Byzantine Generic Broadcast

Master’s Thesis submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Master of Science in Informatics

presented by
Pavel Raykov

under the supervision of
Prof. Fernando Pedone and Dr. Nicolas Schiper

June 2010
I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Pavel Raykov
Lugano, 20 June 2010
Abstract

Nowadays, Internet services (Google, Facebook, Flickr, Yahoo and etc.) are used by millions of people. Every second these services receive plenty of requests from different visitors. State Machine Replication is a popular approach to ensure a reliability of such systems. It is a highly efficient technique as well, providing availability of such services under a high load. Roughly speaking, State Machine Replication provides clients the illusion of accessing a robust server. Clients’ requests are broadcast among replicas, replicas decide on the execution order of requests, execute them, and then send responses for the requests to clients. Request ordering is needed to ensure that client will receive the same responses from different replicas. In this work, we focus on the broadcast primitives that allow ordering of messages taking into account their semantics.

The standard way to ensure that replicas will receive all broadcast requests in the same order is Atomic Broadcast. Generic Broadcast is a distributed primitive defined in [PS99] as a generalization of Atomic Broadcast. It exploits message semantics to speed up delivery of messages by avoiding their ordering in some cases, as opposed to Atomic Broadcast that orders all messages. In [PS99] the authors proposed a Generic Broadcast algorithm for an asynchronous crash-stop model with the total number of processes n to be at least $3f + 1$, when at most $f$ processes can crash. To the best of the author’s knowledge there is no Generic Broadcast algorithms in an asynchronous Byzantine model. In this work we will present a parameterized algorithm that allows to implement Generic Broadcast in the presence of byzantine faults. In the context of this parameterized algorithm we derive three implementations for an asynchronous Byzantine environment:

- With Interactive Consistency and $n > 4f$.
- With Atomic Broadcast, signatures and $n > 5f$.
- With a Selection Protocol, no signatures, and $n > 9f$. 
In all implementations, non-conflicting messages are delivered in two network delays when all processes are correct.
Acknowledgements

I would like to thank Fernando Pedone for his support and supervision during all two years of the master’s programme. I am grateful to Nicolas Schiper for patiently answering to the heaps of my questions and his help in writing this thesis. I am very thankful to the Faculty of Informatics at USI for giving me a scholarship for studying and living in such a marvelous place as Lugano. It was one in a lifetime experience to be here.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 State Machine Replication</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Atomic Broadcast and Generic Broadcast</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Related work and motivation</td>
<td>2</td>
</tr>
<tr>
<td>2 System model and definitions</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Processes and links</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Message model</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Byzantine Generic Broadcast (BGB) definition</td>
<td>6</td>
</tr>
<tr>
<td>3 Generic Broadcast algorithms overview</td>
<td>7</td>
</tr>
<tr>
<td>3.1 The case when two messages are g-Broadcast</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Case when multiple messages are g-Broadcast</td>
<td>8</td>
</tr>
<tr>
<td>3.3 The Check Protocol specification</td>
<td>8</td>
</tr>
<tr>
<td>4 A parameterized Generic Broadcast (PGB)</td>
<td>11</td>
</tr>
<tr>
<td>4.1 A PGB algorithm</td>
<td>11</td>
</tr>
<tr>
<td>4.2 A proof of correctness</td>
<td>15</td>
</tr>
<tr>
<td>5 BGB with Interactive Consistency and $n &gt; 4f$</td>
<td>21</td>
</tr>
<tr>
<td>5.1 Interactive Consistency definition</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Algorithm $C_{ic}$</td>
<td>22</td>
</tr>
<tr>
<td>5.3 A proof of correctness</td>
<td>23</td>
</tr>
<tr>
<td>6 BGB with Atomic Broadcast, signatures and $n &gt; 5f$</td>
<td>25</td>
</tr>
<tr>
<td>6.1 Algorithm $C_{absign}$</td>
<td>25</td>
</tr>
<tr>
<td>6.2 A proof of correctness</td>
<td>26</td>
</tr>
</tbody>
</table>
6.3 On the usage of signatures ........................................... 29

