **out**, it is a horizontal message that will be sent over the network. These types of messages are assumed to be sent in a reliable and confidential way, where the order of the messages are preserved. In practice, another protocol layer will take care of this. If its type is **in** or **out**, it is a vertical message which is sent between protocol layers of one party.

Protocols are described in an event-driven way. A protocol receives a vertical message with the clause ‘ON RECEIVING MESSAGE (**in**, **type**, parameters)’, and a horizontal message with the clause ‘ON RE-

CEIVING MESSAGE (**type**, parameters) FROM *source*'. A party sends a vertical message by executing 'send (**out**, **type**, parameters)', thereby triggering the ON RECEIVING clauses of the other protocol layer. A horizontal message may be sent by executing 'send (**type**, parameters) to *destination*'. Once this message is received by the other party, it triggers the appropriate ON RECEIVING clause.

When a received message contains a signature, its validity is implicitly verified. If the signature does not match, the message is discarded and the contents of the ON RECEIVING MESSAGE clause is not executed. When a received message contains bound parameters, those parameters are first treated as unbound and compared to the other variables with the same name. If they are not equal, the message is discarded and the contents of the ON RECEIVING MESSAGE clause is not executed.

8.1.2 Timers

In addition to responding to incoming messages, some protocols need to take certain actions when after a certain period they *did not* receive a message. To facilitate this, two additional constructions are defined. The first one is the PERIODICALLY clause, which is triggered regularly. The second one is the ON TIMER clause. This clause is identified by a name and a number of parameters. After a short time after the protocol executes **start timer** (**identifier**, parameters), the corresponding ON TIMER clause is triggered once with the given parameters. If the protocol decides that the timer does not need to be triggered anymore, it can execute **stop timer** (**identifier**, parameters) to prevent the ON TIMER clause from triggering. If the clause is already triggered, or if the timer has not been started, that command has no effect.

8.1.3 Signed Data

Data between angle brackets followed by a subscript expression represents the signature of data. For instance, $\langle x, \textbf{ack}, y \rangle_z$ is the signature obtained by signing the variables x and y and the identifier **ack** with private key z. Note that in most other publications, this would imply that x, **ack** and y are sent too. In this document, only the signature is sent. It is often not necessary to send all fields, and the protocols in this document must have enough detail to specify which fields are sent and which fields are not send.

Each datablock shall contain an identifier in boldface, to prevent a signature to be used in a different place than in which it was intended. Contrary to the order of the arguments in the notation $\langle x, \textbf{ack}, y \rangle_z$, which has the intuitive meaning of the signature of 'x gives an acknowledgement about y', an implementation must always put this identifier in front of variables, to prevent type-error attacks. There may be other ways to prevent from these kind of attacks, but moving a static identifier to the front of the signature is very simple and highly effective.

8.2 Composition of the Protocols

Rampart is composed of several layers of protocols. Before we describe each layer, we give a brief outline what each protocol's function is, and how the protocols cooperate.

Rampart uses a *Secure Group Membership* protocol. Our bulletin board consists of several servers, some of which may be unresponsive, some of which may even be corrupt. The protocol layers are

created in such a way that corrupt parties cannot destroy the integrity of the bulletin board, so corrupt parties may at most be unresponsive. The secure group membership protocol deals with unresponsive parties, by maintaining a *group view* of currently responsive parties. At first, each party is in the active group view. If a party becomes unresponsive, other parties may vote it out of the current group view. The secure group membership protocol then composes a new group view, where the unresponsive party is expelled. The other protocols then only deal with this new group, and are therefore able to continue making progress.

Rampart's multicast functionality is composed of three layers: the *Echo Multicast* protocol, the *Reliable Multicast* protocol, and the *Atomic Multicast* protocol. The echo multicast protocol is the lowest protocol, and does most of the work of the multicast, ensuring that each multicast message is received as the same message at every (honest) party. If a message is received by the echo multicast protocol at an honest party, that party has the assurance that every honest party will (eventually) receive that same message. These messages, however, are sent to one specific group view. If one of the parties becomes unresponsive, a change in the group view has to occur. The messages sent to the old group view have to be delivered before messages are delivered to the new group view. The reliable multicast protocol ensures just that. When a change in the group view occurs, the reliable multicast protocol queues new multicast messages, and first ends all messages sent in the previous group view. After closing that group view, the queued messages may be sent for the new group view.

The echo multicast and reliable multicast protocols ensure that the order in which messages are received from a particular party is the same as the order in which they were sent. Those protocols, however, do nothing to ensure that two messages sent by two different parties are received in the same order by every party. The atomic multicast protocol ensures that the order in which messages are received is the same at each party.

Even atomic multicast does not suffice for our bulletin board. Parties may have been removed from a group view to rejoin later. Such parties have missed a set of messages. An extra layer is introduced with which rejoining parties can safely recover each message it did not receive yet. This is the *Synchronized Atomic Multicast* protocol. On top of this protocol, we can easily build our bulletin board. Or, we choose to build our key generation on top of the synchronized atomic multicast protocol.

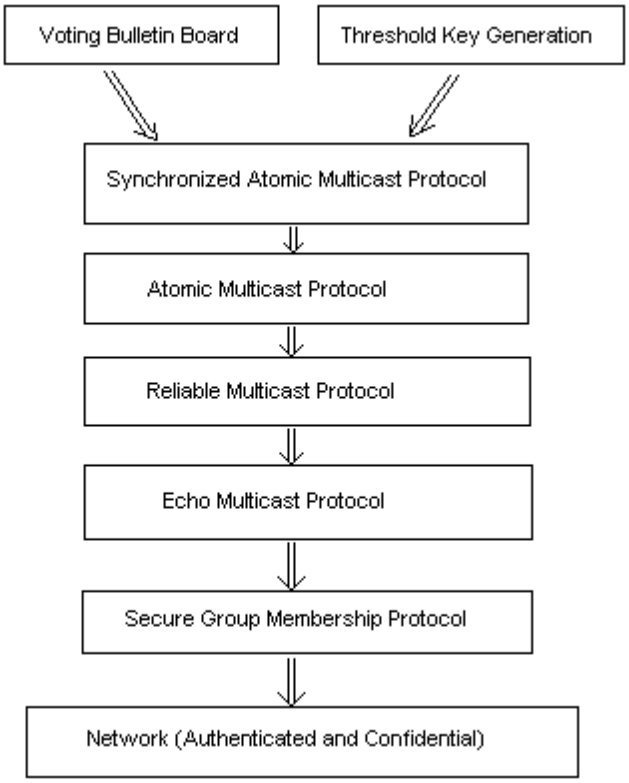

Figure 10: The Protocol Layers

The picture displays how the various protocols interact. The secure group membership protocol is the lowest protocol, which interacts with the network. The echo multicast protocol uses the group membership protocol, etc. On top of the synchronized multicast protocol, our bulletin board protocol is built.

8.3 The Group Membership Protocol

The protocol described in this section is detailed in [Rei96]. A secure group membership protocol is needed in the implementation of the various broadcast protocols in Rampart [Rei94]. This protocol ensures that in a set of parties P, the honest parties agree on the subgroup of P consisting of currently operational and correct parties. To accomplish this, each party p_i has a view V_i^x which consists of the currently operational parties. This view changes over time, so an x is used to denote the x-th view. Initially, each party is in view V_i^0, which will be configured manually by an administrator. When party p_i creates its view V_i^x, view V_i^x is said to be *defined*, and is *undefined* otherwise. The protocol ensures that all x-th views at each correct party are the same. Therefore, the subscript i is usually omitted: V^x denotes the x-th view.

Members of the current view can remove other members or invite new parties into the group. When, in view V^x, a party $p_i \in V^x$ discovers that party $p_j \in V^x$ is faulty, *faulty*(p_j) is said to hold at p_i. Otherwise, *correct*(p_j) holds at p_i.

One assumption is placed on the parties: We assume that at least $\lceil (2|P| + 1)/3 \rceil$ members of P are

correct, so that at most $\lfloor(|P| - 1)/3\rfloor$ parties are corrupted.

The protocol ensures that the following four predicates hold:

Uniqueness If p_i and p_j are correct and V_i^x and V_j^x are defined, then $V_i^x = V_j^x$.

Validity If p_i is correct and V_i^x is defined, then $p_i \in V_i^x$ and for all correct $p_j \in V_i^x$ it holds that V_j^x is (eventually) defined.

Integrity If $p_i \in V^x \backslash V^{x+1}$, then *faulty*$(p_i)$ held at some correct $p_j \in V^x$, and if $p_i \in V^{x+1} \backslash V^x$, then *correct*$(p_i)$ held at some correct $p_j \in V^x$.

Liveness If there is a correct $p_i \in V^x$ such that *correct*(p_i) holds at $\lceil(2|V^x| + 1)/3\rceil$ correct members of V^x, and a process $p_j \in V^x$ or a process $p_k \notin V^x$ such that *faulty*(p_j) holds at $\lfloor(|V^x| - 1)/3\rfloor$ correct members of V^x or *correct*(p_k) holds at $\lfloor(|V^x| - 1)/3\rfloor$ correct members of V^x, then eventually V^{x+1} is defined.

8.3.1 Informal Description

A total order is assumed to exist on the parties. The party with the highest rank acts as the manager, who is responsible for proposing updates to the group view. When a party suspects another party of being faulty, it reports this to the manager. Once the manager receives $\lfloor(|V^x| - 1)/3\rfloor + 1$ requests to remove party p, it knows that at least one honest party accuses p of being faulty. The manager then sends a **suggest** message to every party, upon which every honest party responds with a **ack** message. When the manager receives $\lceil(2|V^x| + 1)/3\rceil$ **ack** messages, it sends a **proposal** message to everyone. Every party responds with a **ready** message, and after receiving $\lceil(2|V^x| + 1)/3\rceil$ **ready** messages the manager sends a **commit** message, upon which the new view is formed.

If the manager is accused of being faulty, a party sends a **deputy** message to the party with the highest rank, whom is not accused of being faulty. When a party receives $\lfloor(|V^x| - 1)/3\rfloor + 1$ **deputy** messages, it sends a **query** message to every party, thereby seizing the manager role. Each party responds to this with a **last** message, possibly containing the last proposal sent by the previous manager. The deputy sends a **suggest-last** message, upon which each party forms the new view by removing the party suggested by the party with the lowest rank, higher than the deputy.

A cheating manager may try to convince one party of one update to the groupview, while having another party forming another view. Before a manager can propose a party to be removed, it has to accumulate $\lfloor(|V^x| - 1)/3\rfloor + 1$ signed requests to remove some party p, so at least one honest party wants p removed. The manager then forms a proposal, by accumulating $\lceil(2|V^x| + 1)/3\rceil$ signed **ack** responses. A variable ProtocolState is used in such a way that once a party signed a **ack** response for some party p intended to be used by a manager or deputy p_d, it refuses to sign **ack** responses for other parties. If that party is requested to sign a **ack** response for another party p' by another deputy, it even refuses to do so when the rank of p' is higher than the rank of p. Since each honest party only supports one proposal by signing only one **ack** response, and since a proposal needs $\lceil(2|V^x| + 1)/3\rceil$ of these responses, a manager can form at most one proposal.

It may seem that if a party receives a correct **proposal** message, it has enough information to update the current view. Only one correct **proposal** message can be formed, and if that party broadcasts

this message to every other party, and every other party broadcasts it again, ensuring that each party receives the **proposal**. However, this does not work, since before the **proposal** message would arive at every party, a deputy may be chosen to remove the manager. Then, the situation could occur that some parties remove the manager, and other parties follow the **proposal** message. To prevent this situation, the manager first sends a **proposal**, receives the **ready** responses from the other parties, and combines those **ready** messages into a **commit** message. Now if some party receives a **commit** message while a deputy tries to remove the faulty manager, the deputy receives the **proposal** message with its **query** message, and then follows that **proposal** instead of removing the manager. Agreement of the group views is now maintained.

8.3.2 Interface of the Protocol

The protocol provides a set of group views V^x for $x \geq 1$, where either V^x is undefined or $V^x \subseteq P$.

Let V^x be the current group view.
When a party $p_i \in V^x$ suspects another party $p_j \in V^x$ to be faulty, it sends

(**in**, **faulty**, p_j, x)

When a party $p_i \in V^x$ wants another party $p_j \notin V^x$ to join the group, it sends

(**in**, **correct**, p_j, x)

When a new group view V^{x+1}, is delivered, x is increased such that V^x is the new group view. The following message is received

(**out**, **view**, V^x)

This message is received even when the party executing this message is removed from the group. When the party becomes a group member again, the according view V^x is sent with (**out**, **view**, V^x).

A message not defined in the original paper, but needed by the [Rei94] protocol, is a message to enable additions to view x. When no correct party $\in V^x$ sends (**in**, **adds**, x) before V^{x+1} is generated, $V^x \supseteq V^{x+1}$.

(**in**, **adds**, x)

This message is used in the reliable multicast protocol, to prevent that no progress is made when there are several corrupt parties. If several corrupt parties need to be removed in order to output an r-mcast of the next view, additions to the current view are put on hold until that view is really delivered. Otherwise, two corrupt parties could be added and removed over and over again without making progress.

8.3.3 The Secure Group Membership Protocol SGM

The protocol presented in figures 12, 13, and 14 is a translation of the protocol presented in appendix A of [Rei96]. Although the latter protocol has a formal representation, with the notation used in [Rei96] it is not easy to make a nice translation to an implementation. An implementation acts on received messages, it does not check large predicates in a loop, like the specification in [Rei96] does. Rather, it executes statements as a consequence of incoming messages. There is still a correspondence with the original protocol: When a certain block of statements is executed, the corresponding predicate

of the protocol must hold, and when a predicate of the protocol is true, the corresponding block of statements must be executed. Therefore, the protocol is adapted so that it is message-driven: instead of evaluating predicates, it updates its state and sends messages when it receives a message.

We show a small part of the protocol described in [Rei96], with which we illustrate how the whole protocol is translated:

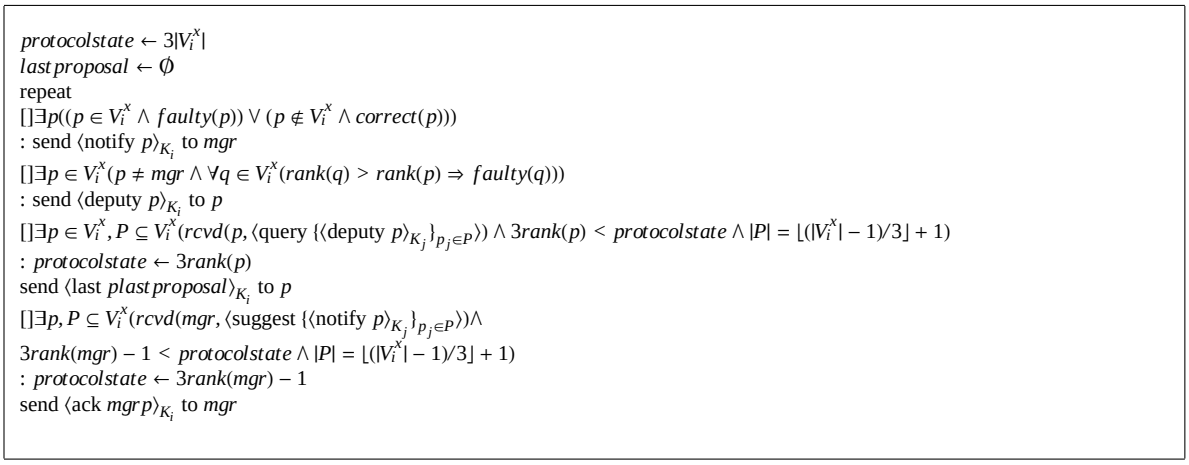

Figure 11: Part of the original Group Membership Protocol

It is not difficult to argue that the translation from the protocol of [Rei96] to the protocol presented in figures 12, 13, and 14 is valid: the predicates in the protocol of [Rei96] almost always contain some form of *rcvd(p, …)*, which correspond to our UPON RECEIVING MESSAGE clauses. This translates directly to a message-driven approach. This is not enough for correctness: it also has to be checked that no predicate becomes true by other events. Luckily, this is the case, as can easily be verified manually. The only non-trivial manner in which predicates in [Rei96] can become true is by the variable ProtocolState. ProtocolState is monotonically decreasing, and in the predicates it is only used in the form $x <$ ProtocolState, so once a predicate is false because the part containing the comparison to ProtocolState is false, the predicate cannot become true anymore. We therefore do not need to check if clauses need to be executed when we change the value of ProtocolState.

