Bern University of Applied Sciences
Engineering and Information Technology

Realization of a Secure Distributed Bulletin Board

by
José Beuchat
jose.beuchat@bluewin.ch

Master Thesis

Bern University of Applied Sciences
Engineering and Information Technology
Department of Computer Sciences

January 30, 2012

Advisor: Prof. Dr. Eric Dubuis
Expert: Prof. Dr. Andreas Steffen

Abstract

A large number of papers make the assumption that *secure bulletin boards* exist. Nevertheless there is no broadly recognized realization.

In this master thesis we identify the requirements and present a solution. Our distributed protocol running at n parties provides the assurance that a message will never be modified or deleted and that the order in which the messages are posted will stay the same. Also, the board is available for reading to everybody and for writing to every authorized user. These properties are true when less than one third of the parties are corrupt or not available.

Contents

Acknowledgment

I would like to thank my advisor *Eric Dubuis* for his continuous support, motivation and guidance. His help was precious and brought this thesis forward.

Furthermore, I would like to express my gratitude to *Severin Hauser*, for his help with the implementation and to *Stephan Fischli* for his support and advices concerning the notation.

Last but not least, I would like to thank my family and my girlfriend *Ana*, who supported me throughout my studies.

1. Introduction

More and more data are published on the Internet everyday. How can we ensure that the displayed content has not been modified? In serious contexts (e.g., e-voting) it is essential to prove the correctness of the data.

Using a secure bulletin board, authorized users will be able to post messages and have the assurance that they will never be changed, moved or deleted. Also, the messages will be available to everyone. Unfortunately, even if the existence of the secure bulletin board is broadly assumed, the information about its requirements or a working solution remain poor.

The goal of this paper is to describe a working solution that produces correct results even in presence of actively corrupt parties. The basis of our work are the paper of R.A. Peters [Pet05] and the protocols described by M.K Reiter in [Rei96] and [Rei94].

This report is organized as follows: first, the possible applications, requirements and actors of a secure bulletin board are identified. In Section 3, we introduce our solution. Section 4 contains a formal description of the necessary protocols. Section 5 describes the way we implemented the protocols and finally, Section 6 is the conclusion.

2. Informal Description of a Bulletin Board

A bulletin board is responsible for publishing something and giving the proof that its content has not been altered.

2.1. Typical Applications

2.1.1. E-voting

Voters want to be sure that their ballots have been counted correctly and that they have the possibility to revoke the vote otherwise. A bulletin board is responsible for publishing the encrypted ballots and providing a receipt. At the end the votes are decrypted and published (without anyone knowing the link from encrypted to decrypted votes). It is then possible to count the decrypted ballots. To ensure that both parts correspond without linking them, zero knowledge proofs described in [Kra07] are used. If a ballot is missing, the voter can prove it by giving his receipt.

2.1.2. Auctions

A bidder who wishes to place a bid wants to receive a proof for it, otherwise anyone could refute his bid. The sequence is very important here.

2.1.3. Auditable Discussion Boards

For evident reasons, it could be interesting to provide a forum with a secure history.

2.1.4. System Logs

Logs are system activities written in text files. It is useful to have security here if we do not trust the logger.

2.1.5. Online Petitions

In the context of online petitions, security could be needed, for example, to prove that the text of a petition hasn't been changed.

2.2. Requirements

The following requirements should be satisfied by every implementation of a bulletin board:

R1. Availability: Each authorized user is able to successfully publish messages on the board. Similarly, everybody should be able to read the content of the board.

R2. Unalterable History: Once published, messages should not be removed or modified without notice. Moreover, no message should be moved to another position and the new ones should be placed at the end.

R3. No single point of failure: There should be no single point of failure. If a component is corrupt or simply stops working for any reason, this should not have any consequence for the system. For the same reason, trusted third parties should be avoided.

R4. Failure Detection: If those properties are not respected, the users are able to prove it.

Those requirements must be fulfilled even in the presence of corrupt components.

2.3. Actors

Three types of actors exist. Some of them are active (modifying the data) and some are not:

Bulletin Board The bulletin board contains a possibly empty list of messages, the *history*. It is responsible for allowing the writers to publish information and making it accessible to any of the readers. The board itself is not allowed to post. The content of the messages (data and metadata) is verified by the board itself before being added to its history. The bulletin board is also responsible for ensuring that its content does not change and is able to detect it if it does.

Writer The writer is active. He is the only user allowed to post information on the bulletin board and is responsible for generating the necessary metadata (e.g., digital signature) to be sent to the board.

Reader The reader is not active but is still very important. Each time he reads the data, the validity of the history is tested.

3. Approach

In this chapter, we introduce the problem we have to solve and the protocols we will use so that the requirements described in Section 2.2 can be satisfied.

3.1. A Distributed Solution

In order to avoid a single point of failure, we need a distributed protocol running at n parties. A party is either *correct* or *faulty*. Correct parties are honest and always respond correctly within some timeout, while faulty parties either respond too late, not at all, or incorrectly. An honest party having connection troubles will be considered faulty so that the protocol can make progress.

As there are many parties, they must reach consensus on the history (the list of messages). As required, our solution must tolerate the presence of faulty components.

In other terms, we want to reliably and atomically multicast messages in a group that possibly contains corrupt members.

3.2. Byzantine Generals Problem

The problem we have to solve is the so-called *Byzantine Generals Problem*. The following extract is taken from [LLP82]:

A reliable computer system must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked–namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the Byzantine Generals Problem.

3.3. Bounds on the Number of Faulty Parties

We cannot expect our solution to work correctly if every party is corrupt. However, we must tolerate a few faulty parties. As shown in [LLP82], the *Byzantine Generals Problem* is solvable if and only if more than two thirds of the generals are loyal. Thus, our protocol tolerates at most $\lfloor \frac{n-1}{3} \rfloor$ faulty parties or, similarly, at least $\lceil \frac{2n+1}{3} \rceil$ parties must be correct.

3.4. Protocols

As represented in Figure 3.1, our solution is composed of several layers of protocols, described later in this section. The lowest layer is the network, which is used to send data to the other parties.

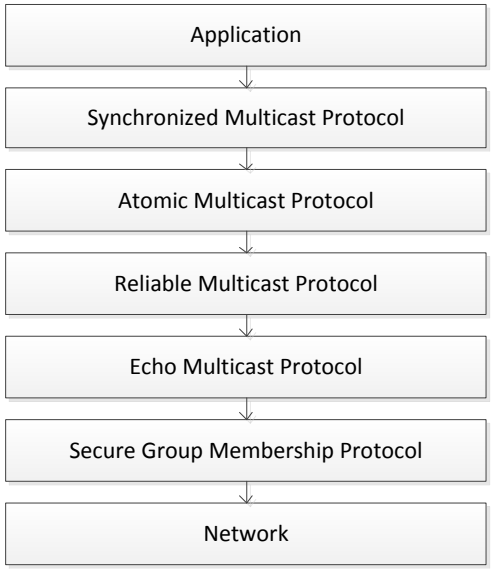

Figure 3.1.: Protocols stack

3.4.1. Rampart

The **secure broadcast channel** used in Rampart [Rei94] is the basis of our work. It is a distributed solution composed of several layers: the Echo Multicast Protocol, the Reliable Multicast Protocol and the Atomic Multicast Protocol. The protocols presented here are more detailed in chapter 4.

The Echo Multicast Protocol

The Echo Multicast Protocol, represented in Figure 3.2, is the core protocol. It is used by honest parties to ensure that all other parties receive a certain message. To simplify the understanding we state that a single party p_0 sends at most one message m. Party p_0 has to convince the other parties that it sent the same m to everyone. It starts by sending a message $\langle$**init**: $m\rangle$ to everyone. When a party p_i receives this message, it answers by digitally signing it: $\langle$**echo**: $m\rangle_{K_i}$. Once p_0 receives $\lceil\frac{2n+1}{3}\rceil$ echoes for m, it sends them to all members in a message $\langle$**commit**: $\{\langle$**echo**: $m\rangle_{K_i}\}\rangle$, giving the proof that m is the message sent to everyone.

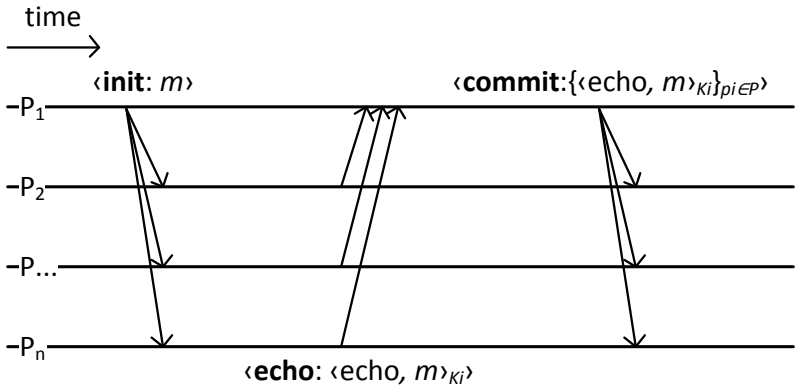

Figure 3.2.: Simplified Echo Multicast

The Reliable Multicast Protocol

The Reliable Multicast Protocol uses the Echo Multicast Protocol and ensures that the received messages are the same at each party, even if a message is sent by a corrupt party.

The Atomic Multicast Protocol

The Atomic Multicast Protocol uses the Reliable Multicast Protocol and ensures that the messages are received in the same order by every party.

3.4.2. Group Membership Protocol

The three layers used in Rampart are built on the top of the Secure Group Membership Protocol described in [Rei96]. With this protocol, each member maintains a view of parties V^x, which is the set of active and responsive parties. Once a member suspects another one of being faulty, it votes to remove it from the group. When enough parties have done the same, the faulty member is removed. A new group view V^{x+1} is formed, which then becomes the current view. Similarly, a new group is formed if enough members want to add a new party. This protocol is detailed in Section 4.4.

3.4.3. Synchronization Protocol

Since we want to offer the possibility to add a new party in the group, an atomic multicast is not enough. New parties or parties that have been removed and later rejoin the group have missed messages. An extra layer is introduced so that joining parties can safely recover messages they did not receive. It is the Synchronized Atomic Multicast Protocol defined in [Pet05] and described in more details in Section4.8.

3.4.4. Application

Additionally to the previous protocols, we need a write, a read and a consolidation protocol so that a multicast can be used to implement a bulletin board. Write and read protocols are used between a client and a party, whereas the Consolidation Protocol is used between emphn parties. They all are built in the application layer and can differ depending on the context.

A client who posts a message on the board will receive a *threshold signature* as receipt. Threshold signatures are made of *signature shares* combined into a single signature. We assume such a solution is available, but this remains out of scope of this document. See [Sho00] for more details.

When a client wants to write something on the board, he randomly selects a party who will broadcast the message and request threshold signature shares. These shares combined form a receipt which is returned to the client as a receipt.

When a client wants to read the messages on the board, it sends a nonce which will be broadcast by a randomly selected party. A signature constructed on the messages and the challenge is returned together with the messages. Since each party has exactly the same sequence of messages, only a signature share needs to be transmitted to the selected party. The messages and the threshold signature are sent by the selected party to the client. This works under the assumption that no writes occur during the read operation.

3.5. A Scenario

When examining a scenario, only the Write, Read, Consolidation and Echo Multicast Protocols are of interest. Indeed, without corrupt parties, the actions performed in the other layers take a constant amount of time.

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

1. Client c sends a message m to some party $p \in P$.
2. Party p sends $\langle \textbf{init}\text{: } m \rangle$ to every $p' \in P$.
3. Each p' answers with a signed message $\langle \textbf{echo}\text{: } m \rangle_{K_{p'}}$.
4. After receiving $\lceil \frac{2n+1}{3} \rceil$ messages, p sends $\langle \textbf{commit}\text{: } \{\langle \textbf{echo}\text{: } m \rangle_{K_{p'}}\}_{p' \in P' \subset P} \rangle$ to every party, where P' is the subset of parties from which messages have been received.
5. After p' receives the commit message, it persists the message, computes a signature share and sends it to p.
6. Party p combines the shares into a single signature and sends it to c.

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

1. Client c chooses a nonce d and sends it to some party $p \in P$.
2. Party p sends the challenge to each $p' \in P$.
3. Each p' creates a signature share over the messages and sends it to p.

4. Party p combines these shares into a single signature and sends it together with the messages to c.

4. Specifications of the Protocols

The protocols presented in this chapter are described in an event-driven way. That means that once a message is sent, it starts the appropriate action at the receiving party or layer. The exact notation is described in Section 4.2.

4.1. Requirements

Additionally to those presented in Section 2.2, our distributed solution has to satisfy the following requirements:

R5. Broadcast: If an honest party sends a message to the bulletin board, every correct party receives that message.

R6. Agreement: Once a message has been successfully sent to the board, every correct party receives the same message.

4.2. Notation

In this section, we introduce the notation used to describe the different protocols.

Parties. The different parties are defined:

p_t: this party.

p_s: the party who sent a message.

p_m: the manager of the actual view.

p_d: a party acting as deputy.

p_c: a party we want to change (add or remove)

Messages. The protocols or parties communicate with each other using messages. The messages are described in the form $\langle$**type**: $*\rangle$. The symbol $*$, used to mean "anything", represents the list of arguments contained in the message: $\langle$**type**: $a_1, ..., a_n\rangle$, $n \geq 0$.