7 BIB with Selection Protocol and \( n > 9f \) .......................... 31
   7.1 The Selection Protocol specification .............................. 31
   7.2 Algorithm \( \mathcal{C}_{select} \) ....................................... 32
   7.3 A proof of correctness ......................................... 32

8 A deterministic Selection Protocol .................................. 37
   8.1 A Selection Protocol with leader-based consensus ............ 37
   8.2 A Selection Protocol with many consensus instances ......... 38

9 Conclusion .................................................................. 41

A An example of incorrect reasoning for Algorithm 4 in Chapter 7 43

Bibliography .................................................................. 45
## List of Algorithms

1. Parameterized *Generic Broadcast* ($\mathcal{GB}$) ............... 14
2. A *Check Protocol* with the *Interactive Consistency* ($\mathcal{Ci}$) ............... 22
3. A *Check Protocol* with *Atomic Broadcast* and signatures ($\mathcal{Cabsign}$) ............... 26
4. A *Check Protocol* with the *Selection Protocol* ($\mathcal{Cselect}$) ............... 33
5. A *Selection Protocol* with leader-based consensus ............... 38
Chapter 1

Introduction

State Machine Replication [Sch90, Lam84] is a popular approach ensuring fault-tolerance of client-server systems. It is opposed to more simplistic centralized system approaches, where all requests are sent to a single server. In State Machine Replication requests are broadcast to a group of replicas to tolerate server failures. The main difficulty of implementing such a replicated system is to allow clients to send requests to different replicas, while giving them the illusion that they communicate with a single server. Many systems with different properties in various communication models have been proposed so far to implement State Machine Replication. In this work we will focus on a particular distributed primitive Generic Broadcast that can be used to implement State Machine Replication.

1.1 State Machine Replication

We present a general overview of State Machine Replication. We consider the system composed of a set of clients and a set of replicas, where clients broadcast requests to replicas. The execution of client’s requests can be devised in the following steps:

1. Client broadcasts request \( c \) to replicas.

2. Replicas order requests to determine in what order to execute them. Then they process request \( c \).

3. Replicas send responses for the request \( c \) to the client.
Upon receiving enough responses the client decides if its request has been executed successfully or not. In case of a failure, the client could try to execute it again by repeating the whole procedure.

One of the key requirements for such system is a proper ordering of requests for the execution on replicas. It is an essential characteristic that helps to maintain linearizability [HW87]: (1) the execution of commands on different replicas is equivalent to a serial execution on a single replica and (2) if a command $c_1$ is executed before a command $c_2$, then $c_2$ sees the modifications made by $c_1$. If all replicas receive and execute commands in the same order, then linearizability is guaranteed.

Usually a message ordering task is solved by using an Atomic Broadcast [BT85, DSU04] primitive, which ensures that broadcast messages are delivered in the same order. However, Atomic Broadcast can be expensive in terms of one of the key characteristics of replicated system — the number of network delays required to execute one request. In that context was proposed Generic Broadcast, which aimed to exploit requests semantics to avoid ordering all of them, thus being able, in certain cases, to perform significantly better than Atomic Broadcast protocols.

1.2 Atomic Broadcast and Generic Broadcast

Consider the case when replicas execute commands like read and write for the same database entry. We can notice that if all operations are read, we can skip message ordering since this particular entry is never changed. Once a write operation is issued for this entry, we are immediately required to synchronize all replicas in a way that their responses become consistent from the client’s point of view (satisfy linearizability). To generalize this idea we will employ the message conflict relation $\sim$ that defines if two particular messages should be ordered. Generic Broadcast only guarantees that conflicting messages are ordered, while Atomic Broadcast orders all messages. We can see Atomic Broadcast as Generic Broadcast with a full conflict relation $\sim$.