The SGM-View Protocol The Secure Group Membership protocol uses the SGM-View protocol as a subprotocol to manage one particular view. SGM starts SGM-View with V^x as parameter. If the current view is changed, SGM-View exits and returns view V^{x+1} to the Secure Group Membership protocol, which starts this View protocol with view V^{x+1} as a parameter.

Each signature created in this protocol is assumed to include the view number x. For brevity, this x is omitted.

In the following protocol, the following conventions are used: p_t (*this party*) is the party executing the protocol, p_c (*change party*) is the party on which a change request is made, p_s (*sender party*) is the sender of a message, and p_m (*party manager*) is the party with the manager role.

Protocol SGM-View (member role) for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **SGM-View**, V^x):
    LastProposal := $\emptyset$
    $V := V^x$
    $p_m := p \in V$ where $\neg(\exists q \in V :: rank(q) > rank(p))$
    ProtocolState := $3|V|$
    MDState := begin

UPON RECEIVING MESSAGE (**in**, **faulty**, p_c):
    send (**notify**, p_c, $\langle p_t$, **notify**, $p_c\rangle_{K_t}$) to p_m
    start timer (**remove-manager**)
    if $\neg\exists p \in V : rank(p) < rank(p_c) : correct(p)$ **then**
        Let $p_d \in V$ such that $correct(p_d) \wedge (\forall p_i \in V : rank(p_i) > rank(p_d) : faulty(p_i))$
        send (**deputy**, $\langle$**deputy**, $p_d\rangle_{K_t}$) to p_d

ON TIMEOUT (**remove-manager**)
    Let $p_d \in V$ such that $correct(p_d) \wedge (\forall p_i \in V : rank(p_i) > rank(p_d) : faulty(p_i))$
    send (**in**, **faulty**, p_d)

UPON RECEIVING MESSAGE (**in**, **correct**, p_c):
    send (**notify**, p_c, $\langle p_t$, **notify**, $p_c\rangle_{K_t}$) to p_m
    start timer (**remove-manager**)

UPON RECEIVING MESSAGE (**suggest**, p_c, NotifySet) FROM p_s:
    if $p_s = p_m \wedge 3\ rank(p_m) - 1 <$ ProtocolState $\wedge$|NotifySet| $= \lfloor(|V| - 1)/3\rfloor + 1$ **then**
        ProtocolState := $3\ rank(p_m) - 1$
        send (**ack**, p_c, $\langle p_m$, **ack**, $p_c\rangle_{K_t}$) to p_m

UPON RECEIVING MESSAGE (**proposal**, p_c, AckSet) FROM p_s:
    if $3\ rank(p_s) - 2 <$ ProtocolState $\wedge$|AckSet| $= \lceil(2|V| + 1)/3\rceil$ **then**
        ProtocolState := $3\ rank(p_s) - 2$
        LastProposal := $\langle p_s, p_c,$ AckSet$\rangle$
        send (**ready**, p_c, $\langle p_s$, **ready**, $p_c\rangle_{K_t}$) to p_s

UPON RECEIVING MESSAGE (**commit**, p_c, p_d, ReadySet) FROM p_s:
    if |ReadySet| $= \lceil(2|V| + 1)/3\rceil$ **then**
        if $rank(p_t) \leq \lceil(2|V| + 1)/3\rceil$ **then**
            send (**commit**, p_c, p_d, ReadySet) to p where $rank(p_t) < rank(p) < rank(p_t) + \lfloor(|V| - 1)/3\rfloor + 1$

        else
            send (**commit**, p_c, p_d, ReadySet) to p where $0 < rank(p) < \lfloor(|V| - 1)/3\rfloor + 1 - (|V| - rank(p_t))$

        send (**out**, **view**, p_c, p_d, ReadySet)
        send (**out**, **halt**)

UPON RECEIVING MESSAGE (**query**, DeputySet) FROM p_s:
    if $3\ rank(p_s) - 2 <$ ProtocolState $\wedge$|DeputySet| $= \lfloor(|V| - 1)/3\rfloor + 1$ **then**
        ProtocolState := $3\ rank(p_s)$
        send (**last**, LastProposal, $\langle$**last**, p_s, LastProposal$\rangle_{K_t}$) to p_s

UPON RECEIVING MESSAGE (**suggest-last**, LastSet) FROM p_s:
    if $3\ rank(p_s) - 1 <$ ProtocolState $\wedge$|LastSet| $= \lceil(2|V| + 1)/3\rceil$ **then**
        LowestRank := $|V| + 1$
        LowestUpdate := p_m
        for each LastProposal $\in$ LastSet **do**
            split LastProposal in $\langle p_d, p_c,$ AckSet$\rangle$
            if $rank(p_s) < rank(p_d) <$ LowestRank $\wedge$|AckSet| $= \lceil(2|V| + 1)/3\rceil$ **then**
                LowestRank := $rank(p_d)$
                LowestUpdate := p_c

        ProtocolState := $3\ rank(p_s)$ - 1
        send (**ack**, LowestUpdate, $\langle p_t$, **ack**, LowestUpdate$\rangle_{K_t}$) to p_s

Figure 12: Secure Group Membership View

Protocol SGM-View (manager/deputy role) for party p_t

UPON RECEIVING MESSAGE (**notify**, p_c, $\langle p_s$, **notify**, $p_c\rangle_{K_s}$) FROM p_s:
  $\text{NotifySet}_c = \text{NotifySet}_c \cup \{\langle s, \langle p_s$, **notify**, $p_c\rangle_{K_s}\rangle\}$
  if MDState = begin $\wedge|\text{NotifySet}_c| = \lfloor(|V| - 1)/3\rfloor + 1$ **then**
    send (**suggest**, p_c, NotifySet_c) to each $p \in V$
    MDState := sent-suggest

UPON RECEIVING MESSAGE (**ack**, p_c, $\langle p_t$, **ack**, $p_c\rangle_{K_s}$) FROM p_s:
  $\text{AckSet}_c = \text{AckSet}_c \cup \{\langle s, \langle p_t$, **ack**, $p_c\rangle_{K_s}\rangle\}$
  if MDState = sent-suggest $\wedge|\ \text{AckSet}_c| = \lceil(2|V| + 1)/3\rceil$ **then**
    send (**proposal**, p_c, AckSet_c) to each $p \in V$
    MDState := sent-proposal

UPON RECEIVING MESSAGE (**ready**, p_c, $\langle p_t$, **ready**, $p_c\rangle_{K_s}$) FROM p_s:
  $\text{ReadySet}_c = \text{ReadySet}_c \cup \{\langle s, \langle p_t$, **ready**, $p_c\rangle_{K_s}\rangle\}$
  if MDState = sent-proposal $\wedge|\ \text{ReadySet}_c| = \lceil(2|V| + 1)/3\rceil$ **then**
    broadcast (**commit**, p_c, p_t, ReadySet_c) to each $p \in V$ by sending this message to self

UPON RECEIVING MESSAGE (**deputy**, $\langle$**deputy**, $p_t\rangle_{K_s}$) FROM p_s:
  $\text{DeputySet} = \text{DeputySet} \cup \{\langle s, \langle$**deputy**, $p_t\rangle_{K_s}\rangle\}$
  if MDState = begin $\wedge|\text{DeputySet}| = \lfloor(|V| - 1)/3\rfloor + 1$ **then**
    send (**query**, DeputySet) to each $p \in V$
    MDState := sent-query

UPON RECEIVING MESSAGE (**last**, LastProposal, $\langle$**last**, p_t, LastProposal$\rangle_{K_s}$) FROM p_s:
  $\text{LastSet[LastProposal]} = \text{LastSet[LastProposal]} \cup \{\langle s$, LastProposal, $\langle$**last**, p_t, LastProposal $\rangle_{K_s}\rangle\}$
  if MDState = sent-query $\wedge|\ \text{LastSet[LastProposal]}| = \lceil(2|V| + 1)/3\rceil$ **then**
    send (**suggest-last**, LastProposal, LastSet[LastProposal])
    MDState := sent-suggest

Figure 13: Secure Group Membership View (Continued)

The SGM Protocol The protocol presented in figure 14 is a slightly modified version of the protocol presented in figure 7 of appendix A of [Rei96]. Section 4.3 of [Rei96] gives three options on how the first group view is obtained. We have chosen to fix the participants with which the bulletin board starts.

Protocol SGM for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **SGM**, Parties):
    $x := 1$
    History := $\emptyset$
    MessageQueue := $\emptyset$
    Faulty := $\emptyset$
    P := Parties
    $V[x] := P$
    View := **new** (**in**, **open**, **SGM-View**, $V[x]$)
    Adds := **false**

UPON RECEIVING MESSAGE (**in**, **faulty**, p_c, x'):
    Faulty := Faulty $\cup\{p_c\}$
    send (**in**, **faulty**, p_c) to View

UPON RECEIVING MESSAGE (**in**, **correct**, p_c, x'):
    Correct := Correct $\cup\{p_c\}$
    if Adds **and** $x' = x$ **then**
        send (**in**, **correct**, p_c) to View

UPON RECEIVING MESSAGE (**in**, **adds**, x'):
    if $x' = x$ **then**
        Adds := **true**
        for each $p_c \in$ Correct **do**
            send (**in**, **correct**, p_c) to View

UPON RECEIVING MESSAGE (**out**, **view**, p_c, p_d, ReadySet) FROM View:
    $x := x + 1$
    History := History $\cup\langle x, p_c, p_d,$ ReadySet$\rangle$
    if $p_c \notin V[x-1]$ **then**
        send (**history**, History) to p_c
        $V[x] := V[x-1] \cap \{p_c\}$
        Correct := Correct $\backslash\{p_c\}$

    else
        $V[x] := V[x-1]\backslash\{p_c\}$
        Faulty := Faulty $\backslash\{p_c\}$

    Adds := **false**
    View := **new** (**in**, **open**, **SGM-View**, $V[x]$)
    send (**in**, **faulty**, p) to View **for each** $p \in$ Faulty
    send (**out**, **view**, x, $V[x]$)
    while $\langle$type, parameters, $p_s\rangle$ = *head*(MessageQueue[x]) **do**
        pop *head*(MessageQueue[x])
        send (**type**, parameters) to View with sender p_s

UPON RECEIVING MESSAGE (**out**, **history**, History, p_c) FROM View:
    send (**history**, History) to p_c

UPON RECEIVING MESSAGE (**history**, History) FROM p_s:
    while $\langle x, p_c,$ ReadySet$\rangle \in$ History **do**
        let View send a (**commit**, $\{\langle p_c,$ ReadySet$\rangle\}$) to each $p \in V[x] \cup \{p_t\}$

UPON RECEIVING MESSAGE (**type**, parameters, p) FROM View:
    send (**view-message**, x, type, parameters) to p

UPON RECEIVING MESSAGE (**view-message**, x', type, parameters) FROM p_s:
    if $x = x'$ **then**
        send (**type**, parameters) to View with sender p_s

    if $x < x'$ **then**
        enqueue $\langle$type, parameters, $p_s\rangle$ on MessageQueue[x']

Figure 14: Secure Group Membership View (Continued)

8.4 The Echo Multicast Protocol

The echo multicast protocol is the basis of the reliable multicast protocol. In the absence of membership changes, a reliable multicast reduces to a single echo multicast. This protocol ensures that the l-th echo multicasts from p for view x at any two honest parties are the same.

Under the assumption that at most $\lfloor(|V^x| - 1)/3\rfloor$ parties in V^x are corrupt, the following statements hold:

1. If p is honest and some honest process sends (**out**, **e-mcast**, p, x, m), then p sent (**in**, **e-mcast**, x, m).
2. If the l-th message of the form (**out**, **e-mcast**, p, x, …) at two honest processes are (**out**, **e-mcast**, p, x, m) and (**out**, **e-mcast**, p, x, m'), then $m = m'$.

8.4.1 Informal Description

Because the echo multicast protocol is quite complicated, we first simplify it by stating that a party p sends at most one message. Party p has to convince every party that he sends the same m to every other party.

Suppose a party p wishes to multicast a single message m. p first tries to obtain $\lceil(2|V^x| + 1)/3\rceil$ signatures binding p's message to m by sending an **init** message containing m. On receipt of a **init** message, a party p' creates a signature on the name of the requesting party and the message. This signature is sent with a **echo** message as a response to the **init** message. After party p receives enough correct **echo** replies, it sends a **commit** message containing the message m and the signatures, indicating as proof that m is the message sent to everyone. Once a party receives this **commit** message, it verifies the signatures, and delivers message m.

Why are $\lceil(2|V^x| + 1)/3\rceil$ signatures enough to convince other parties that p has sent m to everyone? Every honest party only ever creates one signature for each party. A party that tries to cheat by trying to convince one party that m is p's message, and another party that m' is p's message, will have to obtain two sets of signatures. Corrupt parties might be willing to create multiple signatures for party p, so that p can send different messages to different parties. Assume that a party has obtained two sets of signatures. These two sets are each created by $\lceil(2|V^x| + 1)/3\rceil$ parties. That means that there are at least $\lfloor(|V^x| - 1)/3 + 1\rfloor$ parties that created two signatures for the same m. This is in contradiction with our assumption that there are at most $\lfloor(|V^x| - 1)/3\rfloor$ corrupt parties: if p obtained two sets of signatures, one for m and one for m', then there are at least $\lfloor(|V^x| - 1)/3 + 1\rfloor$ corrupt parties, which is one more than we allowed.

We now gradually expand this simplified protocol. First, we explain how parties can send multiple messages: if p sends a message, it stores its index l. Each party also stores the number of messages received from other parties in a variable l_p. When p requests signatures on a particular message m on index l, each party creates the signature on p, message m, and index l. This way, p's l-th message is unique.

The index l is also used to maintain the correct order of all messages sent by a party. Upon receiving a **commit** message for a message m and index l, the message and index are stored in a 'Commits' set, and a variable $c[p']$ is used to indicate the index of the next message to be delivered from party p'.

After delivering the message, it stays in the Commits set until the messages is *stable*. A message is stable once every party has added the message to its own Commits set. Each party periodically notifies the other parties of the messages in Commits by multicasting a **counters** message containing the $c[p]$ values. Each party records these values in a set $c_p[p']$. If a party has a message m from p' with index l in its Commits set, and $c_p[p']$ is at least l for each p, then it concludes that every party has added m to its Commits set, so it delivers m to a higher protocol layer and removes it from the Commits set.

If a message m stays in the Commits set for too long, a party sends the message to the parties which have not delivered m yet, in order to try to make it stable. If a second timeout expires, parties who still have not delivered m are unresponsive, and are voted out of the current view.

This informal description assumes that the group view does not change. If a party is voted out, a new group view is formed, and parties have to send their messages to a different set of parties. In effect, each party indexes its variables by the group view number x, so it uses $l[x]$ as the index of its messages, $c[p, x]$ as the index of the next message to deliver from party p, etc.