A message can be either *vertical* or *horizontal*. Vertical messages are sent between layers at the same party. Horizontal messages are sent between parties and are assumed to be sent in a reliable and confidential way. In practice, this is done by another protocol layer.

Sending messages to other parties or other layers is done as follow:

send $\langle$**type**: $*\rangle$ to p_i: send a message to another party p_i.

send up $\langle$**type**: $*\rangle$: send a message to one or more upper layer(s).

send down $\langle$**type**: $*\rangle$: send a message to exactly one lower layer.

Receiving messages from other parties or other layers is done as follow:

ON MESSAGE ⟨**type**: ∗⟩ FROM *source*: *action*

When a message containing a signature is received, its validity is implicitly verified. If the signature is wrong, the message is discarded and no action is performed.

Collections. We use two types of collections in our protocols: sets and sequences. The first element of the collections has index 1 and the last has index n. The symbol $\emptyset$ represents an empty collection. Moreover, collections variables start with a capital letter.

Sets are collections of unordered and distinct elements. The following notation is used:

$S = \{e_1, ..., e_n\}$: the set S

$S := S \cup \{e\}$: the element e is added in S

$S := S \setminus \{e\}$: the element e is removed from S

$|S|$: the number of elements in s

Sequences are collections of ordered elements. The following notation is used:

$S = (s[1], ..., s[n])$: the set S

$S[1]$: the first element (the head) of S

$S[2..]$: the tail of S

$S := S \mid e$: the element e is pushed in S

$e,\ S := S[1],\ S[2..]$: the element e is popped from S

$|S|$: the number of elements in S

Timers. So that the protocols do not wait forever on a message that is not coming, timers are introduced. Two different constructions exist:

PERIODICALLY: an action which is periodically executed.

ON TIMEOUT ⟨**identifier**: ∗⟩: triggered a short time after the protocol executed start timer ⟨**identifier**: ∗⟩.

Note that timers and messages have a similar form. To stop a timer if the awaited message is received, the protocol executes stop timer ⟨**identifier**: ∗⟩. The command has no effect if the timer was already triggered.

Hash function. A hash function $f(m)$ is an algorithm or subroutine that maps large data sets m to smaller data sets d and that is collision free in practice. It should not be computationally possible to find two inputs m and m' mapping to same hash value d. $f(m) \neq f(m')$. Moreover, a hash function is one-way. We can not find m if we only know d.

Digital signatures. Digital signatures are in the form $\langle * \rangle_{K_i}$, where $_{K_i}$ is the key of the party p_i and $*$ the content. Note that this does not mean that the content

is sent. For example in a message $\langle$**echo**: $x, l, \langle echo, p_s, x, l, d\rangle_{K_i}\rangle$, the variable d is used in the signature but not sent. This is because the addressee can deduct d from parameters x and l.

In order to prevent a signature to be used for a different purpose or to be re-used, an unique identifier is used. As you will see later, additionally to the parameters, we include an identifier (*echo* in the example above) and the index of the view in each signature.

4.3. Assumptions

Many assumptions are needed in order to achieve our security requirements:

A1. Digital signatures can be verified by everybody but only the owner of the private key is able to produce them.

A2. A solution to publish the public keys exist (e.g., PKI) so that all actors know the public keys of all writers and parties of the board, but private keys remain secret.

A3. As no trusted third party is allowed, each party must be able to generate its own key pair.

A4. Communication between different parties is done using security mechanisms, able to provide privacy, authenticity and integrity.

A5. If hash functions are used, they are collision-resistant. Two distinct terms will almost never result to the same hash value.

A6. The bulletin board is responsible for ensuring the correctness of the data provided by the writers.

A7. Every time a message containing a signature is received, its validity is verified.

A8. In each view V, at most $\lfloor \frac{|V|-1}{3} \rfloor$ parties are faulty.

A9. A threshold signature scheme is available and can be used to generate receipts for the clients.

4.4. The Secure Group Membership Protocol

The Secure Group Membership Protocol described in [Rei96] ensures that in a set of parties P, the honest parties agree on a subgroup of correct parties. For that purpose, each party p_i maintains a view V_i^x, consisting of the currently operational parties. Every time a party is added or removed from the group, the view changes and the index x, denoting the x-th view, is incremented. When a party creates its view V_i^x, view V_i^x is said to be *defined*, and *undefined* otherwise. The protocol assures that all x-th views at each correct party are the same. Therefore, the index i is usually omitted: V^x denotes the x-th view. The first view V^0, is manually configured by an administrator.

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

4.4.1. Informal Description

The parties in the group are totally ordered using their public keys. The lowest rank is 1 and the highest rank is n. The function *rank(p_i)* determines the rank of party p_i. The party with rank n acts as the manager. Its responsibility is to propose updates to the group members. When a party suspects another party p_c to be faulty or wants a new party to be added, it reports it to the manager in a **notify** message. Once the manager received $\lfloor(|V^x|-1)/3\rfloor+1$ same requests, it knows that at least one honest party wants an update and sends a suggestion to the view. An honest party receiving a suggestion from the manager answers with an **ack** message. Once the manager received $\lceil(2|V^x|+1)/3\rceil$ **ack** messages, it sends a proposal to the view. Every party responds with a **ready** message, and after receiving $\lceil(2|V^x|+1)/3\rceil$ **ready** messages, the manager broadcast a **commit** message, upon which a new view is formed. The messages are broadcast as follow: a party p_t receiving a **commit** message sends it to the parties with a $rank = rank(p_t) + r \bmod |V| + 1$, where $0 \leq r \leq \lfloor(|V|-1)/3\rfloor$. As each party sends the message to at least one correct party, we are sure that every party receives the message. Figure 4.1 represents the protocol when the manager is correct:

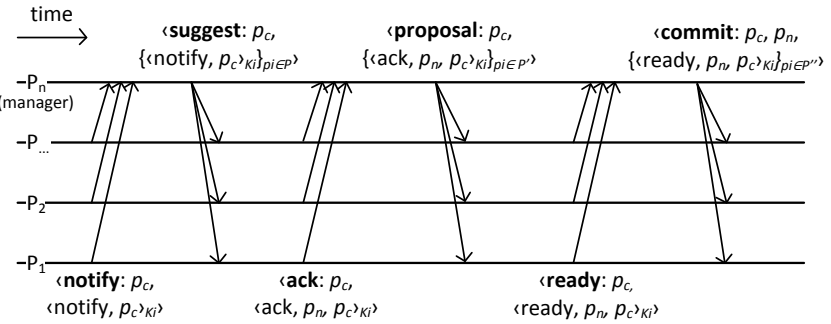

Figure 4.1.: Protocol when the manager is correct

If the manager is suspected of being faulty, a party sends a **deputy** message to p_d, the correct party with the highest rank. When a party receives $\lfloor(|V^x|-1)/3\rfloor+1$ messages of that kind, it sends a **query** message to the view. Each honest party answers with a **last** message, possibly containing the last proposal sent by the previous manager. The deputy p_d sends a **suggest-last** message to the group, upon which each party answers with an **ack** message. From here, the messages are the same as those with a correct manager: once the deputy received $\lceil(2|V^x|+1)/3\rceil$ **ack** messages, it sends a proposal to the view. Every party responds with a **ready** message, and after receiving $\lceil(2|V^x|+1)/3\rceil$ **ready** messages, the deputy sends a **commit** message, upon which a new view is formed. In Figure 4.2, the manager p_n is faulty and replaced by the deputy p_{n-1}. Note that the last proposal *lp* is empty

in the following example:

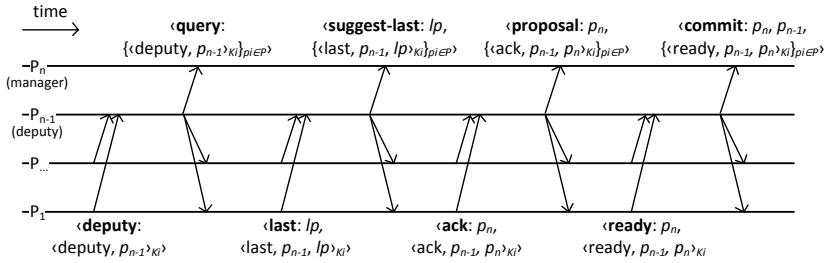

Figure 4.2.: Protocol when the manager is faulty

A corrupt manager may try to convince a party p_i to do one update and another party p_j to do another update. Before a manager can propose an update, it must accumulate $\lceil(2|V^x|+1)/3\rceil$ **ack** messages. To avoid the possibility that a manager obtains two different sets of **ack** messages, an honest party is allowed to create only one **ack** message in each view. Under the assumptions that at most $\lfloor(|V^x|-1)/3\rfloor$ parties are faulty and thus at least $\lceil(2|V^x|+1)/3\rceil$ parties are correct, it is now infeasible for a manager to obtain two different sets of **ack** messages. Even if corrupt parties create two different **ack** messages: $2*(\lfloor(|V^x|-1)/3\rfloor)$, and the correct parties one: $\lceil(2|V^x|+1)/3\rceil$, the sum of them is smaller than the necessary amount to create two different sets of **ack** messages: $2*(\lceil(2|V^x|+1)/3\rceil)$. We thus have the assurance only one change can be orchestrated by the manager in each view.

It may seem that if a party receives a correct **proposal** message, it has enough information to form the new view. Indeed, only one **proposal** message can be formed and using broadcast, we have the assurance that every party receives that proposal. However, there is still a problem, since before the proposal would arrive at every party, a deputy may want to remove the manager. A situation could occur where some parties remove the manager and some other follow the proposal sent by the manager. To prevent this situation, the manager sends the proposal, receives ready responses from the other parties and combines them in a **commit** message. Now if some party receives a **commit** message while a deputy tried to remove the faulty manager, the deputy receives the last proposal with its **query** message and follows that proposal instead of removing the manager. Agreement of the group is now maintained.

The SGM Protocol

The pseudocode in Section 4.4.5 is the translation of the protocol presented in the appendix A of [Rei96].

The **adds** messages, introduced in [Pet05], are used by the Reliable Multicast Protocol to prevent that no progress is made when several parties are corrupt. If several corrupt parties need to be removed in order to deliver a message of the next view, additions of new members to the current view is held until that view is really delivered. Otherwise, corrupt parties could be added and removed over and over again without making progress.

The SGM-View Protocol

The Secure Group Membership Protocol uses the SGM-View Protocol as a subprotocol to manage one particular view. If a new view is created, the protocol finishes and a new instance is created. The subprotocol is initialized with two arguments: the index of the view x and the set of parties V^x in it.
As a general rule, each time a message is received, the view in which it was sent is tested. If it is intended to a future view, it is pushed in the sequence $Defer^x$ of the *SGM* protocol, otherwise it is directly processed. For brevity, this is omitted in the pseudocode.

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

4.4.2. Properties

The protocol ensures the following properties [Pet05]:

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 party $p_j \in V^x$ or a party $p_k \notin V^x$ 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.

4.4.3. Interface

The following messages are received from an upper layer:

$\langle$**faulty**: p_j, $x\rangle$: when a party suspects another party p_j of being faulty in view x.

$\langle$**correct**: p_j, $x\rangle$: when a party wants another party $p_j \notin V^x$ to join the group.

$\langle$**adds**: $x\rangle$: a message defined in [Pet05] so that no party can be added until this message has been sent in view x.

The following message is sent to one or more upper layer(s):

$\langle$**view**: $x\rangle$: when a new group view with index x is created.