1.3 Related work and motivation

Generic Broadcast has already been implemented in for a crash-stop model [PS99, Zie05]. However, in the last decade the following question was raised — is the crash-stop model reasonable for all practical cases? Leslie lamport [Lam83] and
Barbara Liskov [CL02] state that systems with extremely high reliability require a stronger model, a Byzantine model. The main difference is that in a crash-stop model processes can fail only by crashing, while in a Byzantine model they can start exhibiting an arbitrary behaviour. The main difficulty in switching from one model to another is the complication of the algorithms accompanied by their considerable performance degradation. One notable example of such a transformation are the algorithms that solve the consensus problem [PSL80]. Suppose that the total number of processes is $n$ and up to $f$ processes are faulty. When failures are byzantine, $f$ should always be less than $n/3$ to solve consensus [Lam83], however, there exist algorithms for the crash-stop case that solve consensus with $f < n/2$ [Lam01].

This work provides a parameterized algorithm (PGB) for implementing Generic Broadcast protocols in the presence of byzantine faults. It is a generalization of the GB protocol from [PS99]. We provide three different algorithms in the context of this framework for a Byzantine model in Chapters 5, 6, 7. The new protocols are more clumsy and require more correct processes than the one in [PS99] due to the inherent cost of tolerating byzantine failures. In Chapter 9, we discuss how using new Generic Broadcast protocols we can implement State Machine Replication in a Byzantine model and compare the resulting algorithms to the state of the art.
1.3 Related work and motivation
Chapter 2

System model and definitions

2.1 Processes and links

We consider an asynchronous system composed of \( n \) processes \( \Pi = \{p_1, p_2, \ldots, p_n\} \), which communicate by message passing. From these \( n \) processes, up to \( f \) are byzantine — they can behave arbitrarily. A process that never crashes and is not byzantine is correct. We will say that the adversary, which controls byzantine processes, is computationally bounded (i.e., it cannot break cryptography primitives) and cannot change contents of messages sent by one correct process to another correct process. Moreover, the adversary can select which \( f \) processes to corrupt during the execution of a protocol. The network is fully connected and quasi-reliable: if a correct process \( p \) sends a message \( m \) to a correct process \( q \), then \( q \) receives \( m \). All the links between processes are authenticated, so the recipient of a message \( m \) always knows who the sender of \( m \) is.

Due to the [FLP85] result it is impossible to solve consensus in the system above, so it is impossible to solve problems equivalent to consensus: Atomic Broadcast and Generic Broadcast. The standard way to overcome this impossibility result is to strengthen the model using failure detectors [CT96], synchrony assumptions [DLS88], or randomization [BO83, Tou84]. Instead of strengthening the model explicitly, we assume that we have as “black-boxes” the following distributed primitives: Interactive Consistency [PSL80], Atomic Broadcast [CKPS01], and Selection Protocol [BOEY03]. The resulting model takes into account assumptions we need to implement the corresponding black-boxes. For example, if we use the implementation of Atomic Broadcast based on the Byzantine Agreement in [CL02], we need some timing assumptions and digital signatures; while if we use the implementation of Atomic Broadcast from [CKPS01], we require cryptography primitives.
and the assumption of probabilistic termination of the protocol.

## 2.2 Message model

All messages belong to the predefined set \( M \). We define the following relations on \( M \):

- The conflict relation \( \sim \) on \( M \), s.t. \( \forall m_1, m_2 \in M \), if \( m_1 \sim m_2 \), then \( m_1, m_2 \) need to be ordered. This is a symmetric and non-reflexive relation.

- The relation \( <_{age} \) on \( M \) induced by the \( age \) function that assigns ages to the messages. We will say that \( m_1 <_{age} m_2 \), iff \( age(m_1) < age(m_2) \). This relation is a strict total order relation if \( \forall m_1 \neq m_2 : age(m_1) \neq age(m_2) \), we will see later that this condition holds due to the way the \( age \) function is built.