Instead of acquiring $\lceil (2|V^x| + 1)/3 \rceil$ signatures, a party could also generate a threshold signature by acquiring signature shares. Two threshold signatures schemes are specified, and the differences in notational complexity are profound. The first threshold signature scheme specified is the adapted version of Pedersen's Threshold Signature Scheme, described in section 5.6. First, each party is requested to choose a secret k_i and reply with g^{k_i}. On receiving $\lceil (2|V^x| + 1)/3 \rceil$ of those g^{k_i} values, they are combined into a value g^k. Then, the $\lceil (2|V^x| + 1)/3 \rceil$ parties participating in the g^k are requested to contribute a signature share based on their g^{k_i} value. If they all respond within a timeout, a threshold signature is created, and used to commit the message m. If they do not respond within a timeout, the protocol reverts to the normal version of the protocol, where care is taken so that parties still cannot commit two different messages to the same index l.

The second threshold signature is much simpler. Since a non-interactive threshold signature scheme is used, a party can just request $\lceil (2|V^x| + 1)/3 \rceil$ signature shares on a message m, combine those shares into one signature, and commit the message m.

8.4.2 Interface of the Protocol

To echo-multicast a message m in view x, send the following message:

(**in**, **e-mcast**, x, m)

When an echo-multicast message m is received in view x from party p_i, the following message is received:

(**out**, **e-mcast**, p_i, x, m)

When the echo-multicast message m is received in view x from party p_i by all members of V^x, the following message is received:

(**out**, **e-mcast-stable**, p_i, x, m)

8.4.3 Translating the Protocol

Translating the echo multicast protocol into the specification presented in figures 17-21 was more difficult than translating the secure group membership protocol. The protocol described in [Rei94] uses lots of text to describe the functionality, and the parts specifying how to achieve stability are stated very informally. We show a small piece to illustrate what we had to translate:

1. If *E-mcast*(x, m) is executed at some $p \in V^x$, p sends

   $$\langle \text{init} : x, f(m) \rangle$$

   to each member of V^x. (A message sent by a process to itself is received immediately when it is sent.) This process is called the multicast *initiator*.

2. If p_j receives $\langle$init: $x, d\rangle$from some $p \in V^x$ and this is the l-th message of the form $\langle$init: $x, *\rangle$that p_j has received from p, then p_j sends

   $$\langle \text{echo} : p, x, l, d \rangle_{K_j}$$

   to p.

3. Once the initator p hasreceived a set of echoes $\{\langle \text{echo} : p, x, l, f(m)\rangle_{K_j}\}_{p_j \in P}$ for some l and some $P \subseteq V^x$ where $|P| = \lceil (2|V^x| + 1)/3 \rceil$, it sends

   $$\langle \text{commit} : p, x, m, \{\langle \text{echo} : p, x, l, f(m)\rangle_{K_j}\}_{p_j \in P} \rangle$$

   to each member of V^x.

Figure 15: Part of the original Echo Multicast Protocol

And we show a typical passage from the section describing how to achieve stability:

So that processes can tell when echo multicasts are stable, each $q \in V^x$ periodically echo multicasts its set $\{c_i^x\}_{p_i \in V^x}$ of counter values to view x. $\langle \ldots \rangle$A process q knows that the l-th echo multicast from p_i to view x is stable if q has E-delivered a message for view x containing a counter value $c_i^x \geq l$ from each member of V^x. An honest process does not permit a multicast to remain unstable for longer than a prespecified timeout duration. That is, if a process q retains (1) in $commits^x$ beyond some timeout duration after executing *E-deliver*(p_i, x, m), it attempts to make this multicast stable by sending (1) to each $r \in V^x$ from which q has not E-delivered a counter value $c_i^x \geq l$.

Figure 16: Another part of the original Echo Multicast Protocol

While stated less formally than the secure group membership protocol, it is easier to translate this protocol, because this protocol is already stated in a more reactive way: it explicitly states that if a party p_j receives a message of the form $\langle$init: $x, f(m)\rangle$, then it does this and that. This translates nicely into our formalism. We show how we translated the part of the protocol indexed with 1. It is text of the form 'If E-mcast(x, m) is executed, then send (**init**, x, $f(m)$) to every party in V^x'. This is translated into an **in** message, producing the following code:

```
UPON RECEIVING MESSAGE (in, e-mcast, x, m):
    for each p ∈ SGM.V^x do
        send (init, x, f(m)) to p
```

Later on in the protocol, we notice that we should be able to retrieve the message m by its hash, and that we must record its index. We therefore introduce a variable l which records its index, and we introduce a map lm which, when indexed by the hash of a message, contains both the message and its index. We add the code:

UPON RECEIVING MESSAGE (**in**, **e-mcast**, x, m):
    $l[x] := l[x] + 1$
    $lm[\, f(m), x] := \langle l[x], m \rangle$
    for each $p \in SGM.V^x$ **do**
        send (**init**, x, $f(m)$) to p

Then, we add code to facilitate for our threshold signature schemes, but we omit that trivial code.

It was more difficult to translate the text about stability, because in its informality the text that describes the protocol is mixed with text that is merely flavour. It does however emphasize the importance of first translating the original protocols into a set of coherent formal specifications.

8.4.4 Variables used in the Protocol

Let x be a view number. Then $l[x]$ denotes the number of messages that this party has multicast in view x. $lm[x]$ is a set of pairs of messages and indexes. When a party sends the l-th message m, then $\langle m, l \rangle$ is added to $lm[x]$. Because this set of pairs is used as a function from messages to indexes, a special notation is used to insert and find elements. To insert that pair into $lm[x]$, the notation $lm[l, x] := m$ is used. To find the index of message m, $lm[m, x]$ is used. An implementation must check if $lm[x]$ indeed contains a message m, and otherwise ignore the received message.

$c[p, x]$ contains for each party p the index up until which messages received from p are delivered. This list is used to preserve the order in which messages received from p are delivered. If a message m received from p with a higher index than $c[p, x] + 1$ is received, it is enqueued until previous messages from p are received.

$c_q[p, x]$ is the list $c[p, x]$ at party q, as broadcasted by q. This list is used to determine if every party has broadcast every message.

$s[p, x]$ denotes for every party p the index of the message that is stable. $(\forall p \in V^x :: s[p, x] = (\downarrow q \in V^x :: c_q[p, x]))$ (Here, the notation $(\downarrow a :: b)$ means the minimum over the values b in the domain a). This variable is not strictly necessary, but simplifies the notation.

8.4.5 The Echo Multicast Protocol EMP

Protocol EMP for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **EMP**, SecureGroupMembershipProtocol):
SGM = SecureGroupMembershipProtocol
$l[SGM.x] := 0$
$lm[SGM.x] := \emptyset$
for each $p \in SGM.V^{SGM.x}$ **do**
$l_p[SGM.x] := 0$
$c[p, SGM.x] := 0$
$c_{p_i}[p, x] := 0$ **for each** $p_i \in SGM.V^x$
$s[p, x] := 0$

UPON RECEIVING MESSAGE (**in**, **e-mcast**, x, m):
$l[x] := l[x] + 1$
$lm[l[x], x] := \langle l[x], m \rangle$
for each $p \in SGM.V^x$ **do**
if non-interactive threshold signatures are available **then**
send (**init-ni-threshold**, x, $f(m)$) to p

else if threshold signatures are available **then**
send (**request-threshold**, x, $f(m)$) to p

else
send (**init**, x, $f(m)$) to p

UPON RECEIVING MESSAGE (**init**, x, d) FROM p_s:
verify $p_s \in SGM.V^x$
$l_{p_s}[x] := l_{p_s}[x] + 1$
send (**echo**, $x, l_{p_s}[x], \langle p_s, x, l_{p_s}[x], d \rangle_{K_t}$) to p_s

UPON RECEIVING MESSAGE (**echo**, x, l, $\langle p_t, x, lm[l, x], d \rangle_{K_s}$) FROM p_s:
verify $p_s \in SGM.V^x$
EchoSet[x, l] := EchoSet[x, l] $\cup \{\langle s, \langle p_t, x, lm[l, x], d \rangle_{K_s} \rangle\}$
if |EchoSet[x, l]| = $\lceil (2|V^x| + 1)/3 \rceil$ **then**
for each $p \in SGM.V^x$ **do**
send (**commit**, x, l, $lm[l, x]$, m, EchoSet[x, l]) to p

UPON RECEIVING MESSAGE (**commit**, x, l, m, Signature) FROM p_s:
verify $p_s \in SGM.V^x$
for each $x' \in [x, SGM.x]$ **do**
verify $p_s \in SGM.V^{x'}$

AddToCommit(p_s, x, l, m, Signature)

UPON RECEIVING MESSAGE (**out**, **view**, V):
$l[SGM.x] := 0$
$lm[SGM.x] := \emptyset$
for each $p \in SGM.V^{SGM.x}$ **do**
$l_p[SGM.x] := 0$
$c[p, SGM.x] := 0$
$c_{p_i}[p, x] := 0$ **for each** $p_i \in SGM.V^x$
$s[p, x] := 0$

Figure 17: Echo Multicast Protocol

Protocol EMP (threshold signatures for the optimistic case) for party p_t

UPON RECEIVING MESSAGE (**request-threshold**, x, d) FROM p_s:
  choose k_t
  send (**reply-threshold**, x, d, g^{k_t}) to p_s

UPON RECEIVING MESSAGE (**reply-threshold**, x, d, a_s) FROM p_s:
  verify $p_s \in SGM.V^x$
  if $|\text{Threshold}[x, d]| < \lceil (2|SGM.P| + 1)/3 \rceil$ **then**
    $\text{Threshold}[x, d] := \text{Threshold}[x, d] \cup \{\langle s, a_s \rangle\}$
    if $|\text{Threshold}[x, d]| = \lceil (2|SGM.P| + 1)/3 \rceil$ **then**
      $l[x] := l[x] + 1$
      combine $\{a_i\}_{\langle i, a_i \rangle \in \text{Threshold}[x,d]}$ into a
      send (**init-threshold**, x, d, a) to each p_i where $\langle i, a_i \rangle \in \text{Threshold}[x, d]$
      send (**unused-threshold**, x, d, a) to each p_i where $\langle i, a_i \rangle \notin \text{Threshold}[x, d]$
      $lm[d, x] := l[x]$
      start timer (**threshold**, x, d, l)

ON TIMEOUT (**threshold**, x, d, l)
  send (**init-no-threshold**, x, d, l) to each $p \in SGM.V^x$

UPON RECEIVING MESSAGE (**init-threshold**, x, d, a) FROM p_s:
  verify $p_s \in SGM.V^x$
  send (**echo-threshold**, x, d, $\langle p_s, x, l_{p_s}[x], d \rangle_{K_t(a)}$) to p_s
  $tl := tl \cup \{\langle d, l_{p_s}[x], p_s \rangle\}$
  $l_{p_s}[x] := l_{p_s}[x] + 1$

UPON RECEIVING MESSAGE (**unused-threshold**, x, d) FROM p_s:
  verify $p_s \in SGM.V^x$
  $tl := tl \cup \{\langle d, l_{p_s}[x], p_s \rangle\}$
  $l_{p_s}[x] := l_{p_s}[x] + 1$

UPON RECEIVING MESSAGE (**init-no-threshold**, x, d, l) FROM p_s:
  verify $p_s \in SGM.V^x$
  verify $\langle d, l, p_s \rangle \in tl$
  send (**echo**, x, d, $\langle p_s, x, l[x], d \rangle_{K_t}$) to p_s

UPON RECEIVING MESSAGE (**echo-threshold**, x, d, $\langle p_t, x, lm[d, x], d \rangle_{K_s(a)}$) FROM p_s:
  verify $p_s \in SGM.V^x$
  $\text{EchoThresholdSet}[x, d] := \text{EchoThresholdSet}[x, d] \cup \langle s, \langle p_t, x, lm[d, x], d \rangle_{K_s(a)} \rangle$
  if $|\text{EchoThresholdSet}[x, d]| = \lceil (2|SGM.P| + 1)/3 \rceil$ **and** timer is running **then**
    stop timer (**threshold**, x, d, $lm[d, x]$)
    combine the signatures in $\text{EchoThresholdSet}[x, d]$ into $\langle p_t, x, lm[d, x], d \rangle_{K(a)}$
    for each $p \in SGM.V^x$ **do**
      send (**commit**, x, $lm[d, x]$, m, $\langle p_t, x, lm[d, x], d \rangle_{K(a)}$) to p

Figure 18: Echo Multicast Protocol (Continued)

Protocol EMP (non-interactive threshold signatures) for party p_t

UPON RECEIVING MESSAGE (**init-ni-threshold**, x, d) FROM p_s:
  verify $p_s \in SGM.V^x$
  send (**echo-ni-threshold**, x, d, $\langle p_s, x, l_{p_s}[x], d \rangle_{K_t}$) to p_s
  $l_{p_s}[x] := l_{p_s}[x] + 1$

UPON RECEIVING MESSAGE (**echo-ni-threshold**, x, d, sig) FROM p_s:
  verify $p_s \in SGM.V^x$
  if $|\text{NIThreshold}[x, d]| < \lceil (2|SGM.P| + 1)/3 \rceil$ **then**
    $\text{NIThreshold}[x, d] := \text{NIThreshold} \cup \{\langle p_s, sig \rangle\}$
    if $|\text{NIThreshold}[x, d]| = \lceil (2|SGM.P| + 1)/3 \rceil$ **then**
      $l[x] := l[x] + 1$
      combine $\{sig_i\}_{\langle sig_i, p_i \rangle \in}$ $\text{Treshold}_{[x,d]}$ into sig
      for each $p \in SGM.V^x$ **do**
        send (**commit**, x, $lm[d, x]$, m, sig) to p

Figure 19: Echo Multicast Protocol (Continued)

Protocol EMP (stability) for party p_t

PERIODICALLY
send (**in**, **e-mcast**, $\langle$**counters**, $\{c[p,x]\}_{p \in SGM.V^x}\rangle$)

UPON RECEIVING MESSAGE (**out**, **e-mcast**, p_s, $\langle$**counters**, $\{c'[p,x]\}_{p \in SGM.V^x}\rangle$):
$c_{p_s}[p,x] := c'[p,x]$ **for each** $p \in V^x$
for each s **where** $s[p,x] < s \leq (\downarrow q \in V^x :: c_q[p,x])$ **do**
stop timer (**unstable**, p, x, s)
stop timer (**remove-unstable**, p, x, s) if it is running
$s[p,x] := s$
retrieve m and Signature from Commits[x]
Commits[x] := Commits[x] $\setminus \langle p, x, s, m,$ Signature$\rangle\}$
send (**out**, **e-mcast-stable**, p, x, m)

ON TIMEOUT (**unstable**, p, x, l)
retrieve m and Signature from Commits[x]
for each p' **where** $c[p,x] > c_{p'}[p,x]$ **do**
send (**make-stable**, $p, x, c[p,x], m,$ Signature)
start timer (**remove-unstable**, p, x, l)

ON TIMEOUT (**remove-unstable**, p, x, l)
for each p' **where** $c[p,x] > c_{p'}[p,x]$ **do**
send (**in**, **faulty**, p')

UPON RECEIVING MESSAGE (**make-stable**, $p, x, l, m,$ Signature) FROM p_s:
for each $x' \in [x, SGM.x]$ **do**
verify $p \in SGM.V^{x'}$

AddToCommit(p, x, l, m, Signature)

Figure 20: Echo Multicast Protocol (Continued)

Protocol EMP (procedures) for party p_t

PRODUCEDURE AddToCommit(p_s, x, l, m, Signature)
if $c[p_s,x] < l \wedge \langle p_s, x, l, m,$ Signature$\rangle \notin$ Commits[x] **then**
Signature is either $\{\langle i, \langle p_s, x, lm[d,x], d\rangle_{K_i}\rangle\}$ or $\langle p_s, x, l, f(m)\rangle_{K(a)}$ or $\langle p_s, x, l, f(m)\rangle_K$
if Signature = $\{\langle i, \langle p_s, x, lm[d,x], d\rangle_{K_i}\rangle\}$ **then**
verify |Signature| = $\lceil (2|V^x| + 1)/3 \rceil$ and $\langle i, s\rangle \in SGM.V^x$ for all $\langle i, s\rangle \in$ Signature

Commits[x] := Commits[x] $\cup \{\langle p_s, l, m,$ Signature$\rangle\}$
start timer (**unstable**, p_s, x, l)
while $\langle p, c[p,x] + 1, m',$ Signature$\rangle \in$ Commits[x] **do**
send (**out**, **e-mcast**, p, x, m')
$c[p,x] := c[p,x] + 1$

Figure 21: Echo Multicast Protocol (Continued)

8.5 The Reliable Multicast Protocol

The reliable multicast protocol offers functionality to send a message to every member in a group, as determined by a secure group membership protocol, such that each group member has the assurance that every other party has received the same message. The reliable multicast protocol uses the echo multicast protocol. In the absence of membership changes, messages are just forwarded to and from the echo multicast protocol. When a membership change occurs, this protocol ensures that echo multicast messages are not held forever in the queue.