4.4.4. Variables

x: the index of the current view, $x \geq 1$

V: sequence containing sets of parties. V^x contains the parties in view x

P: the first set of parties (V^1)

History: set containing each modification of the group

$Defer^x$: sequence containing the messages intended to a future view

Correct: set containing parties we want to add to the group

Faulty: set containing the parties we suspect of being faulty

Adds: boolean variable which is true if an **adds** message has been received in the actual view

ProtocolState: monotonically decreasing variable used to be sure that p_t processes only one message of each following types: **suggest**, **proposal**, **query** and **suggest-last**

LastProposal: the last proposal sent by the manager

MDState: used so that messages are processed in a coherent order by the manager/deputy

4.4.5. Pseudocode

Protocol SGM-View (member role) for party p_t

INITIALLY (x, V^x):
$x := x$
$LastProposal := NIL$
$V := V^x$
$p_m := p \in V^x$ where $rank(p) \geq rank(q)\ \forall q \in V$
$ProtocolState := 3|V|$
$MDState := begin$

ON MESSAGE $\langle$**faulty**: $p_c\rangle$ FROM *an upper layer*:
send $\langle$**notify**: $x, p_c, \langle notify, x, p_t, p_c\rangle_{Kt}\rangle$ to p_m
start timer $\langle$**remove-manager**$\rangle$
if $rank(p) < rank(p_c) \vee faulty(p)\ \forall p \in V$ **then**
Let $p_d \in V$ such that $correct(p_d) \wedge (rank(p_i) \leq rank(p_d) \vee faulty(p_i)\ \forall p_i \in V)$
send $\langle$**deputy**: $x, \langle deputy, x, p_d\rangle_{K_t}\rangle$ to p_d

ON TIMEOUT $\langle$**remove-manager**$\rangle$:
Let $p_d \in V$ such that $correct(p_d) \wedge (rank(p_i) \leq rank(p_d) \vee faulty(p_i)\ \forall p_i \in V)$
send $\langle$**faulty**: $p_d, x\rangle$ to SGM

ON MESSAGE $\langle$**correct**: $p_c\rangle$ FROM *an upper layer*:
send $\langle$**notify**: $x, p_c, \langle notify, x, p_t, p_c\rangle_{K_t}\rangle$ to p_m
start timer $\langle$**remove-manager**$\rangle$

ON MESSAGE $\langle$**suggest**: $x, p_c, NotifySet\rangle$ 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 $\langle$**ack**: $x, p_c, \langle ack, x, p_m, p_c\rangle_{K_t}\rangle$ to p_m

ON MESSAGE $\langle$**proposal**: $x, p_c, AckSet\rangle$ 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 $\langle$**ready**: $x, p_c, \langle ready, x, p_s, p_c\rangle_{K_t}\rangle$ to p_s

ON MESSAGE $\langle$**commit**: $x, p_c, p_d, ReadySet\rangle$ FROM p_s:
if $|ReadySet| = \lceil(2|V| + 1)/3\rceil$ **then**
send $\langle$**commit**: $x, p_c, p_d, ReadySet\rangle$ to p with $rank = rank(p_t) + r \bmod |V| + 1$,
where $0 \leq r \leq \lfloor(|V| - 1)/3\rfloor$
stop timers $\langle$**remove-manager**$\rangle$
send $\langle$**view**: $p_c, p_d, ReadySet\rangle$ to SGM

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

Listing 4.1: Secure Group Membership View

On message $\langle$**suggest-last**: $x, \mathit{LastSet}\rangle$ from p_s:
  if $3\ \mathit{rank}(p_s) - 1 < \mathit{ProtocolState} \wedge |\mathit{LastSet}| = \lceil(2|V|+1)/3\rceil$ **then**
    $\mathit{LowestRank} := |V| + 1$
    $\mathit{LowestUpdate} := p_m$
    for each $\langle p_d, p_c, \mathit{AckSet}\rangle \in \mathit{LastSet}$ **do**
      if $\mathit{rank}(p_s) < \mathit{rank}(p_d) < \mathit{LowestRank} \wedge |\mathit{AckSet}| = \lceil(2|V|+1)/3\rceil$ **then**
        $\mathit{LowestRank} := \mathit{rank}(p_d)$
        $\mathit{LowestUpdate} := p_c$

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

Listing 4.2: Secure Group Membership View (continued)

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

On message $\langle$**notify**: $x, p_c, \langle \mathit{notify},\ x,\ p_s, p_c\rangle_{K_s}\rangle$ from p_s:
  $\mathit{NotifySet}_c := \mathit{NotifySet}_c \cup \{\langle p_s, \langle \mathit{notify},\ x,\ p_s, p_c\rangle_{K_s}\rangle\}$
  if $\mathit{MDState} = \mathit{begin} \wedge |\mathit{NotifySet}_c| = \lfloor(|V|-1)/3\rfloor + 1$ **then**
    send $\langle$**suggest**: $x, p_c, \mathit{NotifySet}_c\rangle$ to **each** $p \in V$
    $\mathit{MDState} := \mathit{sent\text{-}suggest}$

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

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

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

On message $\langle$**last**: $x, \mathit{LastProposal}, \langle \mathit{last},\ x,\ p_t, \mathit{LastProposal}\rangle_{K_s}\rangle$ from p_s:
  $\mathit{LastSet}_{\mathit{LastProposal}} = \mathit{LastSet}_{\mathit{LastProposal}} \cup \{\langle p_s, \mathit{LastProposal}, \langle \mathit{last},\ x,\ p_t, \mathit{LastProposal}\rangle_{K_s}\rangle\}$
  if $\mathit{MDState} = \mathit{sent\text{-}query} \wedge |\mathit{LastSet}_{\mathit{LastProposal}}| = \lceil(2|V|+1)/3\rceil$ **then**
    send $\langle$**suggest-last**: $x, \mathit{LastProposal}, \mathit{LastSet}_{\mathit{LastProposal}}\rangle$ to **each** $p \in V$
    $\mathit{MDState} := \mathit{sent\text{-}suggest}$

Listing 4.3: Secure Group Membership View (continued)

Protocol SGM (member role) for party p_t

INITIALLY (*Parties*):
  $x := 1$
  $History := \emptyset$
  $Defer := \emptyset$
  $Faulty := \emptyset$
  $Correct := \emptyset$
  $P := Parties$
  $V^x := P$
  $View := new\ \langle$**SGM-View**: $x, V^x\rangle$
  $Adds := false$

ON MESSAGE $\langle$**faulty**: $p_c, x'\rangle$ FROM *an upper layer*:
  if $p_c \notin Faulty$ **then**
    $Faulty := Faulty \cup \{p_c\}$
    if $x' = x$ **then**
      send $\langle$**faulty**: $p_c\rangle$ to View

ON MESSAGE $\langle$**correct**: $p_c, x'\rangle$ FROM *an upper layer*:
  if $p_c \notin Correct$ **then**
    $Correct := Correct \cup \{p_c\}$
    if $Adds \wedge x' = x$ **then**
      send $\langle$**correct**: $p_c\rangle$ to View

ON MESSAGE $\langle$**Adds**: $x'\rangle$ FROM *an upper layer*:
  if $x' = x$ **then**
    $Adds := true$
    for each $p_c \in Correct$ **do**
      send $\langle$**correct**: $p_c\rangle$ to View

ON MESSAGE $\langle$**view**: $p_c, p_c, ReadySet\rangle$ 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 $\langle$**history**: $History\rangle$ to p_c
    $V^x := V^{x-1} \cup \{p_c\}$
    $Correct := Correct \backslash \{p_c\}$
  else
    $V^x := V^{x-1} \backslash \{p_c\}$
    $Faulty := Faulty \backslash \{p_c\}$

  $Adds := false$
  $View := new\ \langle$**SGM-View**: $x, V^x\rangle$
  send $\langle$**faulty**: $p\rangle$ to View **for each** $p \in Faulty$
  send **up** $\langle$**view**: $x\rangle$
  while $Defer^x \neq \emptyset$
    $\langle type, parameters, p_s\rangle, Defer^x = Defer^x[1], Defer^x[2..]$
    send $\langle type, parameters\rangle$ with sender p_s to View

Listing 4.4: Secure Group Membership Protocol

```
On message ⟨history: History′⟩ from p_s:
    done := false
    while ⟨x + 1, p_c, p_d, ReadySet⟩ ∈ History′ do
        if done := false then
            if |ReadySet| = ⌈(2 ∗ |V^x| + 1)/3⌉ then
                x := x + 1
                History := Hystory ∪ {⟨x, p_c, p_d, ReadySet⟩}
                if p_c ∉ V^{x−1} then
                    send ⟨history: History⟩ to p_c
                    V^x := V^{x−1} ∩ {p_c}
                    Correct := Correct\{p_c}
                else
                    V^x := V^{x−1}\{p_c}
                    Faulty := Faulty\{p_c}
                if p_c = p_t ∧ p_c ∉ V^{x−1}
                    send ⟨commit: p_c, p_d, ReadySet⟩ to each p ∈ V^x
                    View := new ⟨SGM-View: x, V^x⟩
                    done = true
            else
                done := true
```

Listing 4.5: Secure Group Membership Protocol (continued)

4.5. The Echo Multicast Protocol

The Echo Multicast Protocol is the core component of the Reliable and Atomic Multicast protocols. It ensures that each echo multicast message sent by p in the view x are the same at each honest party. In absence of membership changes, a reliable multicast reduces to a single echo multicast.

4.5.1. Informal Description

To expand the protocol presented in Section 3.4.1 so that different parties can send multiple messages in different views, we include the initiator p_1, the index of the view x and a sequence number l in the **echo** message. A message digest is also used in **init** and **echo** to reduce the amount of data transmitted on the network. Now, if a party multicasts a message, it stores its index l and each party also stores the number of messages received from other parties in a variable l_p^x. When a party p_1 requests signatures on a particular message m on index l, each party answers with the message $\langle$**echo**: $x, l, \langle echo, p_1, x, l, f(m)\rangle_{K_i}\rangle$. This way, using x, p, l and $f(m)$, each message is unique and identifiable. This expanded version of the protocol is presented in Figure 4.3.

Using index l and a variable $c_{p_i}^x$, we can also maintain a FIFO delivering order of all messages sent by a given party p_i. When a message $\langle$**commit**: $x, l, m, EchoSet_l^x\rangle$ is received, it is stored in a $Commits^x$ set. The variable $c_{p_i}^x$ contains the number of messages coming from p_i which have been delivered. We thus know the index of the next message to be delivered from party p_i. If the last message from p_i had index z, messages with an index higher than $z + 1$ are queued until the message with index $z + 1$ is received.

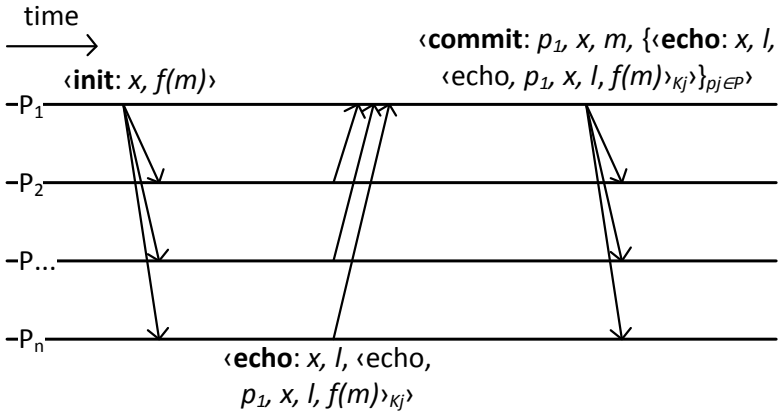

Figure 4.3.: Echo Multicast

After delivering a message, it stays in the $Commits^x$ until it is stable. A message is stable if every party has delivered it and thus added its index to $c^x_{p_i}$. The parties periodically send their own $c^x_{p_i}$ to the other members of the group in a message $\langle$**counters**: $\{c^x_{p_i}\}_{p_i \in SGM.V^x}\rangle$. Each party records those values in a variable $c^x_{p_j,p_i}$. When a party has a message from p_i with index l in its $Commits^x$ and $c^x_{p_j,p_i}$ is at least l for each p_j, it concludes that it is stable, sends up a message $\langle$**e-mcast-stable**: $p_i, x, m\rangle$ and removes the corresponding **commit** message from the set $Commits^x$.

If a message stays in the $Commits^x$ for too long, it is sent to the parties which have not yet delivered it. If after another timeout, some parties have still not delivered it, they are considered faulty and voted out from the group.

Using $\lceil(2|V^x|+1)/3\rceil$ signatures, we have the confidence that if at most $\lfloor(|V^x|-1)/3\rfloor$ members are corrupt, a majority of honest members of V^x echoed m. A party trying to convince a member that m is the message and another member that m' is the message would have to obtain two sets of $\lceil(2|V^x|+1)/3\rceil$ signatures each. Because a correct party provides only one signature and a majority of correct parties echoed m, it is thus impossible to create those two different sets. Even if corrupt parties create two different signatures: $2*(\lfloor(|V^x|-1)/3\rfloor)$, and the correct parties one: $\lceil(2|V^x|+1)/3\rceil$, the sum of them is smaller than the necessary amount to create two different sets of signatures: $2*(\lceil(2|V^x|+1)/3\rceil)$.

4.5.2. Properties

The protocol ensures the following properties [Pet05]:

1. If p_1 is honest and some honest party transmits $\langle$**e-mcast**: $p,\ x,\ m\rangle$ to an upper layer, then the Echo Multicast Protocol of p_1 received a message $\langle$**e-mcast**: $x,\ m\rangle$ from an upper layer.

2. If the l-th message of the form $\langle$**e-mcast**: $p,\ x,\ *\rangle$ at two honest parties are $\langle$**e-mcast**: $p,\ x,\ m\rangle$ and $\langle$**e-mcast**: $p,\ x,\ m'\rangle$, then $m = m'$.

4.5.3. Interface

The following message is received from an upper layer:

⟨**e-mcast**: x, m⟩: to echo-multicast a message m in view x.

The following messages are sent to one or more upper layer(s):

⟨**e-mcast**: p, x, m⟩: sent when an echo-multicast message from party p in view x has been received by p_t.

⟨**e-mcast-stable**: p, x, m⟩: sent when an echo-multicast message from party p in view x has been received by all members of V^x .

4.5.4. Variables

l^x: the number of messages echo multicast by this party in view x

l_p^x: the number of messages echo multicast by the other parties in view x

lm^x: pairs of indexes and messages echo multicast by this party in view x

lm_l^x: the l-th message echo multicast by this party in view x

c_i^x: for each party p_i, the index of the last message from p_i delivered by this party in view x

$c_{j,i}^x$: the list c_i^x at each party p_j as broadcast by p_j

s_p^x: for each party p, the index of the last message that is stable in view x

$EchoSet_l^x$: A set containing tuples made of a party and a signature.