- The \( \text{Id} \) relation on \( M \) that defines some deterministic way to order any set of messages. Usually it is induced by a \( hash \) function and comparing its outcomes on messages. This relation is a strict total order relation if \( \forall m_1 \neq m_2 : hash(m_1) \neq hash(m_2) \). We will assume that this condition always holds.

### 2.3 Byzantine Generic Broadcast (BGB) definition

Generic Broadcast is defined by the two primitives \( g\text{-Broadcast}(m) \) and \( g\text{-Deliver}(m) \). Here I present an adaptation of the properties of the non-uniform Generic Broadcast algorithm [PS99] to the byzantine failure model:

- **Validity** If a correct process \( p \) \( g\text{-Broadcasts} \) a message \( m \), then \( p \) eventually \( g\text{-Delivers} \) \( m \).

- **Agreement** If a correct process \( p \) \( g\text{-Delivers} \) a message \( m \), then every correct process \( q \) eventually \( g\text{-Delivers} \) \( m \).

- **Integrity** For any message \( m \), every correct process \( p \) \( g\text{-Delivers} \) \( m \) at most once.

- **Order** If correct processes \( p \) and \( q \) both \( g\text{-Deliver} \) conflicting messages \( m \) and \( m' \) (\( m \sim m' \)), then \( p \) and \( q \) \( g\text{-Deliver} \) them in the same order.
Chapter 3

Generic Broadcast algorithms overview

The Generic Broadcast protocol could be seen as a repetition of two phases: an acknowledgement (ACK) phase and a check (CHK) phase. During the ACK phase processes g-Deliver non-conflicting messages. When processes receive conflicting messages they proceed to the CHK phase, decide on the delivery order of conflicting messages, g-Deliver them, and go again to the ACK phase. We first present how Generic Broadcast works when only two messages $m_1$ and $m_2$ are g-Broadcast. We then generalize this execution to the case when multiple messages are g-Broadcast. When explaining the protocol we use two constants $n_{ack}$ and $n_{chk}$ that are discussed later. For simplicity the reader can assume for now that $n_{ack} = n_{chk} = n - f$.

3.1 The case when two messages are g-Broadcast

To g-Broadcast a message $m$ processes send $m$ to all other processes. Now let $m_1$ and $m_2$ be two messages that are g-Broadcast. The way a process $p_i$ handles a received message depends on the messages $p_i$ has received previously, as we explain next (symmetric cases are omitted):

1. $p_i$ has received only $m_1$.
2. $p_i$ has received $m_1$ after $m_2$ and $m_1 \not\sim m_2$.
3. $p_i$ has received $m_1$ after $m_2$ and $m_1 \sim m_2$.

In the first two cases, $p_i$ sends a special protocol message to all processes acknowledging the reception of $m_1$ — hereafter such a message is denoted $\text{ACK}(m_1)$. A
3.2 Case when multiple messages are g-Broadcast

In general, a run of the Generic Broadcast algorithm is decomposed into a sequence of two phases: the ACK phase, where processes deliver non-conflicting messages, and the CHK phase, where processes agree on the delivery order of conflicting messages. These two phases form a stage (we will use the term ‘round’ interchangeably with ‘stage’). Hence, processes progress through a possibly infinite sequence of rounds. In the case when only two messages are g-Broadcast we considered only one stage of the protocol. The transition from the ACK to the CHK phase happens when a process receives two conflicting messages. We show later that if one correct process enters the CHK phase, then all correct processes enter the CHK phase, and together they will successfully terminate it. Processes then reenter the ACK phase in the new stage.