The protocol ensures that these predicates hold:

Integrity For all honest p and m, an honest process sends (**out**, **r-mcast**, p, m) in view x at most the number of times that p sent (**in**, **r-mcast**, m) in view x.

Uniform Agreement If q is an honest member of V^{x+k} and an honest p sends (**out**, **r-mcast**, r, m) in view x, then q sends (**out**, **r-mcast**, r, m) in view x.

Validity-1 If p is an honest member of V^{x+k} for all $k \geq 0$, then p sends (**out**, **r-mcast-view**, V^x).

Validity-2 If p and q are honest members of V^{x+k} for all $k \geq 0$ and p sends (**in**, **r-mcast**, m) in view x, then q sends (**out**, **r-mcast**, p, m) in view x.

8.5.1 Informal Description

In the echo multicast protocol, messages are sent and received in a specific view. If a group membership change occurs, new messages must be sent for the new view, and the messages queue for the old view must be cleaned. After this cleaning, the membership change may be passed on to higher protocol layers. The reliable multicast protocol handles these tasks.

The protocol uses a variable x to denote the latest view number delivered to the higher protocol layers. Most of the time, this x has the same value as the x of the secure group membership protocol, but during membership changes, it 'lags behind'. Another variable OpenView denotes for which views messages are still accepted. Like x, this variable is usually equal to the x of the secure group membership protocol. Each view with an index lower than OpenView is said to be *closed*.

When a message m has to be sent, it is echo multicast in the latest view. If a message m is received, it is first verified that this message is not intended for an already closed view. If it is intended for view x, it is delivered instantly. If it is intended for a later view, it is enqueued.

When a group membership change occurs, each party echo multicasts an **end** message to the old view, signalling that this is the last message it will send in the old view. After receiving an **end** message from every party, each party sends a **flush** message to the new view, containing the Commits set of the echo multicast protocol. This way, agreement is obtained on the messages that have to be delivered in the old view. Also, OpenView is incremented, signalling that no more messages for the old view may be received. Once a **flush** message is received from every party, the group membership change is announced to the higher protocol layers and x is incremented. If a party does not send an **end** or **flush** message in time, it is voted out. When a party is voted out during another group membership change, it is assumed to have sent a **end** message and an empty **flush** message.

8.5.2 Interface of the Protocol

The protocol is initiated with the following message, which takes a secure group membership protocol as a parameter:

(**in**, **open**, **RMP**, SecuregroupMembershipProtocol)

To reliably multicast a message m, send the following message:

(**in**, **r-mcast**, m)

When a reliably multicast message *m* is received, the following message is sent by the protocol:

(**out**, **r-deliver**, *m*)

When a new view with index *x* is received from the secure group membership protocol and all messages sent in the old view are delivered, this message is sent:

(**out**, **r-mcast-view**, *x*)

8.5.3 The Reliable Multicast Protocol RMP

Protocol RMP for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **RMP**, SecureGroupMembershipProtocol):
SGM := SecureGroupMembershipProtocol
EMP := **new** (**in**, **open**, **EMP**, SGM)
$x := SGM.x - 1$
OpenView := $SGM.x$
$id[x] := 0$
$id_p[x] := 0$ **for each** $p \in SGM.V(SGM.x)$
Join()

UPON RECEIVING MESSAGE (**in**, **r-mcast**, m):
send (**in**, **e-mcast**, $SGM.x$, ⟨**r-msg**, $id[SGM.x]$, m⟩)
$id[SGM.x] := id[SGM.x] + 1$

UPON RECEIVING MESSAGE (**out**, **e-mcast-stable**, p, x', ⟨**r-msg**, id', m⟩):
if OpenView $\leq x'$ **and** $id_p[x'] < id'$ **then**
$id_p[x'] = id'$
if $x' = x$ **then**
send (**out**, **r-mcast**, p, m)

if $x' > x$ **then**
enqueue $\langle p, m \rangle$ on Defer(x')

UPON RECEIVING MESSAGE (**in**, **view**, V):
$id[SGM.x] := 0$
$id_p[SGM.x] := 0$ **for each** $p \in SGM.V(SGM.x)$
NotReceivedFlush[$SGM.x$] := $SGM.V[SGM.x - 1] \cap SGM.V[SGM.x]$
if $p_t \notin SGM.V[SGM.x - 1]$ **then**
Join()

else
NotReceivedEnd[$SGM.x - 1$] := $SGM.V[SGM.x - 1] \cap SGM.V[SGM.x]$
for each $x' \in (x, SGM.x)$ **do**
ReceivedEnd($SGM.V[SGM.x - 1] \backslash SGM.V[SGM.x]$, $x' - 1$)
ReceivedFlush($SGM.V[SGM.x - 1] \backslash SGM.V[SGM.x]$, x')

send (**in**, **e-mcast**, $SGM.x - 1$, ⟨**end**⟩)
start timer (**end**, $SGM.x - 1$)

ON TIMEOUT (**end**, x')
for each $p \in$ NotReceivedEnd[x'] **do**
send (**in**, **faulty**, p, $x' + 1$)

UPON RECEIVING MESSAGE (**out**, **e-mcast**, p_s, x', ⟨**end**⟩):
ReceivedEnd($\{p_s\}$, x')

ON TIMEOUT (**flush**, x')
for each $p \in$ NotReceivedFlush[x'] **do**
send (**in**, **faulty**, p, x')

UPON RECEIVING MESSAGE (**out**, **e-mcast**, p_s, x', ⟨**flush**, Commits⟩):
verify $p_s \in$ NotReceivedFlush[x']
if $x' = 1$ **or** $p_t \in SGM.V[x' - 1]$ **then**
for each $\langle p, x' - 1, l, m,$ Signature$\rangle \in$ Commits **do**
EMP.AddToCommit(p, $x' - 1$, l, m, Signature)

ReceivedFlush($\{p_s\}$, x')

Figure 22: Reliable Multicast Protocol

Protocol RMP (procedures) for party p_t

PRODUCEDURE ReceivedEnd(P, x')
    NotReceivedEnd[x'] := NotReceivedEnd[x'] \P
    while NotReceivedEnd[OpenView] = $\emptyset$ **do**
        stop timer (**end**, OpenView)
        OpenView := OpenView + 1
        send (**in**, **e-mcast**, OpenView, ⟨**flush**, EMP.Commits[OpenView-1])
        start timer (**flush**, OpenView)

PRODUCEDURE ReceivedFlush(P, x')
    NotReceivedFlush[x'] := NotReceivedFlush[x'] \P
    while NotReceivedFlush[x + 1] = $\emptyset$ **do**
        stop timer (**flush**, x)
        $x := x + 1$
        send (**in**, **adds**, x)
        send (**out**, **r-mcast-view**, x)
        while Defer[x] ≠ $\emptyset$ **do**
            dequeue *head*(Defer[x]) as ⟨p, m⟩
            send (**out**, **r-mcast**, p, m)

PRODUCEDURE Join()
    send (**in**, **e-mcast**, x + 1, ⟨**flush**, $\emptyset$⟩)
    if x = 0 **then**
        start timer (**flush**, x + 1)

Figure 23: Reliable Multicast Protocol (Continued)

8.6 The Atomic Multicast Protocol

This protocol is used together with a secure membership protocol, to send and receive messages to all members of the current group view. The order in which messages are received by each party is the same at each party, hence the name 'atomic'. The protocol uses the reliable multicast protocol to reliably send each message to all members.

The protocol ensures that these predicates hold:

Integrity For all honest p and m, an honest process sends (**out**, **a-mcast**, p, m) in view x at most the number of times that p sent (**in**, **a-mcast**, m) in view x.

Uniform Agreement If q is an honest member of V^{x+k} for all $k \geq 0$ and an honest p sends (**out**, **a-mcast**, r, m) in view x, then q sends (**out**, **a-mcast**, r, m) in view x.

Validity-1 If p is an honest member of V^{x+k} for all $k \geq 0$, then p sends (**out**, **a-mcast-view**, V^x).

Validity-2 If p and q are honest members of V^{x+k} for all $k \geq 0$ and p sends (**in**, **a-mcast**, m) in view x, then q sends (**out**, **a-mcast**, p, m) in view x.

Order If q is an honest members of V^{x+k} for all $k \geq 0$ and an honest p sends (**out**, **a-mcast**, r, m) before (**out**, **a-mcast**, r', m') in view x, then q sends (**out**, **a-mcast**, r, m) before (**out**, **a-mcast**, r', m') in view x.

8.6.1 Informal Description

??

The reliable multicast protocol is used to multicast messages. Once a message m is received from party p it is added to the queue Pending_p. A designated group member, the *sequencer*, periodically sends an order message denoting the order in which the messages are to be delivered. The sequencer keeps a queue Senders, which is a sequence of parties. Upon receiving this sequence, each party takes the first party p of this sequence, and the first message of queue Pending_p, and delivers that message. Then, it takes the second party of the sequence, and delivers the appropriate message, etc. This way, each party delivers all messages in the same order.

If a new view is delivered, the atomic multicast protocol does not wait for an order message, but deterministically chooses an order in which to deliver the queued messages.

8.6.2 The Interface

The protocol is initiated with the following message, which takes a secure group membership protocol as a parameter:

(**in**, **open**, **AMP**, SecureGroupMembershipProtocol)

To atomically multicast a message m, send the following message:

(**in**, **a-mcast**, m)

When an atomically multicast message m is received from party p_i, the following message is sent by the protocol:

(**out**, **a-mcast**, p_i, m)

When a new view with index x is received from the secure group membership protocol and all messages sent in the old view are delivered, this message is sent:

(**out**, **a-mcast-view**, x)

8.6.3 The Atomic Multicast Protocol AMP

Protocol AMP for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **AMP**, SecureGroupMembershipProtocol):
    SGM := SecureGroupMembershipProtocol
    RMP := **new** (**in**, **open**, **RMP**, SGM)

UPON RECEIVING MESSAGE (**in**, **a-mcast**, m):
    send (**in**, **r-mcast**, ⟨**a-msg**, m⟩)

UPON RECEIVING MESSAGE (**out**, **r-mcast**, p_s, ⟨**a-msg**, m⟩):
    Enqueue m on $\text{Pending}_s^{RMP.x}$
    if $\text{Seq}(RMP.x) = p_t$ **then**
        $\text{Senders}^{RMP.x} := \text{Senders}^{RMP.x} \parallel p_t$

    start timer (**sequence**, p_s, m, $RMP.x$)

ON TIMEOUT (**sequence**, p_s, m, x)
    if $x = SGM.x$ **then**
        send (**in**, **faulty**, Seq(x), x)

PERIODICALLY
    if $\text{Seq}(RMP.x) = p_t$ **then**
        send (**in**, **r-mcast**, ⟨**order**, $\text{Senders}^{RMP.x}$⟩)
        $\text{Senders}^{RMP.x} := \emptyset$

UPON RECEIVING MESSAGE (**out**, **r-mcast**, Seq($RMP.x$), ⟨**order**, Senders⟩):
    while Senders $\neq \emptyset$ **do**
        dequeue *head*(Senders) as p_i
        if $p_i \in SGM.V(RMP.x)$ **then**
            enqueue p_i on $\text{Order}^{RMP.x}$

    while $\text{Order}^{RMP.x} \neq \emptyset$ **and** $\text{Pending}_{head(\text{Order}^{RMP.x})}^{RMP.x} \neq \emptyset$ **do**
        dequeue *head*($\text{Order}^{RMP.x}$) as p_s
        dequeue *head*($\text{Pending}_{p_s}^{RMP.x}$) as m
        send (**out**, **a-mcast**, p_s, m)
        stop timer (**sequence**, p_s, m, $RMP.x$)

UPON RECEIVING MESSAGE (**out**, **r-mcast-view**, x):
    for each $p \in_{deterministically} SGM.V(x-1)$ **do**
        while $\text{Pending}_p \neq \emptyset$ **do**
            dequeue *head*(Pending_p) as m
            send (**out**, **a-mcast**, p, m)
            stop timer (**sequence**, p, m, $x-1$)

    send (**out**, **a-mcast-view**, x)

Figure 24: Atomic Multicast Protocol

8.7 The Synchronized Atomic Multicast Protocol

The atomic multicast protocol ensures that received messages are delivered in the same order, but members that have been offline for a moment do not receive all messages. The protocol described in this protocol enables parties to synchronize the messages, so that all honest parties eventually receive the same messages in the same order.

The protocol ensures that these predicates hold:

Integrity For all honest p and m, an honest process sends (**out**, **a-mcast**, p, m) in view x at most the number of times that p sent (**in**, **a-mcast**, m) in view x.

Uniform Agreement If q is an honest member of V^{x+k} for all $k \geq k'$ for a k' and an honest p sends (**out**, **a-mcast**, r, m) in view x, then q sends (**out**, **a-mcast**, r, m) in view x.

Validity-1 If p is an honest member of V^{x+k} for all $k \geq k'$ for a k', then p sends (**out**, **a-mcast-view**, V^x).

Validity-2 If p and q are honest members of V^{x+k} for all $k \geq k'$ for a k' and p sends (**in**, **a-mcast**, m) in view x, then q sends (**out**, **a-mcast**, p, m) in view x.

Order If q is an honest members of V^{x+k} for all $k \geq k'$ for a k' and an honest p sends (**out**, **a-mcast**, r, m) before (**out**, **a-mcast**, r', m') in view x, then q sends (**out**, **a-mcast**, r, m) before (**out**, **a-mcast**, r', m') in view x.

8.7.1 Informal Description

When a party p is not voted out of the group, it receives all messages. When party p is voted out, it does not receive any multicast message. Now when party p is permitted back into the group, we start synchronizing. The party goes into a special mode, setting the flag 'joining' to **true**, and it multicasts a message **request-hashes**, reporting the number of messages that it received before being voted out of the group. The other parties respond by sending the hash of the messages that have been sent after the amount of messages specified in the **request-hashes** message. Eventually, party p receives at least $\lceil(2|V^x| + 1)/3\rceil$ hashes, of which at least $\lfloor(|V^x| - 1)/3|\rfloor + 1$ hashes are the same, since these are contributed by honest parties. Party p now stores the correct hash of the missing messages in a variable 'hash', and selects one party at random from which it requests the missing messages. An honest party responds to this request by sending those messages. Upon receipt, these messages are checked against the obtained hash. If the response times out, or if the response is incorrect, another party is selected at random, until p obtains the correct set of messages.