$Commits^x$: the set of messages delivered in view x but not yet stable

4.5.5. Pseudocode

Protocol EMP for party p_t

INITIALLY (*SecureGroupMembershipProtocol*):
  $SGM := SecureGroupMembershipProtocol$
  $l^{SGM.x} := 0$
  $lm^{SGM.x} := 0$
  for each $p \in SGM.V^{SGM.x}$ **do**
    $l_p^{SGM.x} := 0$
    $c_p^{SGM.x} := 0$
    $c_{p,p_i}^{SGM.x} := 0$ **for each** $p_i \in SGM.V^{SGM.x}$
    $s_p^{SGM.x} := 0$

ON MESSAGE $\langle$**e-mcast**: $x, m\rangle$ FROM *an upper layer*:
  $l^x := l^x + 1$
  $lm_{l^x}^x := m$
  for each $p \in SGM.V^x$ **do**
    send $\langle$**init**: $x, f(m)\rangle$ to p

ON MESSAGE $\langle$**init**: $x, d\rangle$ FROM p_s:
  verify $p_s \in SGM.V^x$
  $l_{p_s}^x := l_{p_s}^x + 1$
  send $\langle$**echo**: $x, l_{p_s}^x, \langle echo, p_s, x, l_{p_s}^x, d\rangle_{K_t}\rangle$ to p_s

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

ON MESSAGE $\langle$**commit**: $x, l, m, EchoSet_l^x\rangle$ 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$(p_s, x, l, m, EchoSet_l^x)$

ON MESSAGE $\langle$**view**: $x\rangle$ FROM *a lower layer*:
  $l^x := 0$
  $lm^x := \emptyset$
  for each $p \in SGM.V^x$ **do**
    $l_p^x := 0$
    $c_p^x := 0$
    $c_{p,p_i}^x := 0$ **for each** $p_i \in SGM.V^x$
    $s_p^x := 0$

Listing 4.6: Echo Multicast Protocol

Protocol EMP (stability) for party p_t

PERIODICALLY:
send $\langle$**e-mcast**: $SGM.x, \langle$**counters**: $\{c^x_{p_i}\}_{p_i \in SGM.V^x}\rangle\rangle$ to *self*

ON MESSAGE $\langle$**e-mcast**: $p_s, x, \langle$**counters**: $\{c'^x_{p_i}\}_{p_i \in SGM.V^x}\rangle\rangle$ FROM *self*:
$c^x_{p_j,p_i} := c'^x_{p_i}$ **for each** $p_i \in SGM.V^x$
for each s where $s^x_p < s \leq (\downarrow q \in SGM.V^x :: c^x_{q,p})$ **do**
stop timer $\langle$**unstable**: $p, x, s\rangle$
stop timer $\langle$**remove-unstable**: $p, x, s\rangle$
$s^x_p := s$
retrieve m and $EchoSet^x_l$ from $Commits^x$
$Commits^x := Commits^x \setminus \langle p, x, s, m, EchoSet^x_l\rangle$
send **up** $\langle$**e-mcast-stable**: $p, x, m\rangle$

ON TIMEOUT $\langle$**unstable**: $p, x, l\rangle$:
retrieve m and $EchoSet^x_l$ from $Commits^x$
for each p' where $l > c^x_{p',p}$ **do**
send $\langle$**make-stable**: $p, x, l, m, EchoSet^x_l\rangle$ to p
start timer $\langle$**remove-unstable**: $p, x, l\rangle$

ON TIMEOUT $\langle$**remove-unstable**: $p, x, l\rangle$:
for each p' where $l > c^x_{p',p}$ **do**
send **down** $\langle$**faulty**: $p'\rangle$

ON MESSAGE $\langle$**make-stable**: $p, x, l, m, EchoSet^x_l\rangle$ FROM p_s:
for each $x' \in \{0, 1, ..., SGM.x\}$ **do**
verify p $\in SGM.V^{x'}$

AddToCommit$(p, x, l, m, EchoSet^x_l)$

Listing 4.7: Echo Multicast Protocol (Continued)

Protocol EMP (procedures) for party p_t

PROCEDURE AddToCommit$(p_s, x, l, m, EchoSet^x_l)$:
if $c_{p_s} < l \wedge \langle p_s, x, l, m, EchoSet^x_l\rangle \notin Commits^x$ **then**
verify $|EchoSet^x_l| = \lceil(2|SGM.V^x| + 1)/3\rceil \wedge \langle p_i, p_s\rangle \in SGM.V^x \; \forall \langle p_i, p_s\rangle \in EchoSet^x_l$

$Commits^x := Commits^x \cup \{p_s, x, l, m, EchoSet^x_l\}$
start timer $\langle$**unstable**: $p_s, x, l\rangle$
while $\langle p, c^x_p + 1, m', EchoSet^x_l\rangle \in Commits^x$ **do**
send **up** $\langle$**e-mcast**: $p, x, m'\rangle$
$c^x_p := c^x_p + 1$

Listing 4.8: Echo Multicast Protocol (Continued)

4.6. The Reliable Multicast Protocol

The Reliable Multicast Protocol offers the possibility to send messages to every member in a group, such that each group member has the assurance that every other party has received the same message.

It uses the Echo Multicast Protocol and Secure Group Membership Protocol. In absence of a membership change, a reliable multicast reduces to a single echo

multicast. But if a party is removed or added to the group, we need a solution for the messages sent in the old view but not yet stable. The Echo Multicast Protocol sends and receives messages in a specific view. The Reliable Multicast Protocol gives us the assurance that all those messages will be stably delivered to the upper layer and not held forever in the set *Commits* of the Echo Multicast Protocol.

4.6.1. Informal Description

The protocol uses a variable y, representing the latest view delivered to the higher protocols. It normally has the same value as the x of the Secure Group Membership Protocol, except during a membership change, during which it lags behind. The variable z represents the views for which messages are still accepted. Again, during membership changes, it lags behind the x of the Secure Group Membership Protocol. Views with an index lower than z are said *closed*.

When a message m has to be sent, it is echo multicast in view y. If a message m is received, we verify if it is intended for an open view (not lower than z) and then if it is intended for view y. If so, it is immediately delivered to the upper layer, otherwise, if intended for a future view, it is placed in a sequence and delivered once the corresponding view is installed.

When a membership change occurs, each party echo multicasts a message $\langle$**end**$\rangle$ in the old view, telling other parties that it is the last messages it sends in it. After receiving the same message from every party (except the one that was possibly removed), the variable z is incremented, so that no more message is accepted for the closed view and a message $\langle$**flush**: *Commits*$\rangle$ is sent in the new view. This message contains all the entries in the set *Commits* of the Echo Multicast Protocol. Those entries are messages of the old view that are not stable yet. Doing so, an agreement is obtained on the messages belonging to the old view. Once every party sent a **flush** message, the index y is incremented and transmitted to the upper layer in a **r-mcast-view** message.

In the case where a party does not send a **flush** or an **end** message after a given period of time, it is voted out of the group. If a party is voted out during another membership change, it is assumed to have sent an **end** message and an empty **flush** message.

4.6.2. Properties

The protocol ensures the following properties [Pet05]:

Integrity For all honest p and m, an honest party sends up $\langle$**r-mcast**: *p, m*$\rangle$ in view x at most the number of times that p sent down $\langle$**r-mcast**: *m*$\rangle$ in view x.

Uniform Agreement If q is an honest member of V^{x+k} and an honest p sends up $\langle$**r-mcast**: *r, m*$\rangle$ in view x, then q sends up $\langle$**r-mcast**: *r, m*$\rangle$ in view x.

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

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

4.6.3. Interface

The following message is received from an upper layer :

⟨**r-mcast**: m⟩: to reliably multicast a message m the actual view.

The following messages are sent to one or more upper layer(s):

⟨**r-mcast**: p, m⟩: message sent when a reliably multicast message is received from p.

⟨**r-mcast-view**: x⟩: when a new view with index x is received from the Secure Group Membership Protocol and all message sent in the past views were delivered.

4.6.4. Variables

y: the latest view delivered to the higher protocols

z: the views for which messages are still accepted

id^x: the index used to identify the messages sent in view x

id_p^x: the index used to identify thes messages sent by party p in view x

$NotReceivedEnd^x$: set of parties who are expected to send an **end** message in view x but have not done it yet

$NotReceivedFlush^x$: set of parties who are expected to send an **flush** message in view x but have not done it yet

$Defer^x$: sequence containing messages received for a view x which is not defined yet

4.6.5. Pseudocode

Protocol RMP for party p_t

INITIALLY (*SecureGroupMembershipProtocol*):
  SGM := *SecureGroupMembershipProtocol*
  EMP := *new* $\langle$**EMP**: $SGM\rangle$
  $y := SGM.x - 1$
  $z := SGM.x$
  $id^y := 0$
  $id_p^y := 0$ **for each** $p \in SGM.V[z]$
  $NotReceivedFlush^z := SGM.V[z]$
  $NotReceivedEnd = \emptyset$
  Join()

ON MESSAGE $\langle$**r-mcast**: $m\rangle$ FROM *an upper layer*:
  send **down** $\langle$**e-mcast**: $SGM.x, \langle$**r-msg**: $id^{SGM.x}, m\rangle\rangle$
  $id^{SGM.x} = id^{SGM.x} + 1$

ON MESSAGE $\langle$**e-mcast-stable**: $p_s, x', \langle$**r-msg**: $id', m\rangle\rangle$ FROM *a lower layer*:
  if $z \leq x'$ and $id_{p_s}^{x'} < id'$ **then**
    $id_{p_s}^{x'} := id'$
    if $x' = y$ **then**
      send **up** $\langle$**r-mcast**: $p_s, m\rangle$
    if $x' > y$ **then**
      $Defer^{x'} := Defer^{x'} \mid \langle p_s, m\rangle$

ON MESSAGE $\langle$**view**: $x\rangle$ FROM *a lower layer*:
  $id^x := 0$
  $id_p^x := 0$ **for each** $p \in SGM.V^x$
  $NotReceivedFlush^x := SGM.V^{x-1} \cap SGM.V^x$
  if $p_t \notin SGM.V^{x-1}$ **then**
    Join()
  else
    $NotReceivedEnd^{x-1} := SGM.V^{x-1} \cap SGM.V^x$
    for each $x' \in \{x+1, ..., SGM.x - 1\}$ **do**
      ReceivedEnd($SGM.V^{x-1} \backslash SGM.V^x, x' - 1$)
      ReceivedFlush($SGM.V^{x-1} \backslash SGM.V^x, x'$)

  send **down** $\langle$**e-mcast**: $x - 1, \langle$**end**$\rangle\rangle$
  start timer $\langle$**end**: $x - 1\rangle$

ON TIMEOUT $\langle$**end**: $x'\rangle$:
  for each $p \in NotReceivedEnd^{x'}$ **do**
    send **down** $\langle$**faulty**: $p, x' + 1\rangle$

ON MESSAGE $\langle$**e-mcast**: $p_s, x', \langle$**end**$\rangle\rangle$ FROM *a lower layer*:
  ReceivedEnd($\{p_s\}$, x')

ON TIMEOUT $\langle$**flush**: $x'\rangle$:
  for each $p \in NotReceivedFlush^{x'}$ **do**
    send **down** $\langle$**faulty**: $p, x'\rangle$

Listing 4.9: Reliable Multicast Protocol

```
ON MESSAGE ⟨e-mcast: p_s, x', ⟨flush: Commits⟩⟩ FROM a lower layer:
    verify p_s ∈ NotReceivedFlush^x'
    if x' = 1 or p_t ∈ SGM.V^(x-1) then
        for each ⟨p, x' − 1, l, m, Signature⟩ ∈ Commits do
            EMP.AddToCommit(p, x' − 1, l, m, Signature)

    ReceivedFlush({p_s}, x')
```

Listing 4.10: Reliable Multicast Protocol (continued)

```
Protocol RMP (procedures) for party p_t

PROCEDURE ReceivedEnd(P, x'):
    NotReceivedEnd^x' := NotReceivedEnd^x' \ P
    while NotReceivedEnd^z = Ø do
        stop timer end, z
        z := z + 1
        send down ⟨e-mcast: z, ⟨flush: EMP.Commits^(z−1)⟩⟩
        start timer ⟨flush: z⟩

PROCEDURE ReceivedFlush(P, x'):
    NotReceivedFlush^x' := NotReceivedFlush^x' \ P
    while NotReceivedFlush^(y+1) = Ø do
        stop timer ⟨flush: y⟩
        y := y + 1
        send down ⟨adds: y⟩
        send up ⟨r-mcast-view: y⟩
        while Defer^y ≠ Ø do
            ⟨p, m⟩, Defer^y := Defer^y[1], Defer^y[2..]
            send up ⟨r-mcast: p, m⟩

PROCEDURE Join():
    send down ⟨e-mcast: y + 1, ⟨flush: Ø⟩⟩
    id y = 0 then
        start timer⟨flush: y + 1⟩
```

Listing 4.11: Reliable Multicast Protocol (Continued)

4.7. The Atomic Multicast Protocol

The Atomic Multicast Protocol adds the property that correct members deliver messages in the same order. The protocol uses the Reliable Multicast Protocol to reliably send each message to all members.