3.3 The Check Protocol specification

The principal scheme of Generic Broadcast algorithms presented in this work and originally proposed in [PS99] is shown in the figure 3.1 (in the case when two conflicting messages are g-Broadcast). In that scheme, the execution is devised in the following steps:

1. Conflicting messages \( m_1 \) and \( m_2 \) are g-Broadcast.

2. Processes \( \{p_1, p_2, p_3, p_4, p_5\} \) receive message \( m_1 \) first and send \( \text{ACK}(m_1) \) to all processes. Process \( p_6 \) receives message \( m_2 \) first and sends \( \text{ACK}(m_2) \) to all processes.
3. Process $p_1$ receives $n_{\text{ack}} \text{ACK}(m_1)$ messages and g-Delivers $m_1$. Afterwards, $p_1$ receives $m_2$, detects the conflict, and sends $R_1 = \{m_1, m_2\}$ to everyone. We call $R_i$ the \texttt{CHK} messages, since they serve as a signal to other processes to launch the \texttt{Check Protocol}.

After sending $R_1$, $p_1$ launches the \texttt{Check Protocol} with the input $\{m_1\}$ — all the messages that $p_1$ has acknowledged so far in this round. We call this input sets of the \texttt{Check Protocol} the pending sets and write them as $\text{pending}_j$ for a process $p_j$.

4. All other processes $p_i$ do not manage to receive enough $\text{ACK}(m_1)$ messages, and firstly receive $m_2$, detect the conflict $m_1 \sim m_2$ and send $R_i = \{m_1, m_2\}$ to everyone. Then $p_i$ launches the \texttt{Check Protocol} with the input $\text{pending}_i$ — all the messages that $p_i$ has acknowledged so far.

5. As an output of the \texttt{Check Protocol} all correct processes have two message sets: $\text{NCSet}$ (a nonconflicting set of messages) and $\text{CSet}$ (a conflicting set of messages). In the figure 3.1, the \texttt{Check Protocol} decides on the $\text{NCSet} = \{m_1\}$ and $\text{CSet} = \{m_2\}$. It is an important property of the \texttt{Check Protocol} that $m_1 \in \text{NCSet}$, since $m_1$ has been g-Delivered by $p_1$ in the $\text{ACK}$ phase. Processes first g-Deliver messages from the $\text{NCSet}$ and then from the $\text{CSet}$, skipping all messages that they have already g-Delivered in the $\text{ACK}$ phase.

![Figure 3.1: One Generic Broadcast protocol stage](image)

- **Figure 3.1:** One \textit{Generic Broadcast} protocol stage. White stars — receive of conflicting messages. Black stars — fast g-Delivery of messages in the $\text{ACK}$ phase. $R_i$ — \texttt{CHK} messages. $\text{pending}_i$ — pending sets of the processes.
Now we can describe the Check Protocol in detail. It is defined by two procedures: proposeCheck\((pending_j)\) and decideCheck\((N\_C\_Set, C\_Set)\). Procedure proposeCheck takes as an input a set of messages \(pending_j\). These \(pending_j\) variables consist of the messages that process \(p_j\) has acknowledged in the current round. Procedure decideCheck returns a pair of sets of messages \((N\_C\_Set, C\_Set)\). The Check Protocol requires that the pending sets of correct processes do not contain conflicting messages; if this condition holds, then the following properties are guaranteed:

- (Agreement and termination) If all correct processes execute proposeCheck\((pending_j)\), then all correct processes eventually decide on the same pair of messages sets \((N\_C\_Set^k, C\_Set^k)\).

- (Validity) If a correct process invokes decideCheck\((N\_C\_Set, C\_Set)\), then:
  
  1. \(N\_C\_Set \cap C\_Set = \emptyset\).
  2. If a message \(m\) belongs to \(n_{ack} - f > 0\) pending sets of correct processes given as an input to the protocol, then \(m \in N\_C\_Set\).
  3. No two messages from \(N\_C\_Set\) conflict.
  4. There exists a \(pending_j\) set of some correct process, s.t \(pending_j \subseteq N\_C\_Set \cup C\_Set\).

We tag instances of the Check Protocol with the stage number \(k\), thus invoking proposeCheck\((k, pending_j)\) and decideCheck\((k, N\_C\_Set, C\_Set)\).