While obtaining these missing messages, p enqueues any message that is received by the atomic multicast protocol on the 'save' queue. After obtaining the missing messages, p delivers them immediately, and then delivers the messsages queued on 'save'. The synchronization is now complete.

8.7.2 The Interface

The protocol is initiated with the following message:

(**in**, **open**, **SAMP**, SecureGroupMembershipProtocol)

(**in**, **s-mcast**, m)

(**out**, **s-mcast**, p_s, m)

(**out**, **s-mcast-view**, x)

8.7.3 The Synchronized Atomic Multicast Protocol SAMP

Protocol SAMP for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **SAMP**, SecureGroupMembershipProtocol):
    AMP := **new** (**in**, **open**, **AMP**, SecureGroupMembershipProtocol)
    messages := $\emptyset$
    member := $p_t \in SGM.V(SGM.x)$
    joining := **false**

UPON RECEIVING MESSAGE (**in**, **s-mcast**, m):
    send (**in**, **a-mcast**, m)

UPON RECEIVING MESSAGE (**out**, **a-mcast**, p_s, m):
    if joining **then**
        enqueue $\langle p_s, m \rangle$ on save
    else
        send (**out**, **s-mcast**, p_s, m)
        enqueue $\langle p_s, m \rangle$ on messages

UPON RECEIVING MESSAGE (**out**, **a-mcast-view**, x):
    if $p_t \notin SGM.V(x) \wedge$ member **then**
        member := **false**

    if $p_t \in SGM.V(x) \wedge \neg$ member **then**
        member := **true**
        joining := **true**
        save := $\emptyset$
        Hashes := $\emptyset$
        hash := $\emptyset$
        stop timer (**messages**)
        send (**in**, **a-mcast**, request-hashes, |messages|)

UPON RECEIVING MESSAGE (**out**, **a-mcast**, p_s, $\langle$request-hashes, $b \rangle$):
    if ¬joining **then**
        send (**reply-hash**, |messsages|, hash on messages[b, |messages|))

UPON RECEIVING MESSAGE (**reply-hash**, e, h) FROM p_s:
    Hashes := Hashes $\cup \langle e, h \rangle$
    if |Hashes| = $\lceil (2|SGM.V(SGM.x)| + 1)/3 \rceil$ **then**
        Hash, end := h, e where $\langle e, h \rangle$ occurs at least $\lfloor (|SGM.V(SGM.x)| - 1)/3 + 1 \rfloor$ times in Hashes
        select a p
        send (**request-messages**, |messages|, end) to some p
        start timer (**messages**)

UPON RECEIVING MESSAGE (**request-messages**, b, e) FROM p_s:
    if |messages| $\geq e$ **then**
        send (**reply-messages**, messages[b, e)) to p_s

Figure 25: Synchronized Atomic Multicast Protocol

```
UPON RECEIVING MESSAGE (reply-messages, m) FROM p_s:
    stop timer (messages)
    if Hash = hash on m ∧ |m| = end − |messages| then
        while m ≠ ∅ do
            dequeue head(m) as ⟨p_s, m′⟩
            enqueue ⟨p_s, m′⟩ on messages
            send (out, s-mcast, p_s, m′)
        while save ≠ ∅ do
            dequeue head(save) as ⟨p_s, m′⟩
            enqueue ⟨p_s, m′⟩ on messages
            send (out, s-mcast, p_s, m′)
        joining := false

    else
        select another p
        send (request-messages, r, end) to some p
        start timer (messages)

ON TIMEOUT (messages)
    select another p
    send (request-messages, |messages|, end) to some p
    start timer (messages)
```

Figure 26: Synchronized Atomic Multicast Protocol (Continued)

8.8 Threshold Key Generation Protocol

This section describes the key generation protocol. After running this protocol, each participating party obtained three things: 1) a private key share, which it uses to sign a message, 2) the public verification keys of each other party, with which a message signed by the private key share of another user can be verified, and 3) a public key. Let P be the set of parties. The private key shares are constructed such that if $\lceil (2|P|-1)/3 \rceil$ parties sign a message m with their private key share, a signature can be constructed from those shares which can be verified to be correct by the public key. Each private key share is unique and only known at a single party, while the public key shares and the public key are the same at each party. The group in which the calculations are done, is chosen to be the same group used by party p_0.

8.8.1 Informal Description

Each party acts as a dealer to share a secret using Feldman's verifiable secret sharing scheme. A random polynomial is selected, of which the constant factor is equal to the shared secret. First, each party receives its own share of the secret. After that, commitments of the coefficients of the polynomial are broadcast, which are used to verify the correctness of the shares. After receiving the shares of all parties, the private key is computed from this secret, and a **correct** message is broadcast signalling that everything went well. After receiving a **correct** message from every party, the protocol halts successfully, returning the public key share of every party, a public key and a private key share.

If any of the parties tries to cheat, any party can complain and the key generation protocol will halt, without having generated keys. While the protocol described in this section is largely taken from [GJKR03], our protocol more easily fails. Instead of trying to recover from errors, the protocol just halts. This is done for a few reasons: In our voting bulletin board application, key generation is probably done at least a few weeks or months before the election, because the resulting public key must

be distributed to the voters. It is important that every party participates in this key generation, because these are the only parties that play a useful role in the bulletin board. If a party is unresponsive during the key generation, the unresponsiveness must be solved and the key generation must be restarted. If an attacker tries to disturb the key generation, he will be discovered long before he can do any damage. Every party can disrupt the key generation by just being silent. Therefore, the protocol does not contain code to recover from errors: If a particular equation does not match, the protocol is just halted. No uncovers of committed secrets are requested, that just adds complexity which opens up potential vulnerabilities. An attacker knows in advance that introducing errors will just halt the protocol, which he could have done himself anyway by just not sending any messages.

This protocol uses the atomic multicast protocol, and therefore also the group membership protocol. Since this protocol is run only once in the startup phase of our bulletin board, and since each party must participate in the creation of the key shares, the protocol will fail if during the run of the protocol a party is removed from the group. That member has to be added to the group again before another attempt at generating the key shares. A precondition of the protocol is that all parties must be in the current group view.

8.8.2 Interface of the Protocol

To start the protocol, send the following message:

(**in**, **open**, **TKGP**, SecureGroupMembershipProtocol)

If the protocol executed correctly, the protocol returns with a public key and a private key share:

(**out**, **halt**, **TKGP**, PublicKeyShares, PublicKey, PrivateKeyShare)

If, however, a member of the group was removed, the protocol returns without keys:

(**out**, **halt**, **TKGP**)

8.8.3 The Protocol

Protocol TKGP for party p_i

UPON RECEIVING MESSAGE (**in**, **open**, **TKGP**, SecureGroupMembershipProtocol, threshold):
SGMP := SecureGroupMembershipProtocol
$\langle p, q, g\rangle$:= head(SGMP.P).Group
n := |SGMP.P|
t := threshold
let $a_k \in_r \mathbb{Z}_p$ for each $k \in [0,t]$, and $f(z) = a_0 + a_1 z + \ldots + a_t z^t$
$x := a_0$
ReceivedCommits := $\emptyset$
send (**share**, $f_i(j)$) to p_j for each $p_j \in$ SGMP.P
send (**in**, **a-mcast**, $\langle$**poly-commits**, $\{g^{a_k}\}_{k\in[0,t]}\rangle$)
start timer (**generated**)

UPON RECEIVING MESSAGE (**share**, x_i) FROM p_s:
ReceivedShares[p_s] := x_i

UPON RECEIVING MESSAGE (**out**, **a-mcast**, p_s, $\langle$**poly-commits**, $\{X'_k\}_k\rangle$):
if $p_s \notin$ ReceivedShares **then**
send (**in**, **a-mcast**, $\langle$**complaint**$\rangle$)

else
$X_{sk} := X'_k$ for each $k \in [0,t]$
if $g^{\text{ReceivedShares}[p_s]} \neq \prod_{k\in[0,t]} X_{sk}^{j^k} \mod p$ **then**
send (**in**, **a-mcast**, $\langle$**complaint**$\rangle$)
else
ReceivedCommits := ReceivedCommits $\cup \{p_s\}$
if |ReceivedCommits| = n **then**
send (**in**, **a-mcast**, $\langle$**correct**$\rangle$)

ON TIMEOUT (**generated**)
send (**in**, **a-mcast**, $\langle$**complaint**$\rangle$)

UPON RECEIVING MESSAGE (**out**, **view**, V^x):
send (**in**, **a-mcast**, $\langle$**complaint**$\rangle$)

UPON RECEIVING MESSAGE (**out**, **a-mcast**, $\langle$**complaint**$\rangle$):
send (**out**, **halt**, **TKGP**)

UPON RECEIVING MESSAGE (**out**, **a-mcast**, p_s, $\langle$**correct**$\rangle$):
Correct := Correct $\cup p_s$
if |Correct| = n **then**
stop timer (**generated**)
$y := \prod_{k\in[0,n]} X_{k0} \mod p$
send (**out**, **halt**, **TKGP**, $\{X_{k0}\}_{k\in[0,n]}$, y, x)

Figure 27: Threshold Key Generation Protocol

8.9 The Voting Protocol

This section describes three protocols. The first protocol is used by servers to receive votes from clients, broadcast it to the other servers and record the vote in a list, and to answer requests to read votes. The second one is used by a client to cast a vote, and to read votes that are cast by others. The third protocol is used by the talliers to read all votes cast, and write back the results of the election.

Note that the voting protocol should contain authorization verification. Clients that send their vote to the bulletin board must be authorized to vote, in order to prevent them from voting multiple times or casting votes for others. This information might be implicit in the vote that they cast, using signed votes for example. Clients that want to read the entire contents of the bulletin board must also be

authorized to do so, otherwise many clients would be able to overload the bulletin board. Such authorization, however, is outside the scope of this project and clients are therefore assumed to be authorized.

8.9.1 Informal Description

The bulletin board has three states, NotStarted, Voting, and Tallying. When the protocol is started, the bulletin board is in the NotStarted state, and is unable to receive votes. When it is time to start the voting protocol, the message **start-voting** is multicast by each party. On receiving $\lfloor (n-1)/3+1 \rfloor$ of these messages, the protocol switches to the Voting state, and is ready to receive votes. When contacted by a client, a party p multicasts the vote. Upon receiving a multicast vote, each party responds with a signature on this vote, sent in private to p. Party p then combines the signatures and presents it to the client as a proof of voting. If the client does not receive this proof of voting in time, it assumes that p is unresponsive, selects another party and tries to vote again.

When it is time to stop the voting, the message **stop-voting** is multicast by every party. On receiving $\lfloor (n-1)/3+1 \rfloor$ of these messages, the protocol switches to the Signature state, upon which the bulletin board jointly produces a signature on all votes. In this state, clients are no longer able to cast votes. Once this signature is produced, the bulletin board switches to the Tallying state. Talliers are now able to read all votes, by contacting one of the parties, which responds with all votes and the signature created in the Signature state.

8.9.2 Interface of the Server Protocol

To start the bulletin board, send the following **open** message. Clients will not immediately be able to cast votes.

(**in**, **open**, **SVP**)

When the voting period starts, send the following message. If $\lfloor (n-1)/3+1 \rfloor$ parties have sent this message, clients can start sending in their votes, or more precisely, servers can start forwarding votes and record them.

(**in**, **start-voting**)

When the voting period is over, send the following message. If $\lfloor (n-1)/3+1 \rfloor$ parties have sent this message, clients cannot send any more votes. Immediately, talliers are allowed to read the votes and write back the information.

(**in**, **stop-voting**)

8.9.3 Interface of the Client Protocol

To cast a vote, send your vote v with the following **open** message.

(**in**, **open**, **CVP**, v)

If the vote was successfully cast, the protocol terminates with a certificate proving successful voting.

(**out**, **halt**, **CVP**, v, signature on v)

8.9.4 Interface of the Tally Protocol

This document does not describe the whole tally protocol, but it does describe the steps it takes to read the votes and write the results to the bulletin board. It is assumed that the tally protocol starts after the servers entered the tallying state. To start the tally protocol, send this opening message:

(**in**, **open**, **TP**)

Once opened, the protocol will retrieve the votes from the bulletin board and report them to a higher protocol layer, which does the work of actually tallying the votes. Once the votes are read, the protocol halts with the following message:

(**out**, **halt**, **TP**, *v*)

8.9.5 The Server Voting Protocol SVP

Protocol SVP for party p_t

UPON RECEIVING MESSAGE (**in**, **open**, **SVP**):
    State := NotStarted

UPON RECEIVING MESSAGE (**in**, **start-voting**):
    send (**in**, **s-mcast**, ⟨start-voting ⟩)

UPON RECEIVING MESSAGE (**in**, **stop-voting**):
    send (**in**, **s-mcast**, ⟨stop-voting ⟩)

UPON RECEIVING MESSAGE (**out**, **s-mcast**, p_s, ⟨start-voting ⟩):
    StartVoting := StartVoting $\cup\{p_s\}$
    if |StartVoting| = $\lfloor(n-1)/3\rfloor + 1$ **then**
        State := Voting

UPON RECEIVING MESSAGE (**out**, **s-mcast**, p_s, ⟨stop-voting ⟩):
    StopVoting := StopVoting $\cup\{p_s\}$
    if |StartVoting| = $\lfloor(n-1)/3\rfloor + 1$ **then**
        State := Signature
        send (**in**, **s-mcast**, ⟨**votes-signature**, $\langle \text{Votes} \rangle_{K_t}$ ⟩)

UPON RECEIVING MESSAGE (**cast**, *v*) FROM *c*:
    Verify State = Voting
    Verify that *c* is an authorized voter and that *v* is a valid vote
    send (**in**, **s-mcast**, ⟨cast, *c*, *v*⟩)

UPON RECEIVING MESSAGE (**in**, **s-mcast**, p_s, ⟨cast, *c*, *v*⟩):
    Verify that *c* is an authorized voter and that *v* is a valid vote
    if $c \in$ Votes **then**
        send (**signature**, *c*, Votes[*c*], signature on *c* and Votes[*c*]) to p_s

    else
        enqueue *v* on Votes with key *c*
        send (**signature**, *c*, *v*, signature on *c* and *v*) to p_s

UPON RECEIVING MESSAGE (**signature**, *c*, *v*, signature) FROM p_s:
    Signatures[*c*, *v*] := Signatures[*c*, *v*] $\cup\langle p_s, \text{signature}\rangle$
    if |Signatures[*c*, *v*]| = $\lfloor(n-1)/3\rfloor + 1$ **then**
        send (**signature**, *v*, Signatures[*c*, *v*]) to *c*

UPON RECEIVING MESSAGE (**in**, **s-mcast**, p_s, ⟨**votes-signature**, signature⟩):
    Verify State = Signature
    VotesSignatures := VotesSignatures $\cup\langle p_s, \text{signature}\rangle$
    if |VotesSignatures| = $\lfloor(n-1)/3\rfloor + 1$ **then**
        State = Tallying

UPON RECEIVING MESSAGE (**read**) FROM *t*:
    Verify State = Tallying
    send (**read**, Votes, VotesSignatures) to *t*

Figure 28: Voting Protocol for Servers

8.9.6 The Client Voting Protocol CVP

Protocol CVP for party *c*

UPON RECEIVING MESSAGE (**in**, **open**, **CVP**, *v*):
    select a bulletin board server *p*
    send (**cast**, *v*) to *p*
    start timer (**vote**, *v*)

ON TIMEOUT (**vote**, *v*)
    select another bulletin board server *p*
    send (**cast**, *v*) to *p*
    start timer (**vote**, *v*)

UPON RECEIVING MESSAGE (**signature**, *v*, signature) FROM p_s:
    verify that the signature is correct
    send (**out**, **halt**, **CVP**, *v*, signature)

Figure 29: Voting Protocol for Clients

8.9.7 The Tally Protocol TP

Protocol TP for party *t*

UPON RECEIVING MESSAGE (**in**, **open**, **TP**):
    select a bulletin board server *p*
    send (**read**) to *p*
    start timer (**read**)

ON TIMEOUT (**read**)
    select another bulletin board server *p*
    send (**read**) to *p*
    start timer (**read**)

UPON RECEIVING MESSAGE (**read**, *v*, *s*) FROM p_s:
    stop timer (**read**)
    if *s* are $\lfloor (n-1)/3 \rfloor + 1$ valid signatures from different parties on *v* **then**
        send (**out**, **halt**, **TP**, *v*)

    else
        select another bulletin board server *p*
        send (**read**) to *p*
        start timer (**read**)

Figure 30: Voting Protocol for Talliers

9 The Implementation

With the complete set of specifications of the protocols of the previous sections, the next task is to write the code for the bulletin board. C++ is the language chosen for this implementation. In this section, we first examine methods and constructions used in the code. Later, we discuss each protocol layer, and describe how the specification is translated into code.