4.7.1. Informal Description

This protocol uses the Reliable Multicast Protocol to multicast messages. Once a message m is received from party p it is added to a sequence $Pending_p$. A designated group member, the *sequencer*, periodically sends an **order** message, indicating the order in which the messages are to be delivered. The *sequencer*, determined by the function seq(x), keeps a sequence $Senders^x$, which is a sequence of parties. When receiving a message ⟨**order**: *Senders*⟩, a party takes the first party p in *Senders* and the first message in $Pendings_p$, and delivers them in a message ⟨**a-mcast**: *p, m*⟩.

Then, it takes the second party of the sequence and so on. Doing so, 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 the order in which the queued messages will be delivered.

4.7.2. Interface

The following message is received from an upper layer:

⟨**a-mcast**: m⟩: to atomically multicast a message m.

The following messages are sent to one or more upper layer(s):

⟨**a-mcast**: p,m⟩: message sent when an atomically multicast message m is received from p.

⟨**a-mcast-view**: x⟩: when a new view with index x is received and all messages sent in the old view are delivered.

4.7.3. Properties

The protocol ensures the following properties [Pet05]:

Integrity For all honest p and m, an honest party sends up ⟨**a-mcast**: p, m⟩ in view x at most the number of times that p sent down ⟨**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 up ⟨**a-mcast**: r, m⟩ in view x, then q sends up ⟨**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 up ⟨**a-mcast-view**: x⟩.

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

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

4.7.4. Variables

$Senders^x$: sequence of parties keeped by the sequencer in view x

$Pending_s^x$: sequence of messages received from p_s by each party in view x

$Order^x$: sequence of parties used to determine the order in which messages are delivered

4.7.5. Pseudocode

Protocol AMP for party p_t

INITIALLY (*SecureGroupMembershipProtocol*):
  SGM := *SecureGroupMembershipProtocol*
  RMP := *new* ⟨**EMP**: SGM⟩

ON MESSAGE ⟨**a-mcast**: m⟩ FROM *an upper layer*:
  send **down** ⟨**r-mcast**: ⟨**a-msg**: m⟩⟩

ON MESSAGE ⟨**r-mcast**: p_s, ⟨**a-msg**: m⟩⟩ FROM *a lower layer*:
  $Pending_s^{RMP.y}$:= $Pending_s^{RMP.y} \mid m$
  if $seq(RMP.y) = p_t$ **then**
    $Senders^{RMP.y}$:= $Senders^{RMP.y} \mid \{p_s\}$

  start timer ⟨**sequence**: $p_s, m, RMP.y$⟩

ON TIMEOUT ⟨**sequence**: p_s, m, x⟩
  if $x = SGM.x$ **then**
    send **down** ⟨**faulty**: $seq(x), x$⟩

PERIODICALLY:
  if $seq(RMP.y) = p_t$ **then**
    send **down** ⟨**r-mcast**: ⟨**order**: $Senders^{RMP.y}$⟩⟩
    $Senders^{RMP.y} = \emptyset$

ON MESSAGE ⟨**r-mcast**: $Seq(RMP.y)$, ⟨**order**: $Senders$⟩⟩ FROM *a lower layer*:
  while $Senders \neq \emptyset$ **do**
    $p_i, Senders$:= $Senders[1], Senders[2..]$
    if $p_i \in SGM.V[RMP.y]$ **then**
      $Order^{RMP.y}$:= $Order^{RMP.y} \mid p_i$

  while $Order^{RMP.y} \neq \emptyset$ and $Pending^{RMP.y}{}_{Order^{RMP.y}[1]} \neq \emptyset$ **do**
    $p_s, Order^{RMP.y}$:= $Order^{RMP.y}[1], Order^{RMP.y}[2..]$
    $m, Pending_{p_s}^{RMP.y}$:= $Pending_{p_s}^{RMP.y}[1], Pending_{p_s}^{RMP.y}[2..]$
    send **up** ⟨**a-mcast**: p_s, m⟩
    stop timer ⟨**sequence**: $p_s, m, RMP.y$⟩

ON MESSAGE ⟨**r-mcast-view**: x⟩ FROM *a lower layer*:
  for each $p \in$ deterministically ordered $SGM.V[x-1]$ **do**
    while $Pending_p \neq \emptyset$ **do**
      $m, Pending_p$:= $Pending_p[1], Pending_p[2..]$
      send **up** ⟨**a-mcast**: p, m⟩
      stop timer ⟨**sequence**: $p, m, x-1$⟩

  send **up** ⟨**a-mcast-view**: x⟩

Listing 4.12: Atomic Multicast Protocol

4.8. The Synchronized Atomic Multicast Protocol

Members that have been offline for a moment or that just joined the group need to synchronize their messages. This protocol is responsible to do it so that all honest parties eventually receive the same messages in the same order. If a member has always been online, this protocol reduces to an Atomic Multicast Protocol and saving messages in a sequence.

4.8.1. Informal Description

A party p that did not receive messages because it was voted out of the group or because it just joined it needs to synchronize. The party goes into the *joining* mode. It starts by multicasting a message $\langle$**request-hashes**: $b\rangle$, where b is the amount of messages it already received. In answer, the other parties respond with a message $\langle$**reply-hash**: $e,\ h\rangle$ containing the amount of messages they possess and a hash on the messages missing at p. Since the messages are received in the same order by correct parties, party p eventually receives at least $\lfloor(|V|-1)/3\rfloor+1$ identical hashes. Party p stores this hash in a variable *hash*. It then randomly selects a party from which it requests the missing messages by sending a message $\langle$**request-messages**: $b,\ e\rangle$. An honest party responds by sending a message $\langle$**reply-messages**: *Messages[b+1, e]*$\rangle$. Once received, a new hash is created with the received messages and compared with the variable *hash*. If the party does not answer or if the messages are not correct, another party is selected at random, until p obtains correct messages.

While waiting for the messages, p saves any message received from the Atomic Multicast Protocol in a sequence *Save*. Once the missing messages are obtained and delivered, the messages in *Save* are also delivered.

4.8.2. Interface

The following message is received from an upper layer:

$\langle$**s-mcast**: $m\rangle$: to atomically multicast a synchronizable message m.

The following messages are sent to one or more upper layer(s):

$\langle$**s-mcast**: p,$m\rangle$: sent when a synchronizable atomically multicast message m is received from p.

$\langle$**s-mcast-view**: $x\rangle$: sent when a new view with index x is received and all messages sent in the old view are delivered.

4.8.3. Properties

The protocol ensures the following properties [Pet05]:

Integrity For all honest p and m, an honest party sends up $\langle$**s-mcast**: $p,\ m\rangle$ in view x at most the number of times that p sent down $\langle$**s-mcast**: $m\rangle$ 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 up $\langle$**s-mcast**: $r,\ m\rangle$ in view x, then q sends up $\langle$**s-mcast**: $r,\ m\rangle$ in view x.

Validity-1 If p is an honest member of V^{x+k} for all $k \geq k'$, then p sends up $\langle$**s-mcast-view**: $x\rangle$.

Validity-2 If p and q are honest members of V^{x+k} for all $k \geq k'$ and p sends down $\langle$**s-mcast**: $m\rangle$ in view x, then q sends up $\langle$**s-mcast**: p, $m\rangle$ in view x.

Order If q is an honest member of V^{x+k} for all $k \geq k'$ for a k' and an honest p sends up $\langle$**s-mcast**: r, $m\rangle$ before $\langle$**s-mcast**: r', $m'\rangle$ in view x, then q sends up $\langle$**s-mcast**: r, $m\rangle$ before $\langle$**s-mcast**: r', $m'\rangle$ in view x.

4.8.4. Variables

Messages: sequence of messages delivered to the upper layer

Hashes: collection of **reply-hash** messages

hash: hash of the missing messages at p_t

end: the number of messages already received by p_t

member: true if party p_t is member of the actual view

joining: true if p_t just joined the group and needs to synchronize

Save: sequence of new messages received when p_t is synchronizing

NotReceivedHash : set of parties who did not send a **reply-hash** message yet

4.8.5. Pseudocode

Protocol SAMP for party p_t

INITIALLY (*SecureGroupMembershipProtocol*):
    SGM := *SecureGroupMembershipProtocol*
    AMP := *new* $\langle$**AMP**: $SGM\rangle$
    Messages := $\emptyset$
    member := $p_t \in SGM.V(SGM.x)$
    joining := *false*
    hash := *NIL*
    end := *NIL*

ON MESSAGE $\langle$**s-mcast**: $m\rangle$ FROM *an upper layer* :
    send **down** $\langle$**a-mcast**: $\langle$**s-msg**: $m\rangle\rangle$

ON MESSAGE $\langle$**a-mcast**: $p_s, \langle$**s-msg**: $m\rangle\rangle$ FROM *a lower layer* :
    if *joining* **then**
        Save := *Save* | $\langle p_s, m\rangle$
    else
        send **up** $\langle$**s-mcast**: $p_s, m\rangle$
        Messages := *Messages* | $\langle p_s, m\rangle$

Listing 4.13: Synchronized Atomic Multicast Protocol

```
On message ⟨a-mcast-view: x⟩ from a lower layer:
    if p_t ∉ SGM.V(x) and member then
        member := false

    if p_t ∈ SGM.V(x) and ¬member then
        member := true
        joining := true
        Save := ∅
        hashses := ∅
        hash := ∅
        NotReceivedHash := SGM.V(x)
        stop timer ⟨messages⟩
        send down ⟨a-mcast: ⟨request-hashes: |Messages|⟩⟩

On message ⟨a-mcast: p_s, ⟨request-hashes: b⟩⟩ from a lower layer:
    if ¬joining then
        send ⟨reply-hash: |Messages|, hash on Messages[b + 1, |Messages|]⟩ to p_s

On message ⟨reply-hash: e, h⟩ from p_s:
    if joining ∧ p_s ∈ NotReceivedHash then
        NotReceivedHash := NotReceivedHash\p_s
        Hashes := Hashes ∪ ⟨e, h⟩
        if ⟨end, hash⟩ occurs ⌊(|SGM.V(SGM.x)| − 1)/3 + 1⌋ times in Hashes then
            randomly select a p
            send ⟨request-messages: |Messages|, end⟩ to p
            start timer ⟨Messages⟩

On message ⟨request-messages: b, e⟩ from p_s:
    if |Messages| ≥ e then
        send ⟨reply-messages: Messages[b + 1, e]⟩ to p_s

On message ⟨reply-messages: M⟩ from p_s:
    stop timer ⟨Messages⟩
    if hash = hash on M and |M| = end − |Messages| then
        while M ≠ ∅ do
            ⟨p_s, m⟩, M := M[1], M[2..]
            Messages := Messages | ⟨p_s, m⟩
            send up ⟨s-mcast: p_s, m⟩
        while Save ≠ ∅ do
            ⟨p_s, m⟩, Save := Save[1], Save[2..]
            Messages := Messages | ⟨p_s, m⟩
            send up ⟨s-mcast: p_s, m⟩
        joining := false
    else
        randomly select another p
        send ⟨request-messages: |Messages|, end⟩ to p
        start timer ⟨messages⟩

On timeout ⟨messages⟩
    randomly select another p
    send ⟨request-messages: |Messages|, end⟩ to p
    start timer ⟨messages⟩
```

Listing 4.14: Synchronized Atomic Multicast Protocol (continued)

4.9. The Board Protocol

This section describes three protocols. The first one is the Consolidation Protocol. It receives messages from clients, broadcasts them to the other parties, persists them and returns a receipt to the client. It is also responsible for answering to read requests. The second one is the Write Protocol, used by clients to post messages and the third one, the Read Protocol is used by clients to read the content of the board.

Only authorized users should be able to post messages. It can be achieved using signatures but this is outside the scope of this paper. Clients are therefore assumed to be authorized to post and read.

4.9.1. Informal Description

When a party p receives a message m from a client c, it multicasts it to the members of the view. Upon receiving the multicast message, each party adds it in a sequence *Messages* and sends a signature on m to p. Party p then combines the signatures and presents them to the client c as a proof that m has been posted. It is the receipt. If the client does not receive it in time, it selects another party and try to post his message again.

When a client c wants to read messages, it contacts one of the parties, which will request a signature on the messages to the other parties. At the end, it sends the messages and the signatures to the reader.

Note that those protocols can differ depending on the context. For example, in an e-voting context, the number of votes casted may want to be restricted, servers must be closed for writing after a certain date, etc.

4.9.2. Properties

The protocol ensures the following properties:

Availability Every authorized client c is allowed to post messages on the board and everybody is allowed to read the content of the board.

Failure Detection Every time a client c sends a message m to party p, it receives a receipt and is able to prove it if the content of the board has changed.