We see that the Check Protocol actually depends on the value of \(n_{ack}\). For example, it is impossible to implement it when \(n_{ack} \leq n/2\), since in that case two non-intersecting subsets of processes can g-Deliver two conflicting messages \(m_1\) and \(m_2\) independently. In that case we will have to put \(m_1\) and \(m_2\) to the \(N\_C\_Set\), which contradicts its specification. We define \(n_{chk}\) as a minimum number of correct processes that the Check Protocol requires, if the total number of processes is \(n\) and \(n_{ack}\) is the parameter used in the ACK phase.
Chapter 4

A parameterized Generic Broadcast (\(\mathcal{PGB}\))

We now present the parameterized Generic Broadcast. It follows the general architecture of Generic Broadcast algorithms presented in Chapter 3 and depends on the Check Protocol. We write an instance of \(\mathcal{PGB}\) combined with the specific Check Protocol \(X\) as \(\mathcal{PGB}(X)\). The algorithm \(GB\) in [PS99] can be seen as an instance of \(\mathcal{PGB}\) with an implementation of the Check Protocol for the crash-stop model. However, not all Generic Broadcast algorithms can be seen as \(\mathcal{PGB}(Y)\) for some Check Protocol \(Y\), it is just a convenient way to present Generic Broadcast.

4.1 A \(\mathcal{PGB}\) algorithm

We now present the \(\mathcal{PGB}\) algorithm (Algorithm 1) and explain its pseudo-code. We assume that we are provided with some Check Protocol and can execute lines 16, 17 according to its definition in Section 3.3. As defined in the Generic Broadcast specification in Section 2.3 we have to provide two functions: g-Broadcast and g-Deliver. To g-Broadcast a message, a process executes line 3, and messages are g-Delivered at lines 18, 19, and 26. The algorithm consists of five concurrent tasks:

- The first entry at line 9 handles g-Broadcast messages in order to correctly receive them.
- The second and third entries at line 9 handle protocol service messages to correctly receive g-Broadcast messages that have not been received directly (as a result of receiving a message sent at line 3 by some process). Specifically, the second entry handles the protocol’s ACK messages, so processes
can receive g-Broadcast message \( m \) not only from \( m \)'s sender, but also from the process that acknowledged \( m \). The third entry handles the protocol’s \( \text{CHK} \) messages, so if one correct process \( p_x \) sees conflicting messages, then eventually all correct processes receive a \( \text{CHK} \) protocol message from \( p_x \) with conflicting messages.

These two entries guarantee liveness in the presence of the byzantine senders. They serve as Reliable Broadcast [MR97] routines for message retransmission.

- The block starting at line 10 handles sending \( \text{ACK} \) messages and executing the \( \text{CHK} \) phase.
- The block starting at line 23 handles the g-Delivery of messages that received enough \( \text{ACK} \) messages.

The following variables are used by the algorithm in round \( k \):

- \( k \) (line 2) defines the current round number.
- \( \text{Received} \) (line 2) contains all the g-Broadcast messages that the process has received so far.
- \( G \_\text{delivered} \) (line 2) contains all the messages that have been g-Delivered by the process in the rounds \( k' < k \).
- \( \text{pending}^k \) (line 2) defines the set of non-conflicting messages acknowledged by the process so far. Serves as an input to the \( \text{CHK} \) protocol at line 16.
- \( \text{Ack}^k \) (line 2) defines the set of messages g-Delivered in the \( \text{ACK} \) phase in round \( k \) by the process up to the current moment.
- \( \text{age} \) (line 2) defines the ordinal number of the last message put in the \( \text{Received} \) set. It is used to maintain the \( \text{age} \) function that assigns all messages their ordinal number according to their receive order.
- \( N\text{CSet}^k \) (line 17) is the output of the \textit{Check Protocol} in round \( k \) containing non-conflicting messages.
- \( C\text{Set}^k \) (line 17) is the output of the \textit{Check Protocol} in round \( k \) containing possibly conflicting messages.