9.1 Writing Secure Code

The code created in this project must be secure. In real life, programs intended to secure resources, such as a Secure Shell server, are usually not cracked by breaking the protocols or even the cryptographic primitives, but are broken by exploiting bugs in the implementation of the protocols and algorithms. While in non-cryptographic software, a bug may be a mere degradation of certain functionality, in cryptographic software, a single bug may be responsible for the compromising of the entire system. Only one buffer overflow bug is enough to break into an SSH server. A small bug in the random-number generator of an SSL server is enough to eavesdrop on the communication. Therefore, it is very important to carefully consider programming methods that improve the quality of the code. In this section, we will review a few principles which, in our opinion, are very important when writing secure code.

We regard *simplicity* as one of the most important things when writing secure code. Simple specifications and implementations are easier to understand, and therefore bugs are easier to spot. Simple specifications also lead to shorter implementations, and less code means less opportunities to make mistakes.

Another thing that is important is to stay close to the specification. Many different ways exist in which functionality defined in the specification can be implemented. Especially when trying to optimize code, the resulting code may fulfill its specification but it may be unclear to an observer why. It is easy to introduce bugs while trying to optimize code, and these bugs may be very hard to find. An advantage of staying close to the specification, is that code is much easier to check against its specification.

Of course, many others have written about creating secure code. We do not claim that these two principles are even remotely sufficient, but these principles were followed as closely as possible in creating the software, and therefore had a very high impact on the code.

9.2 Libraries Used

This section describes the various libraries that are used in the code of the bulletin board.

The C++ Standard Template Library The C++ Standard Template Library (STL) is part of the C++ language. It contains basic ingredients, such as containers like *vector* and *set*.

The Boost Libraries This set of libraries, available at `http://www.boost.org`, contains various ingenious constructions, which supplement the C++ STL. The library is specifically written to

cooperate with the STL, and almost all of the libraries proposed for the next C++ standard are part of Boost. The section "Constructions Used" describes most of the pieces taken from Boost.

A Cryptographic Library Of course, software dealing with cryptographic constructions needs cryptographic primitives. This library contains the AES block cipher, various hash functions, various modes of operations for both block ciphers and hash functions, and public key algorithms like Schnorr signatures. This library is, like the Boost libraries, written to cooperate nicely with the C++ STL. Unsatisfied with the current, mostly C oriented, cryptographic libraries, I have written this library myself. There exists a cryptographic library for C++ written by Wei Dai, but this library has the drawback that it is a complete framework, it is difficult to use only small portions.

A Big Integer Library Unfortunately, neither the C++ STL nor Boost contains a big integer library (yet). The public key algorithms of the cryptographic library depend on this library for its unlimited precision integer operations. This library is written by myself. For faster calculations than the default implementation, it can serve as a front-end for fast (C oriented) big integer libraries like the Gnu Multi Precision library.

9.3 Constructions Used

One of the requirements is that code is kept as simple as possible. Boost contains various libraries that can make a programmer's life very easy, but to someone unfamiliar to these constructions, it may seem like hocus-pocus. The intention of this section is to provide some background so that most of the constructions code can be understood. Therefore, this section describes the constructions that have had the greatest influence on the design of the code.

Variants A variant, also known as a discriminated union, is an object that can hold another object of any of a few types. One of the Boost libraries provides such a variant type. For example, `variant<int, string>` can hold either an integer or a string. This construction is used in the dispatching of messages. When some protocol sends for example an `init_message`, an `echo_message`, or a `commit_message`, it actually passes an object of type `variant< init_message, echo_message, commit_message>` to a lower protocol layer. The lower protocol is a class templatized (meaning the class has a template parameter) on a single message type, and it has the responsibility of sending objects of that type. This has the advantage that protocols do not need to know the exact type of messages they have to send, because that type is substituted when the higher-level protocol is created. This construction can be chained form protocol layer to protocol layer.

For example:

```
template<class MessageType>
class low_protocol_layer {
    void send(MessageType message);
};

template<class MessageType>
```

```
class mid_protocol_layer {
    low_protocol_layer<variant<MessageType, init_message,
        echo_message, commit_message> > protocol;

    void send(MessageType message) {
        protocol.send(message);
    }
};

class high_protocol_layer {
    mid_protocol_layer<variant<end_message, flush_message> >
        protocol;

    void send(MessageType message) {
        protocol.send(message);
    }
};
```

Here, `low_protocol_layer` is the lowest protocol layer sending messages of a type `MessageType` that is to be substituted later. `mid_protocol_layer` does that substitution, with a variant of four different types. One of those types needs itself to be substituted by a higher protocol. `high_protocol_layer` does that, with a variant of two types.

Next, we need to be able to extract the type of the object currently held by a variant. Suppose the low level protocol layer receives a message. It passes that message on to a higher level protocol as a single object, the variant. That higher protocol now has to unwrap the variant. We will not go into details here, but part of the unwrapping involves calling a function overloaded on all the types held by the variant. Consider the following code:

```
void received_data(variant<end_message, flush_message> message) {
    apply_visitor(make_mcast_visitor(), message);
}

void dispatch_mcast(end_message message) {
    ...
}

void dispatch_mcast(flush_message message) {
    ...
}
```

`received_data` is called by the lower level protocol with the received message. The `apply_visitor` function then calls, with the help of the `make_mcast_visitor`, the proper overload of `dispatch_mcast`, and the contents of the variant is nicely split.

Variant is an enormous help in dispatching messages from layer to layer. Without variant, we would probably have to write a parse function for every protocol layer. With variant, there is only one

problem left: The lowest level protocol has to convert the variant object into a stream of bytes, in order to be able to send it to another party. The next section deals with that problem.

Serializing Objects The Boost Serialization library contains functionality to convert objects to a sequence of bytes and back (or to XML, as we will see later). It is a large and complicated library, but fortunately, its use is extremely simple. Therefore, we will not delve into details, but we just show a few examples.

Consider this class:

```
class init_message{
    int x;
    string message;
};
```

We want to convert an object of type `init_message` into something that we can send across the network. First, we add functionality so that the Serialization library knows how to serialize it:

```
class init_message{
    int x;
    string message;

    template<class Archive>
    void serialize(Archive & ar, const unsigned int version) {
        ar & boost::serialization::make_nvp("x", x);
        ar & boost::serialization::make_nvp("message", message);
    }
};
```

That is all. Just one templatized function, containing a serialization instruction for each field in the object. Every object that has to be serialized, has such a `serialize` function. The code to actually convert an object into a sequence of bytes or the other way around is very short, only 4 lines, and is not very interesting. Only a few places in the code need to do this, so we omitted that code here. What is more interesting, is what the data looks like after converting. The Serialization library has the option not to convert the data to a very compact representation, but also to an XML representation of the object. The XML representation also helps when debugging code. If we serialize an object of type `init_message`, we get the following data:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="3">
<init_message class_id="0" tracking_level="0" version="0">
        <x class_id="1" tracking_level="0" version="0">5</x>
        <message class_id="2" tracking_level="0" version="0">this
            is a message</message>
</init_message>
</boost_serialization>
```

Now ignoring some information used by the Serialization library itself, we can see that the value of `x` is 5, and that the string `message` contains "this is a message".

If we serialized an object of a certain type, we can only deserialize an object of the same type. The Serialization library does not help us find out what that type is. However, we want to serialize messages of various types, and at the other end deserialize the message as an object of the correct type. We do this by combining the messages into a single variant, as seen in the previous section. Now we have a single object we need to serialize, namely the variant. After deserialization, we use the functionality of the variant to extract the correct message.

Combining the Variant and Serialization libraries has given us an enormously powerful tool in sending messages from one party to another. Without those libraries, we would have to write parsers, and deal with many different messages spread across many protocol layers. Those big parsers would have cost much time, and would likely have a few bugs so it would also cost a lot of debugging time tracing hard-to-find bugs. With the Variant and Serialization library, we only need to mechanically write `serialize` functions for each object that we want to serialize, and pass on variants from one protocol layer to another.

There exists a significant drawback, however. Due to the heavy use of templates, the length of the names of types grow very large, resulting in slow compilation times and even triggering internal compiler limits. Consider a variable of type `variant<int, string>`. Internally, the type of this variable is `boost::variant<int, _STL::string, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_, boost::detail::variant::void_>`. Now when several layers of `variant` are combined, the length of this type grows tenfold. When serializing objects of this type, templates are used where types of objects also grow larger. During compilation, types were detected with a size approaching 10,000. Microsoft Visual C++ 7.1 could not handle these constructions, and produced internal compiler errors. GNU's GCC luckily is still able to handle this code. The use of objects with very large types is quite new, and future compilers are expected to be better at handling them. In any case, once the source successfully compiled, the typenames have no influence on the speed or size of the code.

Function and Bind This section discusses the Boost libraries Function and Bind. While these are actually two separate libraries not depending on each other, we use these libraries mostly together. They find their use in *callbacks*. Callbacks are used for example when a low-level protocol has received data, it uses a callback to notify a higher-level protocol that a new message is available.

Using Function is not very difficult. Suppose you want to create a callback taking a string as parameter. You create a variable `callback` of type `function<void(string)>`. Now you can assign anything to `callback` that you would normally be able to call as a function taking one string as a parameter. After that, you can use `callback` just as if it were a normal function. For example:

```
void print(string m);

void do_something() {
    function<void(string)> callback;
    callback = &print;
    callback("hello");
}
```

When `do_something` is executed, the third line will call the function `print`.

Now it gets a little more interesting when Bind is involved. `bind` is a function taking as argument a function (this may be a member function of a class), and a number of parameters. `bind` returns something you can store inside a `function`. When you call that `function`, it works as if the arguments passed to bind are already stored in the arguments of the function specified in `bind`. For example:

```
void print(string m);

void do_something() {
    function<void()> callback;
    callback = bind(&print, "hello");
    callback();
}
```

Here, `callback` is a function taking 0 arguments. `bind` takes the function `print`, binds a string as a parameter, and assigns the result to `callback`. Now when `callback` is actually called, `print` is called with "hello" as a parameter.

There's more. When passing `_1` to `bind`, `bind` returns a function taking one parameter, and that parameter is substituted as the parameter at the position of the `_1`. For example:

```
void print(string m);

void do_something() {
    function<void()> callback;
    callback = bind(&print, _1);
    callback("hello");
}
```

Now, how do we use this in the code? Suppose, we have a protocol layer that want to notify a higher protocol layer when a message has arrived. The following code fragment shows how this is done.

```
template<class MessageType>
class low_protocol_layer {
    function<void(MessageType)> received_data_callback;

    void handle_received_data(function<void(MessageType)> handler) {
        received_data_callback = handler;
```

```
    }

    void something() {
        MessageType message = ...;
        received_data_callback(message);
    }
};

class high_protocol_layer {
    low_protocol_layer low_protocol;

    high_protocol_layer() {
        low_protocol.handle_received_data(bind(
            &high_protocol_layer::received_data, this, _1));
    }

    void received_data(string message);
}
```

First, the constructor of `high_protocol_layer` calls `handle_received_data` to properly assign the callback. It uses `bind(&high_protocol_layer::received_data, this, _1)`, to make the callback point to its `received_data` member function, which takes one parameter, namely the message itself.

Once the protocol is running, the low-level protocol may decide that it wants to notify the higher level protocol of a message. It calls `received_data_callback` with `message` as a parameter. This triggers `high_protocol_layer`'s `received_data` function, and passes the message as a parameter.

The costs to use `function` and `bind` are not high. Calling a `function` is about as fast as calling a virtual function. Using `bind` with a `function` requires space to store the arguments, but does not decrease speed.

Sockets A few classes are created to handle network traffic. One of the classes represents a TCP/IP socket, providing functionality to send data from one party to another. Another class represents a TCP/IP listening socket, listening on a specific port, enabling other parties to make a connection with the party on this port. A function `check_activity` is used to check for network activity, reporting such activity via callbacks described in the previous section. Network activity of a TCP/IP socket includes indication that a connection has been successfully made, that data is ready to be received, that more data may be sent, and that the connection has been lost. Network activity of a TCP/IP listening socket only consists of the indication that another party wants to make a connection. The function `check_activity` checks for network activity on all sockets at the same time. It takes a single parameter indicating the time it may wait for activity. If this time passes without any activity, the function returns without triggering any callbacks. If during that time there was some activity, for example new data has arrived on a particular socket, the appropriate callback is triggered so that protocols may handle the incoming data. After processing network activity, the `check_activity` function returns immediately without waiting for more network activity. The reason of this behaviour

will become clear later on.

Timers The formal protocol descriptions often use timers. One class `timer` was created to easily deal with timer events. Once a `timer` object is created, it is assigned a specific time after which it triggers a callback. The design is similar to the design of the sockets: a single function exists which triggers every `timer` object which is due for triggering. This function, unlike the `check_activity` function of the sockets, does not accept a duration as parameter indicating how long it may wait. The function returns immediately after triggering the appropriate callbacks. Instead, another function computes the duration until the next `timer` object is due for triggering. The next section describes how the timers and the sockets cooperate.

9.4 The Execution Flow of the Program

Each server of the bulletin board has to deal with several network connections and several timers. Where a multi threaded approach might be expected, the implementation of the bulletin board is written as a single threaded program. This section discusses why this is done and how this is accomplished.

First, we discuss how we were able to create the bulletin board as a single threaded program. We have two sorts of events on which we have to respond: timers and network traffic. Each timer is registered in a global list of timers. We can find out how much time is left before the first timer triggers. This time is spent checking the network sockets. All sockets are checked simultaneously for network activity, for a duration equal to the time left before the next timer triggers. If there is no network activity, we trigger the correct timer, and recalculate how much time is left before the next timer triggers, and spend this time checking for network activity. If there is network activity, then the appropriate handlers are triggered. Immediately after returning from these handlers, the program checks if the time left before the next timer triggers has changed. Indeed, the handling of the network activity may have started a new timer, or changed an existing one.

This system of handling events has the condition that each handler may not wait itself for events. If it blocks for an unspecified amount of time, the whole program stops. It is however not difficult to write the event handlers in this way.

The workings of this system may sound quite complicated, but in reality, it is very simple and very effective. There exists one loop in the program which does the checking of timer or network activity:

```
while (true) {
    socket_base::check_activity(timer::next_trigger());
    timer::check();
}
```

The advantage of a single threaded approach over a multi threaded approach is that functions do not need to be made thread-safe. In the single threaded approach, the event handler that is executing does not need to worry about other event handlers touching the same data, and no deadlock can occur. Overall, it makes reasoning about the behaviour of the program a lot easier.

9.5 Translating the Specifications into Code

Having described several constructions, each having their own impact on the code, we describe how a specification of a protocol is translated into an implementation.

First, we observe that each protocol is mainly composed of a series of event handlers, responding to several different message types: the (**in**, **open**, **protocol**, parameters) message starting the protocol, the (**in**, **name**, parameters) message processing data obtained from a higher-level protocol, the (**out**, **name**, parameters) message processing data obtained from a lower-level protocol, the (**name**, parameters) message processing data obtained from this protocol, and timeout events. Furthermore, some protocols have procedures which can be called directly from this protocol or higher-level protocols.

For each protocol, a class is created representing the functionality of that protocol. Since (almost) every protocol has to send messages for a higher-level protocol, of which the type is unknown to that protocol, the class gets a template parameter representing the type of the message it can send. The echo multicast protocol, and higher protocols, are not only capable of sending a message to a single party, but also of sending a message to every party, multicasting that message. Therefore, the classes representing the echo multicast protocol and higher protocols have two template parameters, one containing the type of the message sent to a single party, and one containing the type of the message which is multicast. We say *the* message, even though higher protocols send messages of a variety of types, because those messages are combined into one type using the variant library.

For example, the echo multicast protocol gets this class:

```
template<class MessageType, class MulticastMessageType>
class echo_multicast_protocol {
    ...
};
```

Constructor Messages The (**in**, **open**, **EMP**, SecureGroupMembershipProtocol) message is translated into a constructor. The protocol assumed that the secure group membership protocol is created before the echo multicast protocol, but in code it is easier to let the echo multicast protocol create the secure group membership protocol, so we pass the parameters of the secure group membership protocol constructor to this constructor:

```
template<class MessageType, class MulticastMessageType>
class echo_multicast_protocol {
public:
    echo_multicast_protocol(const own_party_id& identity,
        const party_set& parties);

private:
    secure_group_membership_protocol<...> m_sgmp;
};
```

The secure group membership protocol now is a member of the echo multicast protocol. Each name of a member of a class has the prefix `m_` to distinguish it from other variables.

Messages Between Protocol Layers The (**in**, **name**, parameters) messages are translated into public functions, so that higher-level protocols can call those functions directly. The (**out**, **name**, parameters) messages are translated into callbacks, each with a function with the prefix `handle_` so that the higher-level protocol can point the callback into the right direction.