Variables

Messages: sequence of messages delivered by the Synchronized Atomic Multicast Protocol

$Signatures_m^c$: set of signatures received from the other parties when a message m sent by a client c has been delivered

$Read_c$: set of signatures received from the other parties after a read request for client c

Pseudocode

Protocol CP for party p_t

INITIALLY (*SecureGroupMembershipProtocol*):
    $SGM := SecureGroupMembershipProtocol$
    $Messages := \emptyset$
    $Signatures := \emptyset$
    $Read := \emptyset$

ON MESSAGE $\langle$**message**: $m\rangle$ FROM c:
    verify that c is authorized to post and that m is valid
    send **down** $\langle$**s-mcast**: $\langle$**msg**: $c, m\rangle\rangle$

ON MESSAGE $\langle$**s-mcast**: $p_s, \langle$**msg**: $c, m\rangle\rangle$ FROM *a lower layer*:
    verify that c is authorized to post and that m is valid
    $Messages := Messages|\langle c, m\rangle$
    send $\langle$**signature**: $c, m, \langle c, m\rangle_{K_t}\rangle$ to p_s

ON MESSAGE $\langle$**signature**: $c, m, signature\rangle$ FROM p_s:
    $Signatures_m^c := Signatures_m^c \cup \{\langle p_s, signature\rangle\}$
    if $|Signatures_m^c| = \lfloor (n-1)/3 \rfloor + 1$ **then**
        send $\langle$**signature**: $m, Signatures_m^c\rangle$ to c

ON MESSAGE $\langle$**read**$\rangle$ FROM c:
    $Read_c = \emptyset$
    for each $p \in SGM.V[SGM.x]$ **do**
        send $\langle$**request-read**: $|Messages|, c\rangle$ to p
    start timer $\langle$**read**: $|Messages|, c\rangle$

ON TIMEOUT $\langle$**read**: $e, c\rangle$
    for each $p \in SGM.V[SGM.x] \backslash Read_c$
        send $\langle$**request-read**: $e, c\rangle$ to p
    start timer $\langle$**remove-read**: $c\rangle$

ON TIMEOUT $\langle$**remove-read**: $c\rangle$
    for each $p \in SGM.V[SGM.x] \backslash Read_c$
        send **down** $\langle$**faulty**: $p\rangle$

ON MESSAGE $\langle$**request-read**: $e, c\rangle$ FROM p_s:
    send $\langle$**reply-read**: $e, c, \langle Messages[1, e]\rangle_{K_t}\rangle$ to p_s

ON MESSAGE $\langle$**reply-read**: $e, c, s\rangle$ FROM p_s:
    $Read_c := Read_c \cup \{\langle p_s, s\rangle\}$
    if $|Read_c| = \lfloor (n-1)/3 \rfloor + 1$ **then**
        send $\langle$**messages**: $Messages[1, e], Read_c\rangle$ to c
        stop timer $\langle$**read**: $e, c\rangle$
        stop timer $\langle$**remove-read**: $c\rangle$

Listing 4.15: Consolidation Protocol

Protocol WP for client c

INITIALLY (m):
    randomly select a party p
    send $\langle$**message**: $m\rangle$ to p
    start timer $\langle$**message**: $m\rangle$

ON TIMEOUT $\langle$**message**: $m\rangle$
    randomly select another party p
    send $\langle$**message**: $m\rangle$ to p
    start timer $\langle$**message**: $m\rangle$

ON MESSAGE $\langle$**signature**: $m, \mathit{Signature}\rangle$ FROM p_s:
    stop timer $\langle$**message**: $m\rangle$
    if $|\mathit{Signatures}| = \lfloor (n-1)/3 \rfloor + 1$ **then**
        randomly select another party p
        send $\langle$**message**: $m\rangle$ to p
        start timer $\langle$**message**: $m\rangle$

Listing 4.16: Write Protocol

Protocol RP for client c

INITIALLY ():
    randomly select a party p
    send $\langle$**read**$\rangle$ to p
    start timer $\langle$**read**$\rangle$

ON TIMEOUT $\langle$**read**: $m\rangle$
    randomly select another party p
    send $\langle$**read**$\rangle$ to p
    start timer $\langle$**read**$\rangle$

ON MESSAGE $\langle$**messages**: $\mathit{Messages}, \mathit{Signature}\rangle$ FROM p_s:
    stop timer $\langle$**read**$\rangle$
    if $|\mathit{Signatures}| = \lfloor (n-1)/3 \rfloor + 1$ **then**
        randomly select another party p
        send $\langle$**read**$\rangle$ to p
        start timer $\langle$**read**$\rangle$

Listing 4.17: Read Protocol

5. Implementation

In this chapter, we present the way we implemented the protocols described in Chapter 4.

5.1. Language and Libraries

We chose to implement our protocols in Java. The following libraries were used:

JavaSE-1.6: The following description has been found on *www.oracle.com*:

> *Java Platform, Standard Edition (Java SE) lets you develop and deploy Java applications on desktops and servers, as well as in today's demanding embedded environments. Java offers the rich user interface, performance, versatility, portability, and security that todays applications require.* Visit *www.oracle.com/javase* for more details.

Spring-2.5.6: the different components of our solution are initialized and configured using the Spring framework. The administrator configures the board (e.g., timer delays, queue sizes, ...) using an XML file. Visit *www.springsource.org* for more details.

Log4j-1.2.16: Log4j is the utility we chose to log relevant events. The administrator is able to configure what to do with the different logs using an XML file. Visit *logging.apache.org/log4j* for more details.

JUnit-4.8.2: we did lots of test using this unit testing framework. Visit *www.junit.org* for more details.

5.2. Layered Architecture

As presented in Figure 5.1, our application has a relaxed layered architecture[1]. In opposition to a strict layered architecture, where a layer n is only allowed to communicate with the layer $n+1$ and $n-1$, the layers communicate here with any other layer.

[1] http://msdn.microsoft.com/en-us/library/ff648623.aspx

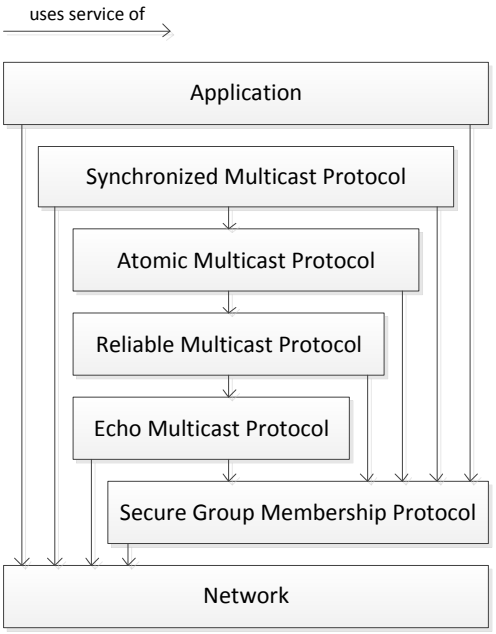

Figure 5.1.: Relaxed layered architecture

5.3. Layer Description

In this section, we focus on a layer n and give details about its components and interfaces.

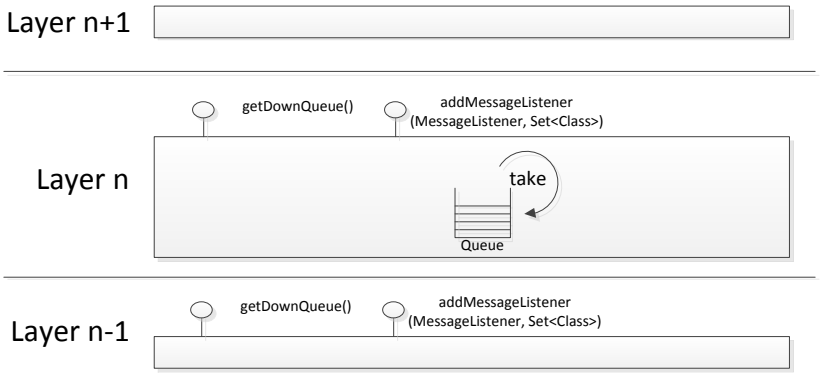

Figure 5.2.: Example of a layer n

As presented in Figure 5.2, each layer possesses the following components:

- one or more queues containing messages to be processed
- an interface used by upper layers to put messages into the queue

- an interface where upper layers can register to receive messages from this layer
- an Activity (more details in Section 5.7) that takes messages from the queue and processes them

Note that a layer possibly contains more than one queue. However, we here present a solution where each layer contains only one queue from which an Activity takes and processes messages. Sending messages to lower layers is done by the Activity, which uses the method *getDownQueue()* of the target layer to put messages into its queue. Reception of messages from lower layers is achieved using the method *addMessageListener*. See Section 5.5 for details about the inter-layer communication.
Every layer runs a protocol, which is *initializable* and *controllable*. More details about the initalization and configuration are provided in Section 5.10.

The *Initializable* interface contains the following methods:

- *init()*: variables requiring information from the other layers (e.g., the index of the view) are initialized here.
- *postInit()*: it is the last initialization step. The other layers and parties are possibly required.

The *Controllable* interface contains the following methods:

- *start()*: starts the protocol. This is done once the protocol has been initialized.
- *terminate()*: terminates the protocol definitely.
- *hold()*: holds the protocol. It can be resumed later.
- *resume()*: resumes an held protocol.
- *reset()*: resets the protocol to its initial state so that it can be started.

Moreover, we choose to use JMX[2] to interact with the layers. Every protocol implements an MBean interface, containing the methods that will be invoked later using a JMX client.

5.3.1. Packages

In order to avoid cyclic dependencies, each protocol is separated in two packages: the service package, containing the interfaces visible to the other protocols and the protocol package, containing the implementation of the protocol.
Additionally, the package *common* contains the classes common to all protocols. Figure 5.3 shows the dependency graph of a layer n and its lower and upper layers. As you can see, a protocol only knows its own service and eventually the services of the lower layers.

[2] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html

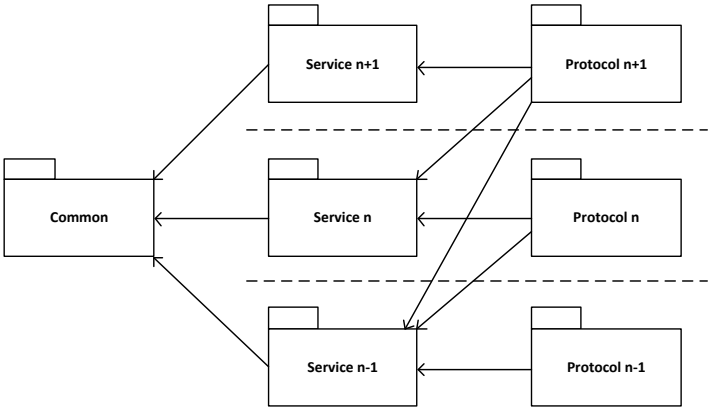

Figure 5.3.: Dependency graph of a layer n

5.3.2. Class Diagram

The class diagram in Figure 5.4 contains the main components present int every layer. Those components are described later in this thesis.

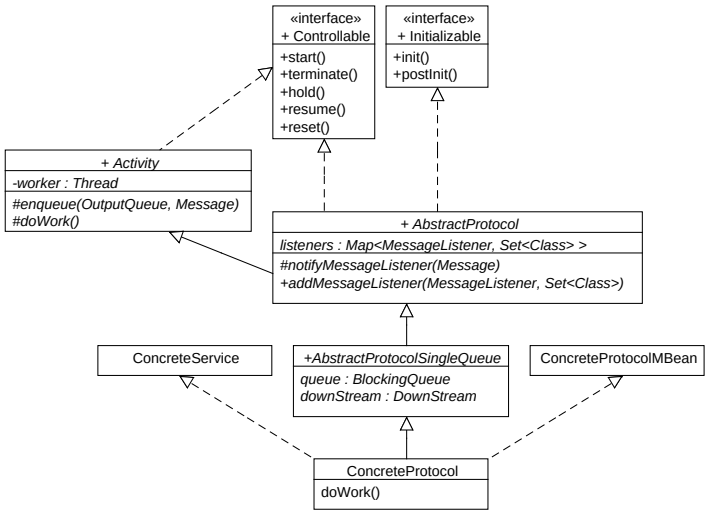

Figure 5.4.: Main components of a protocol

5.4. Messages

The communication between layers is achieved via the exchange of messages. And the communication between peer protocols is also achieved via the exchange of messages (via the service provided by one or more lower layers).

Message and ComposedMessage

The interface *Message* is the super interface of all messages. As presented in Figure 5.5, messages possibly contain other messages. Those are named *ComposedMessages*.

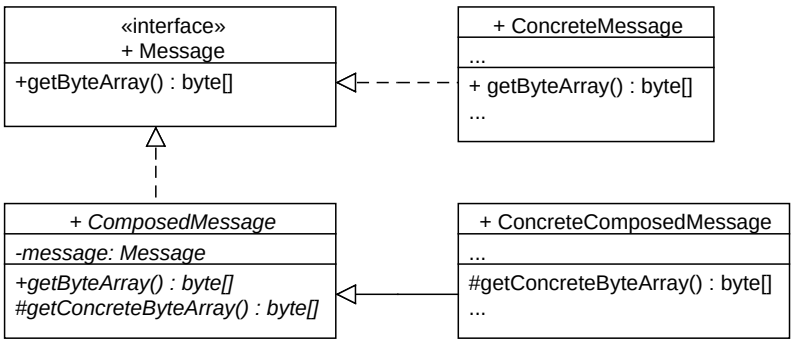

Figure 5.5.: Messages and ComposedMessage

Figures 5.6 and 5.7 represent the data and headers which are added and removed by the different layers.