The algorithm uses one auxiliary function \textit{oldest-unclicteded}. It has the following specification:
4.1 A PGB algorithm

- As input it takes an ordered set \((S, <_{\text{age}})\) of messages. For two messages \(m_1, m_2\) the relation \(<_{\text{age}}\) defines what message was received first. When \(m_1\) was received before \(m_2\), we say that \(m_1\) is older than \(m_2\).

- As output it returns a message set \(Out\), s.t. (1) no two messages in \(Out\) conflict, (2) \(Out\) contains the oldest message in \(S\), (3) if no messages in \(S\) conflict, then \(Out = S\), otherwise \(Out \subset S\) and (4) the ‘stability’ property holds:

\[
\forall S, S', s.t. S \subseteq S' \text{ and } \forall m \in S \forall m' \in (S' \setminus S) \quad m <_{\text{age}} m' \quad \implies \quad \text{oldest-unconflicted}(S, <_{\text{age}}) \subseteq \text{oldest-unconflicted}(S', <_{\text{age}})
\]

We now take a closer look at how the algorithm works. We consider the execution of one stage of the algorithm. Let some correct processes \(p_i\) and \(p_j\) be in the \(\text{ACK}\) phase of round \(k\). When \(p_i\) wants to g-Broadcast a message \(m\), \(p_i\) executes line 4. After \(p_j\) receives \(m\), \(m\) is put in the \(\text{Received}\) set (first entry at line 9). Process \(p_j\) then eventually evaluates line 10 finding out that there is a new message to be handled. Afterwards \(p_j\) tries to put \(m\) into the pending set by rebuilding it using the function \(\text{oldest-unconflicted}\). This function takes as input a set of not yet g-Delivered messages \(\text{Received} \setminus G\_\text{delivered}\) and returns a new pending set. There are two cases to consider here:

- **Non-conflicting messages.** In that case, \(m\) does not conflict with any other messages in \(\text{Received} \setminus G\_\text{delivered}\), so the condition at line 12 is true. Process \(p_j\) now sends \(\text{ACK}(m)\) message containing the whole \(\text{pending}^k\) variable at line 13. When \(p_j\) receives \(m\), \(m\) acknowledges with \(m\), the condition at lines 23, 24 becomes true. Later, \(p_j\) just puts \(m\) in the \(\text{Ack}^k\) set at line 25 and g-Delivers \(m\) at line 26.

- **Conflicting messages.** In that case, \(\text{pending}^k \neq \text{Received} \setminus G\_\text{delivered}\), which means that \(m\) conflicts with a message in \(\text{pending}^k\). Process \(p_j\) enters the \(\text{CHK}\) phase and sends \(\text{Received} \setminus G\_\text{delivered}\) to all other processes to guarantee that all correct processes will see conflicting messages too and will enter the \(\text{CHK}\) phase. Then, at lines 16 and 17 process \(p_j\) participates in the \(\text{Check Protocol}\) and decides on the pair \((\text{NCSet}^k, \text{CSet}^k)\). At line 18 \(p_j\) delivers messages that could have been g-Delivered before in the \(\text{ACK}\) phase \((\text{NCSet}^k)\). At line 19 \(p_j\) delivers all other messages that have been decided in the \(\text{Check Protocol}\) \((\text{CSet}^k)\). Then at line 20 g-Delivered messages from \(\text{NCSet}^k\) and \(\text{CSet}^k\) are put in the \(G\_\text{delivered}\) set in order not to be g-Delivered a second time. At line 21, process \(p_j\) finishes stage \(k\) and its \(\text{CHK}\)
phase by increasing the round counter and initializing $\text{pending}^k$ and $\text{Ack}^k$ of the new round to empty sets.