```
template<class MessageType, class MulticastMessageType>
class echo_multicast_protocol {
public:
    typedef boost::function<void(const party_id& sender,
        const T1& message)> received_data_callback;

    echo_multicast_protocol(const own_party_id& identity,
        const party_set& parties);

    void handle_received_data(received_data_callback handler);

    void e_mcast(std::size_t x, const multicast_message_type&
        message);

private:
    secure_group_membership_protocol<...> m_sgmp;
    received_data_callback m_received_data;
};
```

Here, we first make a typedef of the complicated type of the callback used in the `received_data` callback and name it `received_data_callback`, then we introduce a member holding the target of the callback named `m_received_data`, and we create the function `handle_received_data` to assign a target to `m_received_data`. If the echo multicast protocol wants to notify a higher-level protocol that data has arrived, it simply executes `m_received_data(sender, message)` and the function previously passed to `handle_received_data` is executed.

Messages Between Parties The next task is to translate the messages sent within a protocol layer, between parties. Each different message gets its own class. This class holds variables containing the data of the message, and functionality to serialize it, using the serialization library. For example, the class of the `init` message looks like this:

```
class init_message {
public:
    init_message();
    init_message(std::size_t a_x, const std::string& a_digest);

    std::size_t x;
    std::string digest;

private:
    friend class boost::serialization::access;
```

```
    template<class Archive>
    void serialize(Archive & ar, const unsigned int version)
    {
        ar & boost::serialization::make_nvp("x", x);
        ar & boost::serialization::make_nvp("digest", digest);
    }
};
```

Sending an init message where $x = 5$ and digest="banaan" is done by calling the `send` function of the lower protocol, with a newly created `init_message` object as parameter.

```
    m_sgmp.send(destination, init_message(5, "banaan"));
```

This message eventually arrives at the destination. The secure group membership protocol then triggers the callback originally assigned by the echo multicast protocol with:

```
    m_sgmp.handle_received_data(boost::bind(
        &echo_multicast_protocol::received_data, this, _1, _2));
```

So, eventually the `received_data` function of the echo multicast protocol is called, with the source party and the message as parameters. Since this message is actually a variant containing our `init` message, the `received_data` function has to split the variant so that the correct `dispatch` function is called, which handles the `init` message.

This code is used to dispatch the message (the other code of the class is omitted):

```
template<class MessageType, class MulticastMessageType>
class echo_multicast_protocol {
    void received_data(const party_id& p_s, const message_type&
        message)
    {
        boost::apply_visitor(make_dispatcher(this, p_s), message);
    }

    void dispatch(const party_id& p_s, const init_message& message)
    {
        ...
    }

    void dispatch(const party_id& p_s, const echo_message& message)
    ...
};
```

Timers The final event-processing construction we have to translate are the timeouts. We use the `timer` class to accomplish this. We take the atomic multicast protocol as example, where the sequencer has a timeout signalling that it must send an order message, defining the order in which

messages must be delivered. We introduce a variable `m_send_sequence_timer` of type `timer`, and point the handler to the right function with

```
m_send_sequence_timer.handle_triggered(boost::bind(
    &atomic_multicast_protocol::send_sequence, this));
```

in the constructor of the atomic multicast protocol. When we receive a message, we start the timer with something like

```
m_send_sequence_timer.start(boost::posix_time::seconds(2));
```

After two seconds, the `send_sequence` function is triggered.

Parameterized Variables Lots of variables are parameterized on another variable. For instance, the echo multicast protocol has a variable EchoSet[*x*, *l*], which is a set for each combination of values *x* and *l*. We use nested `std::map` classes to construct these variables. Suppose *x* and *l* each have type `std::size_t`, and the EchoSet contains a set of objects of type `echo_type`. Then the EchoSet variable is translated into

```
std::map<std::size_t, std::map<std::size_t,
    std::set<echo_type> > > m_echo_set;
```

We can now access one of the sets with `m_echo_set[x][l]`. To insert a variable `echo` of type `echo_type`, we write `m_echo_set[x][l].insert(echo);`.

Signatures Many of the protocol layers use signatures. Two functions have been created to facilitate easy signature creation and verification. `sign` takes a *signature generator*, a private key, and a random number generator, and returns a `string` containing the signature put on the data of the signature generator. The signature generator is an object translating data into a stream of bytes. It works mostly in the same way as messages, only it does not contain functionality to translate a stream of bytes back into variables, and it puts a unique identifier in front of the data. For example, this is a signature generator:

```
class deputy_signature_generator {
public:
    deputy_signature_generator(const short_party_id& p_d,
        std::size_t x);

private:
    short_party_id m_p_d;
    std::size_t m_x;

    friend class boost::serialization::access;

    template<class Archive>
```

```
    void serialize(Archive & ar, const unsigned int version);
    {
        std::string name("deputy");
        ar << boost::serialization::make_nvp("name", name);
        ar << boost::serialization::make_nvp("p_d", m_p_d);
        ar << boost::serialization::make_nvp("x", m_x);
    }
};
```

A `valid` function takes a signature generator, a public key, and a signature as arguments, and returns whether the signature is valid on the data presented by the signature generator.

These functions make creating and verifying signatures very easy. To send a message containing a signature, the following code is used:

```
sgmp().send(p_s, echo_message(x, l,
    sign(echo_signature_generator(p_s, x, l, digest), sgmp().p_t(),
        m_rng)));
```

Here, the **echo** message contains a view number x, an index l, and a signature created on the identifier of the sender p_s, the view number x, the index l and the digest of another message $digest$. p_t contains the private key, and m_rng is a nondeterministic random number generator.

To verify that the signature is correct, the receiver of the message executes the following code:

```
if (!valid(echo_signature_generator(sgmp().p_t(), x, l, digest),
    p_s, message.signature))
    return;
```

The same data is passed to the signature generator, and that signature generator is passed to the `valid` function, which checks if the signature sent by the other party is correct under his public key.

9.6 The High-Level Protocol Layers

With the constructions explained in previous sections, we can now easily translate the formal specifications into working code. The next sections explain details, and clarify how certain less obvious expressions are translated. These are the high-level protocol layers:

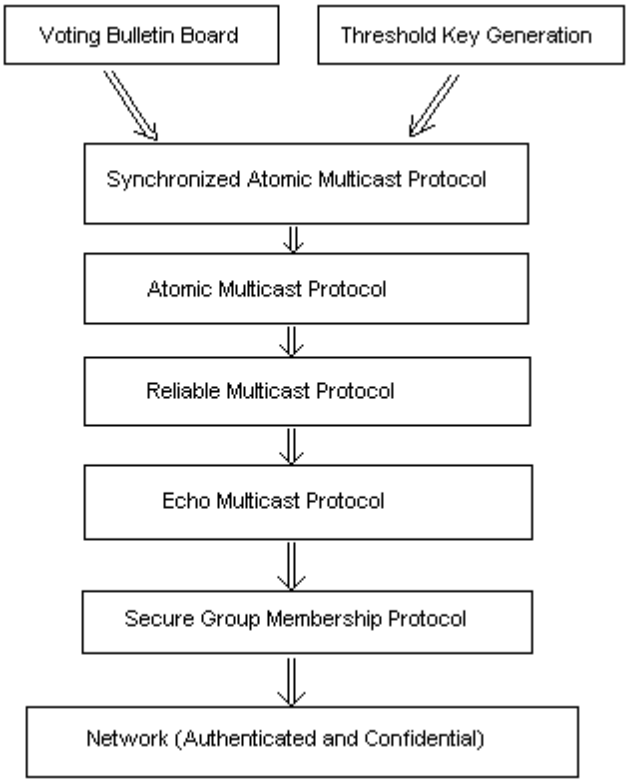

Figure 31: High-Level Protocol Layers

Of these layers, we will discuss parts of the secure group membership protocol and the echo multicast protocol. We will see how nicely the specifications translate into code, and therefore we do not need to treat the other protocol layers in further detail.

9.6.1 Secure Group Membership Protocol

The secure group membership protocol is translated into two classes: the `secure_group_membership_layer` and the `secure_group_membership_view`. These two classes represent the **SGM** protocol, and its sub-protocol **SGM-View**.

Although the secure group membership protocol is not exactly a protocol layer below the other protocols, it was easier to make this the lowest layer. The secure group membership protocol now relays messages from the echo multicast protocol to the network. This has the advantage that the type of messages which have to be sent over the network is easy to compute: The group membership protocol takes the messages of the echo multicast protocol, and adds its own messages to this list. This list is then passed to the network protocol layer. We examine this in detail:

The `secure_group_membership_view` sends messages of type `notify_message`, `suggest_message`, etc. Therefore, it has this type:

```
typedef boost::variant<notify_message, suggest_message, ack_message,
    proposal_message, ready_message, commit_message, deputy_message,
    query_message, last_message, suggest_last_message> message_type;
```

So, it sends messages of type `message_type`, in which the actual messages are embedded. The `secure_group_membership_layer` takes this message type, adds a message which is used by the group membership protocol layer directly, namely `history_message`, and adds the message type of the echo multicast protocol. The message type of the echo multicast protocol is passed to the group membership layer via the template parameter `T`. This is the type of the messages sent by the secure group membership protocol:

```
typedef boost::variant<T, history_message,
    secure_group_membership_view::message_type> message_type;
```

Translating the specification to implementation was quite straightforward. Let's examine a small example. This piece of specification:

> UPON RECEIVING MESSAGE (**suggest**, p_c, NotifySet) FROM p_s:
> **if** $p_s = p_m \wedge 3\ rank(p_m) - 1 <$ ProtocolState $\wedge |\text{NotifySet}| = \lfloor (|V| - 1)/3 \rfloor + 1$ **then**
> ProtocolState := $3\ rank(p_m) - 1$
> send (**ack**, p_c, $\langle p_m$, **ack**, $p_c \rangle_{K_t}$) to p_m

is translated into this code:

```
void secure_group_membership_view::dispatch(const party_id& p_s,
    const suggest_message& message)
{
    if (m_parties.find(p_s) == m_parties.end()
        || !message.valid(m_p_t, p_s, m_parties))
        return;

    if (p_s == p_m
        && 3 * rank(p_m) - 1 < m_protocol_state
        && message.notify_set.size() == low_threshold()) {

        m_protocol_state = 3 * rank(p_s) - 1;
        m_send(p_m, ack_message(message.p_c,
            sign(ack_signature_generator(p_m, message.p_c, m_x),
                m_p_t, m_rng), m_x));
    }
}
```

We already explained that a UPON RECEIVING MESSAGE clause is translated into a dispatch function, taking the sender and the contents of the message as arguments. The first `if` statement is not directly specified by the protocol, but it is mentioned in the text preceding the protocols that each message handler should check that the message is sent by some party in the current group view, and that the signatures in the message are valid. That is checked by the first `if` statement. The second `if` statement is almost directly translated into code. NotifySet is contents of the message, and therefore a member of the `message` object. `low_threshold` is a function evaluating the $\lfloor (|V| - 1)/3 \rfloor + 1$ formula. The two statements in the code are also almost directly translated from the specification. The group view index x, which was only implicitly stated in the specification, is now explicitly mentioned in both the sending of the message and the creation of the signature.

9.6.2 Echo Multicast Protocol

Like the secure group membership protocol, the specifications of the echo multicast protocol translate nicely into code. We take the function with which we start the echo multicast, `e_mcast`, as example. The specification of the `e_mcast` function:

UPON RECEIVING MESSAGE (**in**, **e-mcast**, x, m):
    $l[x] := l[x] + 1$
    $lm[l[x], x] := \langle l[x], m \rangle$
    for each $p \in SGM.V^x$ **do**
        if non-interactive threshold signatures are available **then**
            send (**init-ni-threshold**, x, $f(m)$) to p

        else if threshold signatures are available **then**
            send (**request-threshold**, x, $f(m)$) to p

        else
            send (**init**, x, $f(m)$) to p

Figure 32: Specification of the `e_mcast` function

translates into the following code:

```
void e_mcast(std::size_t x, const multicast_message_type& message)
{
    assert(0 < x && x <= m_sgmp.x());

    std::ostringstream oss;
    boost::archive::xml_oarchive oarchive(oss);
    oarchive << BOOST_SERIALIZATION_NVP(message);