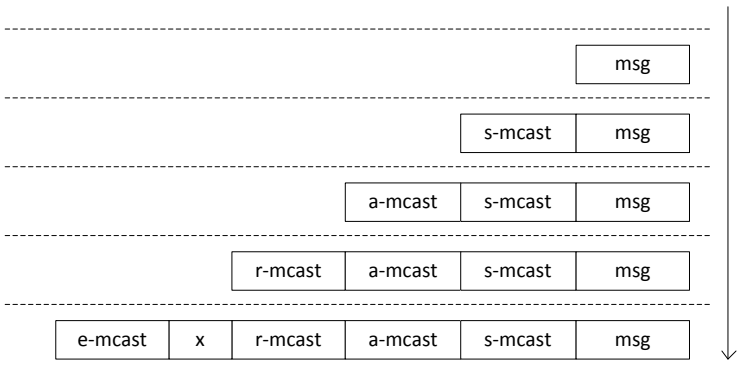

Figure 5.6.: Message sent down

Signatures

When signing a message (see also Section 5.9.2), the content of the signature is given by the method *getByteArray()*. ComposedMessages also use a method *get-*

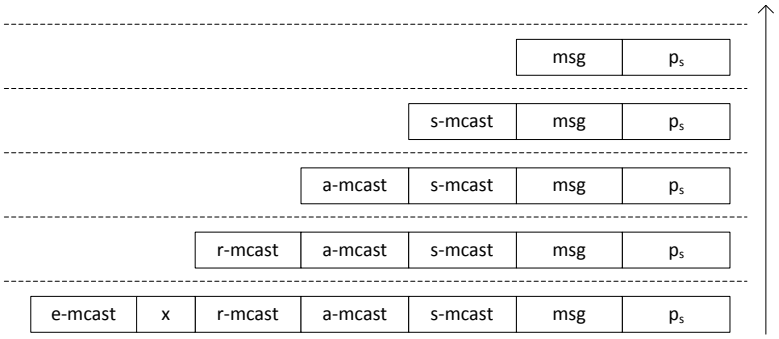

Figure 5.7.: Message sent up

ConcreteByteArray() to deal with the content of the containing and the contained message.

Additionally to the relevant data (e.g., a party), signatures contain the type of the message and the index of the view. These are unique identifiers, necessary to avoid the use of a signature for a different purpose.

5.4.1. Using Collections

We have to be very careful when using collections in the messages. Always keep in mind that the content of the messages will possibly be signed, sent to another party, reconstructed and then verified. If non-sorted collections (e.g., *HashMap* or *HashSet* in Java) are used, the order of their items will possibly not be the same after their reconstruction. Thus, the verification of the message's signature will fail. In order to avoid this problem, use ordered collections or make sure the items have the same order during the creation of a signature and its verification.

The variable p_s (the sender) has to be part of the message. It is verified during the reception of the message. If the p_s contained in the message is different than the p_s obtained using the SSLContext, the message is discarded.

5.5. Communication Between the Layers

The communication between the layers, denoted *send up* and *send down* in the protocols of Chapter 4 is vertical, in opposition to the communication between parties, which is horizontal.

5.5.1. Down-Going Messages

The down-going communication is realized using streams. A layer sending a message down to another one uses the method *enqueue(OutputQueue, Message)* of its Activity. As you can see in Figure 5.8, the interface *DownStream* is an extension of *OutputQueue*. We thus can send a message to a lower protocol using its down stream interface. Moreover, using the method *enqueue* of Activity, the message is

put into the lower protocol's queue in a manner ensuring that it will not be lost if the queue is full or if the protocol is held.

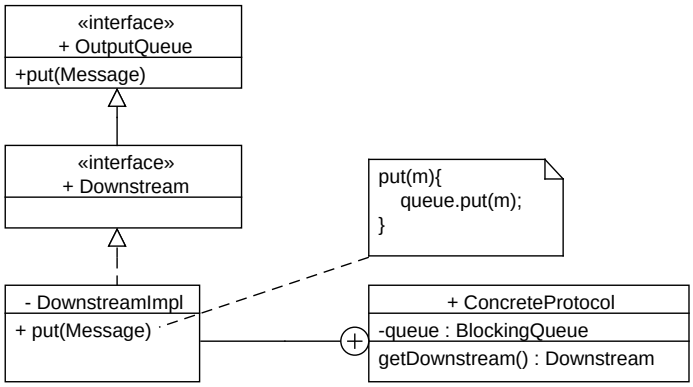

Figure 5.8.: Downstream implementation

In Listing 5.1, an upper layer sends a **faulty** message m down to the Secure Group Membership Protocol.

```
// Get the OutputQueue of the Secure Group Membership Protocol:
OutputQueue queue = sgms.getDownstream();
// Create a FaultyMessage:
Message m = new FaultyMessage(pi, x);
enqueue(queue, message);
```

Listing 5.1: Send a message to a lower layer (send down)

5.5.2. Up-Going Messages

As presented in Figure 5.9, the up-going communication is realized using listeners (the observer pattern). Every layer can subscribe to receive messages from any lower layer by providing a messages filter set. The filter set determines the types of messages for which it wants to receive a notification.

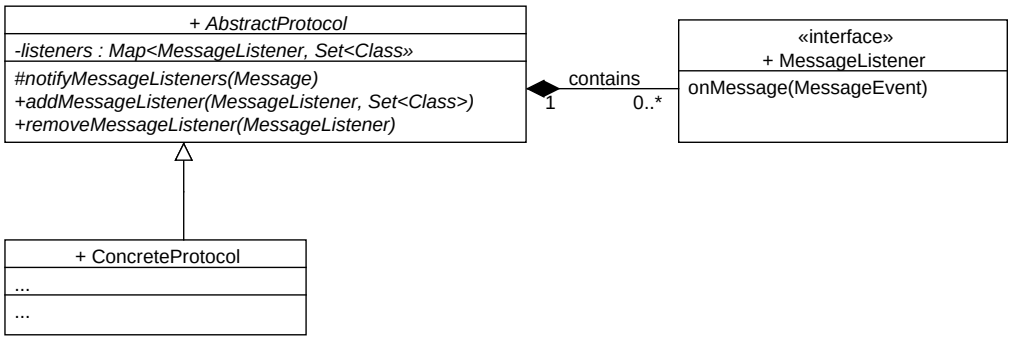

Figure 5.9.: Realization of the observer pattern

In the following listing, an upper layer registers to listen to the view messages coming from the Secure Group Membership Protocol.

```
Set<Class> filterSet = new HashSet<Class>();
filterSet.add(ViewMessage.class);
sgms.addMessageListener(this, filterSet);
```

Listing 5.2: Listen to the messages of a lower layer

Listing 5.3 represents the algorithm used to notify the listeners when a message is to be sent.

```
// In class AbstractProtocol
/**
 * Notify all registered listeners having a corresponding message type in
 * their filter set.
 *
 * @param m  the message to be delivered to the listeners
 * @throws InterruptedException  thrown if worker thread really needs to terminate
 */
protected void notifyMessageListeners(UpMessage m)
        throws InterruptedException {
    MessageEvent e = new MessageEvent(this, m);
    for (MessageListener l : this.listeners.keySet()) {
        for (Class c : this.listeners.get(l)) {
            if (c.isInstance(m.getMessage())) {
                l.onMessage(e);
                break;
            }
        }
    }
}
```

Listing 5.3: Transmissition of a message to a higher layer (send up)

When a message comes from a lower layer (see Listing 5.3), it is put into the addressee's queue and processed later by the Activity. This is represented in Figure 5.10.

5.6. Communication Between the Parties

The horizontal communication (between parties) uses the Network Protocol, responsible for sending and receiving messages to or from other parties.

The Network Protocol

For each party, two links are established: one for sending messages to that particular party and a link end point for receiving messages from it. The link layer manages the links and the link end points. If a link cannot be established, it periodically retries to establish it with the help of an activator. If it succeeds then the link is considered to be up. Links and link end points are tore down whenever the network protocol is put in hold, and are set up again again upon resuming the network protocol. Links and link end points are also tore down whenever the network protocol is terminated.

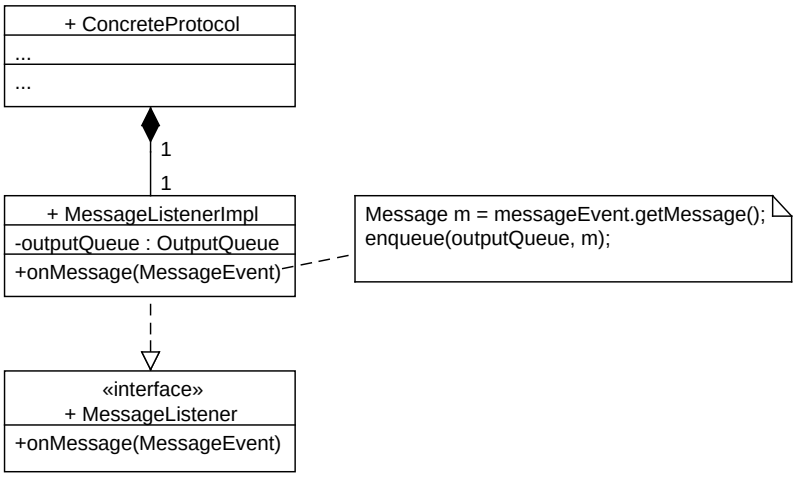

Figure 5.10.: Implementation of a message listener

Network Topology

In order to have direct communication between the parties, we use a *fully connected* network. This leads to a quadratically growing number of connections: $n^2 - n$, where n is the number of parties in the group.

Secure Sockets Layer

We use SSL (Secure Sockets Layer) connections to provide authenticity, confidentiality and integrity.

Each party uses a *SSLServerSocket*, accepting connections from the other parties and possess a *keystore*, containing its own private key and a *truststore*, containing the certificates of parties authorized to send messages.

5.7. Activity and Multi-Threading

Our application is multi-threaded. Each layer possesses its own Activity, which basically is just a worker thread taking messages from a queue and processing them.

In Figure 5.11 you can see the sequence diagram of an Activity taking a message from a queue and processing it. Notice that the queue is in fact a blocking queue. Thus, when the thread of the protocol n wants to take a message from the empty queue, it is blocked until the queue is able to return a message. Similarly, a protocol which puts a message in a full queue will be blocked until there is a free space for the message. We thus can assure that a message will not be lost if a queue is full.

Every protocol must implement a method *doWork()* (abstract in the class Activity), which is used by the worker thread to do the actual work of an Activity. Indeed, the action to perform when a message is taken from the queue differs depending on the protocol and the type of message. This method contains a list of *instanceof*, used to determine the type of the message and call the corresponding process method.

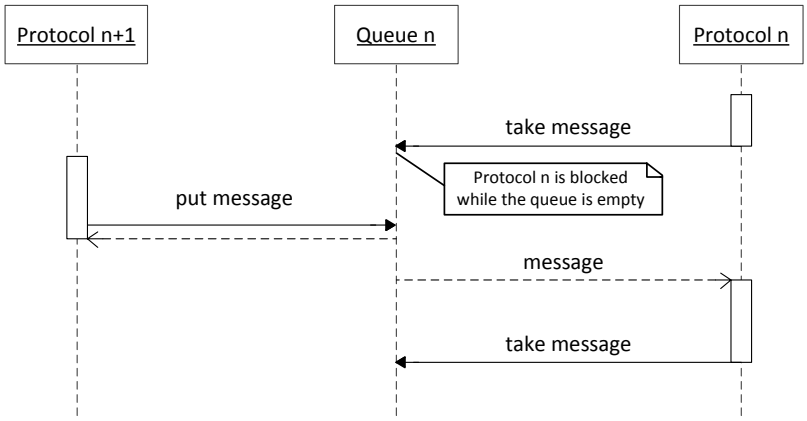

Figure 5.11.: Sequence diagram of an Activity taking a message from a queue

Note that those process methods are public for test purpose but would be private in a production environment.

Additionally, we added the possibility to start, stop, hold and resume an Activity. A layer whose Activity is held, will stop processing messages until its Activity is resumed. Although being useless in production settings of the bulleting board, this is an interesting feature for demonstration purposes. For example, queues can be filled to analyze the subsequent behavior of the protocol.

However, holding a running protocol must be done carefully. Indeed, if a protocol would be held in the middle of a procedure, a part of a procedure could not be executed or a message could be lost. Our strategy to solve this problem is to let the worker thread finish what it was doing and then, hold it. However, if it is waiting on a queue, the thread is directly held.

The following three different states are used in an Activity:

- the ActivityState: describing if the Activity is initialized, running, held, was correctly terminated or if it was forced to terminate for any reason.
- the WorkerState: describing if a worker exists, is running, is blocked on a input/output queue, has correctly terminated or if it was forced to terminate for any reason.
- the Java Thread's state: the thread running in the worker thread. See the Java API[3] for more details.

An Activity also contains instances of timers (periodic or one shot) in order to hold them with the worker thread. Details about the timers are provided in Section 5.8.

5.8. Timers

A distinction is made between two types of timers: *one shot* timers, occurring just once and *periodic* timers, occurring an infinite amount of times.

[3]http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html

Timers are created by the concrete protocol but started and stored by the Activity. This is because when holding an Activity, we also want the timers to be held. However, so that a concrete protocol can cancel the timers, it keeps a reference to them. After a cancellation, the reference is removed from the protocol and the instance from the Activity.