    ++m_l[x];
    m_lm[x][m_l[x]] = oss.str();
    m_sgmp.send(m_sgmp.v(x), emp::init_message(x,
        make_digest(oss.str())));
}
```

First, an `assert` is used to verify the precondition that x is a well-defined view. Next, we translate `message` into a string, so that we can compute its hash value. We can now start with following the specifications. First, the variable l, translated into the member m_l is incremented. Then, we store the message in the variable $lm[l[x], x]$, which is translated into the member `m_lm` of type `map<size_t, map<size_t, string> >`. Note that in the code, the group view index x is always the first argument in maps. After storing the message, we create an `init_message`, and send it to everyone in the current group view. Note that neither of the threshold signatures schemes is implemented.

While most of the protocol is translated into code in a very direct way, we applied a small trick in computing the timeout values when to send the next **counters** message. The specifications only say 'periodically, send a **counters** message', but it does not say how often. If we send a **counters** message every tenth of a second, we have a relatively small latency, but we require much cpu time. If we send a **counters** message every 20 seconds, we have a big latency, but we require less cpu time. By varying the interval between **counters** messages, we try to have a small latency when there are only a few messages to be sent, and we increase the latency when there are many messages to be sent. This way,

we keep a nice balance between cpu time needed and latency. We determine the interval to the next sending of a **counters** message like this: When we receive a message, we start our timer to trigger in a 10th of a second. If we do not receive another message in that time, the timer triggers and we send our **counters** message in a 10th of a second, thus keeping latency low. If we do receive another message before our timer triggers, we add the time elapsed between the two messages to our timer, unless this timeout was already larger than 2 seconds. This way, if many messages arrive, we ensure that we send a **counters** message every 4 seconds. If in one instant a burst of messages arrive, and then much time lapses before the next message, we still send our **counters** message early, because a burst of messages does not add much time to our timeout value. If many messages arrive at regular intervals, the latency increases.

9.7 The Low-Level Protocol Layers

The higher level protocols assume that communication between parties is reliable and private. Such is a common assumption, and the theory of setting up secure channels is not difficult. We do need to implement them, however, and we try to maintain as much simplicity as possible by introducing separate layers, that each solve a small problem, so that each layer is small and easily to test.

We will use TCP/IP as our means of communication. Before we start examining what layers we need, we review the differences between what TCP/IP communication gives us, and what we need for our higher level protocol layers.

TCP/IP Sockets	Needed
bytes are sent	objects are sent
not secure	secure
communication between ports	communication between parties
zero or more connections between parties	exactly one connection between parties

With TCP/IP sockets, bytes are sent over a communication channel, whereas we need to send objects. Those bytes are not sent securely, we need confidentiality and integrity. The communication channel is made between two combinations of servers and ports, instead of between parties. We can now define the protocol layers, which provide us with the necessary functionality, each one improving a small bit over the previous layer. The first, lowest, layer interacts with the operating system and provides exactly the characteristics presented in the left column of the table above. The last layer has provides all characteristics defined in the right column. We create five layers, that provide the following characteristics:

	sent	secure	orientation	# connections
Socket Layer	bytes	no	port	zero or more
Secure Connection Layer	objects	given a secure key	port	zero or more
Key Exchange Layer	objects	yes	port	zero or more
Multiplex Layer	objects	yes	party	zero or one
Reliable Layer	objects	yes	party	one

We now discuss each layer and describe how they are implemented.

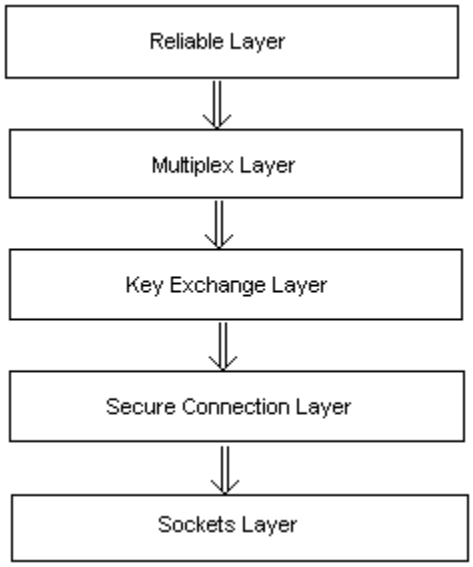

Figure 33: Low-Level Voting Protocol Layers

9.7.1 Socket Layer

The `tcp_socket` class represents the socket layer, and is an abstraction from the functionality provided by the operating system. Sockets are not standardized across different platforms, so implementations of this class may differ between operating systems. The implementation is tested under Windows 2000 and GNU/Linux, and probably works under Windows 98 and later, and all sorts of Unixes.

Before data can be sent and received with the `tcp_socket` class, a connection must be built. This can be done in two ways: the class can initiate a connection with a specific server on a specific port, or an incoming connection can be accepted. If this operation succeeds, the callback `connected` is triggered, otherwise the callback `disconnected` is triggered. The `disconnected` callback is also triggered when the connection is terminated.

Once the connection is built, data can be sent as a vector of octets. Once the other end of the connection receives this data, it triggers the `received_data` callback, after which the data can be retrieved from the socket.

One other callback exists, the `send_ready` callback, which indicates that the socket's buffers are empty and that new data may be sent.

9.7.2 Secure Connection Layer

The secure connection layer applies symmetric cryptographic algorithms to make the data sent over a connection both reliable and confidential. Furthermore, it translates the stream of bytes into objects, using the serialization library described above.

This layer does not do any key negotiation, however, because that is a functionality different from securing and translating data, that is better handled in another layer. This layer takes a key and uses

that key in the algorithms. When the connection has just been built, a default key is used, and therefore provides no security until the key exchange layer provides a secure key.

Many protocols have a separate non-encrypted mode. It is easier, however, to already have the encryption algorithms in place, so that no special mode is needed, and the protocol layer is kept is simple as possible.

The secure connection layer has the same interface as the socket layer, with the addition of methods to set the key for each direction of communication, and the difference that the send function and receive callback deal with objects instead of vectors of octets.

Once the `connected` callback of the secure connection layer is triggered, the keys are reset to a default value, and it triggers the `connected` callback of a higher level protocol. Since the other end of the communication channel resets the keys to the same value, communication is possible, but not secure.

When an object is sent by the secure connection layer, it is first serialized in order to obtain a stream of octets. The information sent to the sockets layer consists of the size of the data, then a MAC (Message Authentication Code) on the size, then the encrypted data, and finally, a MAC on the size and the original data. As encryption algorithm, the AES (Advanced Encryption Standard) is used in CTR (Counter) mode [Dwo01]. As MAC algorithm, CMAC (Cipher-based MAC) [Dwo05] is used, with AES as block cipher.

Since CTR mode is used as encryption, the two keys used for each direction of communication are not allowed to be the same, or else the security is compromised. The key given to the secure connection layer is a vector of octets of any length. The secure connection layer expands the key itself into a key and a counter for the encryption, and a key for the MAC, using the SHA256 hash function.

9.7.3 Key Exchange Layer

The secure connection layer needs a secure key in order to provide the necessary security. The key exchange layer produces such a key using an authenticated Diffie-Hellman key negotiation.

Once the `connected` callback of the key exchange layer is triggered, it first sends its own identity to the other party. When it receives the identity from the other party, it retrieves the public key belonging to that party. A Diffie-Hellman exchange is started, and the key exchange's own private key is used to sign the key exchange. Once the key exchange message from the other party is received, two keys are computed, one for each direction of communication. These keys are then given to the secure connection layer, from which point on the communication is secure. Now the `connected` callback of the higher level protocol is triggered, indicating that a connection has been built and secured.

9.7.4 Multiplex Layer

Like TCP/IP sockets, the key exchange layer is a communication channel between ports at both parties. Two channels may exist between two parties, when both parties decide to connect to the each other at the same time. Zero channels may exist just after starting the protocol, or after a network problem. The multiplex layer provides communication channels between parties, instead of between ports at parties. By automatically closing superfluous connections, it serves as a single communication channel to each party.

9.7.5 Reliable Layer

The multiplex layer provides communication channels that are reliable when connected, but between a disconnect and a new connection, messages may or may not have been received by the other party. The reliable layer has a simplified sliding window protocol, where received packets are acknowledged. If a new connection has been made, any messages that have been sent but not received by the other party are sent again.

Since the reliable layer completely handles reconnecting lost connections, the interface consists of a function to send a message to a party, and a callback reporting that a message is received from some party.

10 Evaluation of the Prototype

In this section, we evaluate the complexity of the code, we discuss possible optimizations, and we measure the performance of the prototype.

10.1 Complexity of the Code

The prototype built is of considerable size: about 7800 lines of C++ code. This may seem to be quite a lot, but with these code, 12 protocol layers are implemented. That makes on average 650 lines of code per protocol. A quarter of this code consists of the message types being translated into classes and signature generators. This code has a low complexity, given a message specification like (**commit**, *p*, *x*, *l*, *d*) augmented with the types of these parameters, it should be very easy to create an automatic code generator. An even larger part of the code consists of very direct translations of the formal specifications. The complexity of this code is higher than the translations of messages, but still the complexity of this code is not that high: it is easily verified against the formal specifications.

By applying constructions like the serialization library and carefully considering how to translate the formal specifications into code, the complexity of the code has been kept low so that it was possible to rapidly implement twelve layers of protocols, and get it to work reliably. Bugs occurred, but they were few and easily reproducible, and easily found by testing each protocol layer separately. Almost every bug was the consequence of either misinterpreting the original specifications, or making mistakes when translating the specification to code. The last category of errors was easy to spot by comparing the specifications to the code.

10.2 Optimizations

In the prototype, each vote is broadcast separately. While this gives a good indication of the latency and throughput of broadcasts, it is unnecessary to broadcast them separately. A bulletin board server could wait for, say, 10 seconds, accumulating votes, and broadcast those 10 votes in one message. This increases the latency by at most 10 seconds, but that may not be a great deal. Depending on how much votes are cast per second, it dramatically increases the throughput. Note that in the prototype, the limiting factor is the time spent on public key operations, network bandwidth was not an issue. Increasing the size of broadcast messages has no effect on the time spent on public key operations.

10.3 Performance

One of the most interesting aspects of building a prototype is that you can measure its performance. Although we did not spent much time on this and our results are therefore not very precise, we do get a general idea of how well this protocol performs.

First, we explain what we measured, and how we measured it. Since the latency is something that is configurable, by adjusting how often the echo multicast protocol sends its **counters** message and the atomic multicast protocol sends its **order** message, we picked a latency of about 4 seconds, which is high enough to not really disturb throughput, but low enough for our bulletin board to be reasonably responsive.

We regarded throughput as the most important aspect of performance. As the number of servers running the bulletin board grow, the throughput will decline. We measured the throughput by letting each server send a message over the atomic multicast protocol at regular intervals. Gradually decreasing the interval times, we determined the point at which the parties could not keep up, and started to get voted out of the group. With this information, we computed how much messages can be sent by the bulletin board per second.

The hardware used for this experiment consisted of a 16-node cluster, where each node has an Intel Pentium 4 processor, running at 3 GHz. Memory and network bandwidth were not the limiting factor, computation time was. Although the hardware is quite fast (at the moment of writing, 2005), our implementation of public key calculations is not optimal, it is estimated that a factor 2 can be gained by improving on the efficiency of the code.

Due to lack of time, our implementation does not include the non-interactive signatures. Being a major improvement on the efficiency of the protocol, we decided to emulate non-interactive signatures, by spending less time verifying signatures. In the echo multicast protocol, a sender of a message eventually sends out a **commit** message containing $\lceil (2n+1)/3 \rceil$ signatures. In a non-interactive signature scheme, only one signature is sent. Therefore, the receiving party would only check one signature. We crippled our implementation so that the receiving party only checks the first signature in the list. Furthermore, with a non-interactive signature scheme, the party building the **commit** message need not check the signatures before combining them, but can do so afterwards, therefore checking only one signature instead of many. We also adapted our implementation to only check one signature. Now we have an implementation that matches the performance of an implementation where non-interactive signatures are used.

We now present how much messages can be broadcast per second:

Parties	Interactive Signatures	Non-Interactive Signatures
4	20/s	40/s
10	5/s	33/s
16	2/s	18/s

We clearly see the quadratic factor with the interactive signatures. Non-interactive signatures give a very good improvement, which will of course be more noticeable when more servers are involved.

With the optimizations presented in the previous section, we can outline how well our bulletin board can perform. Let us take the case where the bulletin board consists of 16 servers, and uses non-interactive signatures. 18 messages may be sent per second, which is about one message per server per second. We already have a latency of about 4 seconds. By introducing a latency of another second, each server can queue messages for a second, and send those messages in one broadcast. Now our bulletin board has a latency of 5 seconds, but a virtually unlimited number of messages may be sent per second. If we tolerate greater latencies, we can increase the number of parties running the bulletin board. In doing so, we can build a secure bulletin board with reasonable latency with 40 servers, of which 13 servers may be corrupted.

11 Conclusion

In this thesis, we studied existing protocols suitable for the implementation of a bulletin board. Comparing their performance characteristics, we selected one and greatly improved on its complexity. We

showed how we converted the original protocols into a nice coherent set of protocols suitable for direct translation into code. We explained how the translation of the code can be done in a very clear and very precise way. This produced a prototype, which we believe is *secure by design*. Finally, we measured its performance, giving an indication of how much parties can be used while maintaining reasonable latency.

References

[AF04] M. Abe and S. Fehr. Adaptively secure feldman VSS and applications to universally-composable threshold cryptography. In *CRYPTO*, 2004.

[AMP+00] Alvisi, Malkhi, Pierce, Reiter, and Wright. Dynamic byzantine quorum systems. In *International Conference on Dependable Systems and Networks (IEEE Computer Society) (replacing both IEEE FTCS (after 29th) and IFIP DCCA (after 7th))*, volume 1, 2000.

[Ben87] Josh Daniel Cohen Benaloh. *Verifiable secret-ballot elections*. PhD thesis, 1987.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. *Lecture Notes in Computer Science*, 2248:514–??, 2001.

[Bol02] Alexandra Boldyreva. Efficient threshold signature, multisignature and blind signature schemes based on the gap-diffie-hellman-group signature scheme, 2002.

[BY86] Josh C Benaloh and Moti Yung. Distributing the power of a government to enhance the privacy of voters. In *PODC '86: Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 52–62, New York, NY, USA, 1986. ACM Press.

[CF85] J. D. Cohen and M. J. Fischer. A robust and verifiable cryptographically secure election scheme. pages 372–382, Portland, 1985.

[CGJ+99] Ran Canetti, Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. Adaptive security for threshold cryptosystems. *Lecture Notes in Computer Science*, 1666:98–115, 1999.

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Proceedings of Eurocrypt '97*, volume 1233 of *Lecture Notes in Computer Science*, page 103, 1997.

[CKPS01] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. *Lecture Notes in Computer Science*, 2139:524+, 2001.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proc. 19th ACM Symp. on Principles of Distributed Computing*, pages 123–132, 2000.

[DK00] I. Damgard and M. Koprowski. Practical threshold RSA signatures without a trusted dealer, 2000.

[Dwo01] Morris Dworkin. NIST SP 800-38A, recommendation for block cipher modes of operation—methods and techniques. National Institute of Standards and Technology/U.S. Department of Commerce, 2001.

[Dwo05] Morris Dworkin. Draft NIST SP 800-38B, recommendation for block cipher modes of operation: The cmac authentication mode—methods and techniques. National Institute of Standards and Technology/U.S. Department of Commerce, 2005.

[Fel87] Paul Feldman. A Practical Scheme for Non-Interactive Verifiable Secret Sharing. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 427–437, Los Angeles, CA, USA, October 1987. IEEE Computer Society, IEEE.

[FMY98] Yair Frankel, Philip D. MacKenzie, and Moti Yung. Robust efficient distributed RSA-key generation. In *The Thirtieth Annual ACM Symposium on Theory of Computing – STOC '98*, pages 663–672, 1998.

[GJKR03] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin. Revisiting the distributed key generation for discrete-log based cryptosystems, 2003.

[JN01] Antoine Joux and Kim Nguyen. Separating Decision Diffie-Hellman from Diffie-Hellman in cryptographic groups. Technical Report 2001/003, 2001.

[KS01] Klaus Kursawe and Victor Shoup. Optimistic asynchronous atomic broadcast. Cryptology ePrint Archive, Report 2001/022, 2001. http://eprint.iacr.org/.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MR97] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578. ACM Press, 1997.

[MR98] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in phalanx. In *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 51. IEEE Computer Society, 1998.

[Ped91] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In *EUROCRYPT*, pages 522–526, 1991.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[Rei94] Michael K. Reiter. Secure agreement protocols: reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 68–80. ACM Press, 1994.

[Rei96] Michael K. Reiter. A secure group membership protocol. *IEEE Trans. Softw. Eng.*, 22(1):31–42, 1996.

[Sch89] Claus P. Schnorr. Efficient identification and signatures for smart cards. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 239–252. Springer-Verlag New York, Inc., 1989.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[Sho00] Victor Shoup. Practical threshold signatures. *Lecture Notes in Computer Science*, 1807:207–??, 2000.