When a timeout occurs, a message corresponding to the timer is put in the protocol's queue. The Activity's worker thread will then process it. Doing so, the exceptions (typically InterruptedException in Java) are safely managed by the Activity. Note that timer messages have a name finishing by "TimerMessage".

5.8.1. One Shot Timers

One shot timers are typically used when a message is expected from another party. If the party does not send the message in time, a procedure is started in order to vote the party out of the group.

The realization of the *remove-unstable* timer, presented in Listing 4.7, is done as follow:

```
Callable<Void> task = new Callable<Void>() {
@Override
public Void call() throws InterruptedException {
    Message rutm = new RemoveUnstableTimerMessage(p, x, l);
    enqueueTimeoutMessage(getTimerOutputQueue(), rutm);
    return null;
}};
ScheduledFuture<?> f = schedule(task, TIMER_DELAY, TimeUnit.MILLISECONDS);
removeUnstableTimers.add(new RemoveUnstableTimer(p, x, l, f));
```

Listing 5.4: Creation of a one shot timer

```
public void processRemoveUnstableTimerMessage(RemoveUnstableTimerMessage rum)
throws InterruptedException {
    int x = rum.getX();
    Party p = rum.getP();
    int l = rum.getL();
    for (Party pi : sgms.getParties(x)) {
            if (l > cp.get(pi).get(p).get(x)) {
            enqueue(sgms.getDownstream(), sgmmf.createFaultyMessage(pi, x));
        }
    }
    stopRemoveUnstableTimer(p, x, l);
}
```

Listing 5.5: Process the message created in the one shot timer above

In Listing 5.4 the timer is created, passed to the Activity using the method *schedule* and a reference is kept in the map *removeUnstableTimers*. If a timeout occurs, a *RemoveUnstableTimerMessage* is put into the queue and processed by the Activity as described in Listing 5.5. Otherwise, the timer is canceled and removed by the protocol.

5.8.2. Periodic Timers

Periodic timers are used when an action has to be performed periodically. It is the case for the *CountersMessages* described in Section 4.6.

The realization of a periodic timer is done as follow:

```
Runnable command = new Runnable() {
    @Override
    public void run() {
        // Put a MulticastMessage in the queue:
        Message em = new MulticastMessage(x, new CountersMessage(x, counters, party));
        enqueueTimeoutMessage(getMyOutputQueue(), em);
    }
};
periodically(command, INITIAL_DELAY, PERIODIC_TIMER_DELAY,
        TimeUnit.MILLISECONDS);
```

Listing 5.6: Creation of a periodic timer

In Listing 5.6, the periodic timer is created and passed to the Activity using the method *periodically*. For now on, a *MulticastMessage* will be periodically added to the queue and processed by Activity worker thread.

5.9. Keys and Algorithms

In this section, we present the algorithms and key lengths we use. Those have been chosen following the recommendations from NIST[4]. With the choices we made, our system is supposed to be secure until 2030.

5.9.1. Hash Functions

As presented in Listing 5.7, we use the SHA-256 hash function.

```
MessageDigest md = MessageDigest.getInstance("SHA-256");
md.update(b, 0, b.length);
byte[] sha256hash = md.digest();
```

Listing 5.7: Hash function

5.9.2. Digital Signatures

RSA signatures are used in the protocols. The exact algorithm is SHA256withRSA found in *java.security.Signature*. The following listing shows how do we generate signatures with the help of class *SignatureServiceImpl*.

```
byte[] toSign = ...;
Signature sig = Signature.getInstance("SHA256withRSA");
sig.initSign(privateKey, new SecureRandom());
sig.update(toSign);
byte[] signature = sig.sign();
```

Listing 5.8: Signature algorithm

The verification of a signature is done similarly, using *sig.initVerify(...)* and *sig.verify(...)* insteads of *sig.initSign(...)* and*sig.sign(...)*.

[4] *www.nist.gov*

5.9.3. Keys

Each party possesses two key pairs. One to sign messages in the protocol and one for the SSL communication. Both are RSA keys of 2048 bits.

5.9.4. Ordering Parties

Parties have to be deterministically ordered, so that the same manager (see Section 4.4) or sequencer (see Section 4.7) can be designated at each party. Additionnally, in the Atomic Multicast Protocol, messages may have to be deterministically delivered. This is done using the rank of the parties: all messages coming from the party with rank 1 are delivered, then those from party with rank 2, and so on until party with rank n.

Parties are ordered on their public keys. The method *String.compareTo(String)* is used to compare two public keys. Doing so, we have the assurance that two parties will never have the same rank (they can not have the same key) and that the order will be the same at every party.

5.10. Configuration and Initialization

5.10.1. Configuration files

The following configuration files are needed at each party:

application-context.xml: used by *Spring* when the application is started, it contains configuration variables and the path to the other configuration files. For more details, visit *www.springsource.org*.

parties-descriptor.xml: describes the parties forming the first view.

sbb-keystore.jks: contains the private key used by p_t to sign messages.

sbb-truststore.jks: contains the public keys of every party in the group.

ssl-keystore.jks: contains the private key used by this party for the SSL communication.

ssl-truststore.jks: contains the public keys of the parties allowed to communicate using SSL.

5.10.2. Spring

The protocol instances (beans) are create by *Spring*, using the *application-context.xml* file. This file contains information to create the beans and configure instance variables (e.g., file paths, interfaces of other protocols, services, etc.).

5.10.3. First View

The first view is installed by an administrator at each party. It is described in the *parties-descriptor.xml* file. An example is given in Listing 5.9, where the first view is composed of three parties and this party is *party1*.

```
<?xml version="1.0" encoding="UTF-8"?>
<description>
    <myAlias>party1</myAlias>
    <parties>
        <party alias="party1">
            <address>
                <ipAddress>127.0.0.1</ipAddress>
                <port>9001</port>
            </address>
        </party>
        <party alias="party2">
            <address>
                <ipAddress>127.0.0.1</ipAddress>
                <port>9002</port>
            </address>
        </party>
        <party alias="party3">
            <address>
                <ipAddress>127.0.0.1</ipAddress>
                <port>9003</port>
            </address>
        </party>
    </parties>
</description>
```

Listing 5.9: Parties forming the first view

5.10.4. Init and Start

Once the protocol instances are created, they have to be initialized. During this phase, they get information from the other protocols (e.g., the index of the view). This is done using the method *init()*, which corresponds to the statement INITIALLY in the pseudocode of Section 4.

We then have to start the protocols (start the Activity's worker thread) and to do their post initialization, during which the network is used to communicate with the other parties (e.g., method *join()* of the Reliable Multicast Protocol). This is done in the same method: *start()*.

The protocols are now ready to work. Note that the two methods *init()* and *start()* can be executed using JMX client.

5.10.5. Template Method Pattern

Because the protocols are different, they possibly have to execute different operations during the start phase. However, the method *start()*, represented in Listing 5.10, is implemented in the class Activity, which is the same for every protocol. We

therefore use the template method pattern as represented in Listing 5.10: first, the method *preStart()* is called, then the worker thread is initialized and started and finally, the method *postStart()* is called. Both *preStart()* and *postStart()* are abstract methods, where operations specific to a concrete protocol are executed.

```
preStart();
setActivityState(ActivityState.RUNNING);
this.t = new Worker();
this.t.start();
postStart();
```

Listing 5.10: Implementation of the method *start()*

Similarly, the template method pattern is used in the method *work()* of the worker thread, so that tasks specific to concrete protocols can be executed by the worker thread before and after it does its work. Typically, the join() procedure of the Reliable Multicast Protocol is executed here. This ensures that exceptions (e.g., full queue) are correctly handled by the Activity.

5.10.6. Initialization Sequence

Here is a summary of the order and the thread who executes the operations during the initialization phase:

constructors:	external thread	initialization of the beans
setters:	external thread	the setters are called to configure the beans
init():	external thread	variables requiring other layers are initialized
preStart():		specific to each protocol
t.start():	external thread	the worker thread is initialized and started
postStart():		specific to each protocol
preWork():		post initialization is done using the network
doWork():	worker thread	the worker thread processes the messages in the queue
postWork():		specific to each protocol

6. Conclusion and Future Work

In this master thesis, we saw that a bulletin board can be used in different contexts to allow users to post messages that will never be removed, modified or moved. Moreover, it must always be available, it possesses no single point of failure and the user is able to prove it if all those properties are not respected.

We also presented a distributed solution running at n parties, of whom less than one third can be corrupt without affecting the correctness of the bulletin board. The user randomly chooses a party, posts his message and receives a receipt for it. This solution, based on the master thesis of R.A. Peters [Pet05], uses the secure broadcast channel described in [Rei94].

Unfortunately, Peters' document is hard to understand and contains errors. A part of the work was to correct them and make this solution easier to understand. Another major difference is that our prototype is multi-threaded. It makes us gain performance and gives us more possibilities to test the system but thread-safe protocols are hard to realize.

Currently, the six lowest protocols (Network, Secure Group Membership, Echo Multicast, Reliable Multicast, Atomic Multicast and Synchronized Multicast) have been implemented and tested. The application layer, described in Section 4.9 remains to be done. However, we realized an simple application (see Appendix A) that gives us the ability run our protocols and to monitor and manage them using a JMX client.

Future work will add group-threshold signatures in the Echo Multicast Protocol, making it more efficient. It will be important to use a scheme not requiring a trusted dealer. Otherwise, a single point of failure will exist.

Additionally, alternative message formats (ASN.1, XML, JSON, etc.) will be used in order to make our bulletin board ready for interoperability with possible other implementations.

A persistence service will also be needed, allowing the board to be stopped and restarted with the same state.

As written earlier, the application layer has to be implemented with a graphical user interface and functionalities. Those functionalities will depend on specific requirements that also are to be defined.

Finally, hash chains, described by J. Heather and D. Lunding [HL09] will be used, giving even more trust in the system. This is to do in the application layer.

A. Simple Application

Insteads of the protocols described in Section 4.9, we implemented a simpler application. Using it, we can test our solution, starting seven parties in different virtual machines (see Figure A.1).

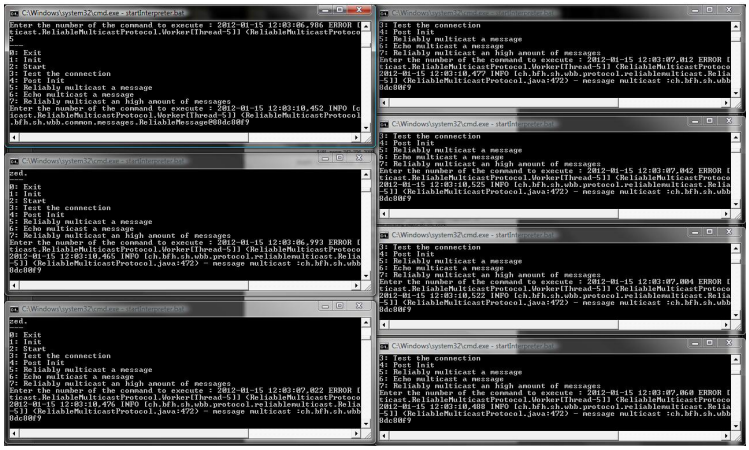

Figure A.1.: The applications running at seven parties

Using JConsole[1] (see Figure A.2) as JMX client we can monitor and manage it. The application initializes the protocoles, starts their worker threads and give the possibility to send one or more messages in a command line interface. The received messages are the printed out.

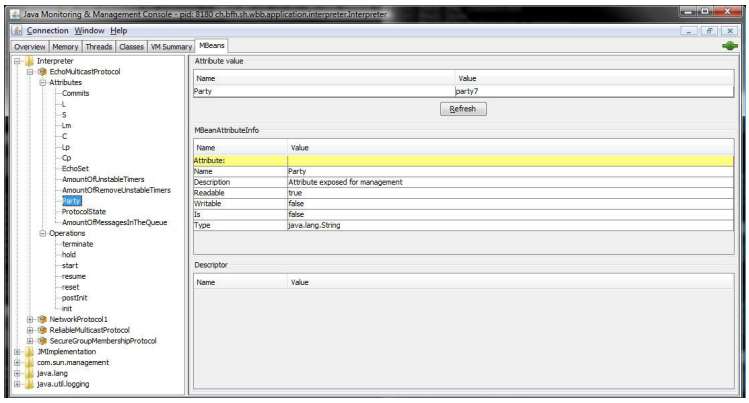

Figure A.2.: The JMX client

[1] http://en.wikipedia.org/wiki/JConsole

Bibliography

[HL09] James Heather and David Lundin. The append-only web bulletin board. *LNCS 5491*, pages 242–256, 2009.

[Kra07] Steven G. Krantz. Zero knowledge proofs. 2007.

[LLP82] Robert Shostak Leslie Lamport and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 682–401, 1982.

[Pet05] Richard A. Peters. A secure bulletin board. Master's thesis, Eindhoven University of Technology, 2005.

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

[Rei96] Michael K. Reiter. A secure group membership protocol. *IEEE Trans. Softw. Eng.*, pages 31–42, 1996.

[Sho00] Victor Shoup. Practical threshold signatures. *Advances in Cryptology EUROCRYPT 2000*, pages 207–220, 2000.