**Algorithm 1:** Parameterized Generic Broadcast ($\mathcal{PGB}$)

1. **Initialization:**
   
   - $\text{Received} \leftarrow \emptyset$, $\text{G}_{\text{delivered}} \leftarrow \emptyset$, $\text{pending}^1 \leftarrow \emptyset$, $k \leftarrow 1$, $\text{age} \leftarrow 0$

2. To execute $g$-Broadcast($m$):
   
   - send($m$) to all

3. To execute Handle($S$):
   
   - for each $m \in S \setminus \text{Received}$ do $\text{age}(m) \leftarrow \text{age}++$
   - $\text{Received} \leftarrow \text{Received} \cup S$

4. $g$-Deliver($m$) occurs as follows:

   - when receive($m$) do $\text{Handle}([m])$
   - when receive($k$, $\text{pending}^k_j$, $\text{ACK}^k_j$) do $\text{Handle}([\text{pending}^k_j])$
   - when $(\text{Received} \setminus (\text{G}_{\text{delivered}} \cup \text{pending}^k)) \neq \emptyset$ do
     
     - $\text{pending}^k \leftarrow \text{oldest-unconflicted}((\text{Received} \setminus \text{G}_{\text{delivered}}), \prec_{\text{age}})$
     - if ($\text{pending}^k = \text{Received} \setminus \text{G}_{\text{delivered}}$) then
       
       - send($k$, $\text{pending}^k$, $\text{ACK}^k$) to all
       
     - else
       
       - send($k$, $(\text{Received} \setminus \text{G}_{\text{delivered}})$, $\text{CHK}$) to all
       
       - $\text{proposeCheck}(\text{pending}^k)$
       
       - $\text{wait until} \text{decideCheck}(\text{NCSet}^k, \text{CSet}^k)$
       
       - for each $m \in \text{NCSet}^k \setminus (\text{G}_{\text{delivered}} \cup \text{Ack}^k)$ do $g$-Deliver($m$)
       
       - $\text{in ID order: for each } m \in \text{CSet}^k \setminus (\text{G}_{\text{delivered}} \cup \text{Ack}^k) \text{ do } g$-Deliver($m$)
       
       - $\text{G}_{\text{delivered}} \leftarrow \text{G}_{\text{delivered}} \cup \text{NCSet}^k \cup \text{CSet}^k$
       
       - $k \leftarrow k + 1$, $\text{pending}^k \leftarrow \emptyset$, $\text{Ack}^k \leftarrow \emptyset$
   

5. $\exists m : [ \text{for } n_{\text{ack}} \text{ processes } p_j : \text{received } (k, \text{pending}^k_j, \text{ACK}) \text{ from } p_j \text{ and }$

   - $m \in (\text{pending}^k_j \setminus \text{Ack}^k) \cap \text{pending}^k ]$ do

   - $\text{Ack}^k \leftarrow \text{Ack}^k \cup \{m\}$

   - $g$-Deliver($m$)

6. To execute oldest-unconflicted($S$, $\prec_{\text{age}}$):

   - $\text{result} \leftarrow \emptyset$

   - in $\prec_{\text{age}} \text{ order: for each } m \in S$ do

     - if ($\forall m' \in \text{result} : m \neq m'$) then

       - $\text{result} \leftarrow \text{result} \cup \{m\}$

   - end

7. return result
4.2 A proof of correctness

In all the proofs we use notation $G_{\text{delivered}}_x$, $\text{Received}_x$, $\text{Ack}_x^k$ to reference the corresponding sets of process $p_x$. All processes progress by moving from round to round ($k = 1, 2, \ldots$) or by staying in the same round forever. After executing the initialization at line 2 all correct processes enter to the first round. The transition from round $k$ to $k+1$ occurs after execution of the first atomic block (lines 18-21). All the lines and blocks marked atomically are executed atomically.

**Lemma 4.1.** (1) If one correct process enters round $k$, then all correct processes enter round $k$; (2) at the beginning of every round $k$ the contents of $G_{\text{delivered}}$ sets are the same for all correct processes, and $G_{\text{delivered}}$ set is not modified during the round $k$.

**Proof.** We will proof this lemma by induction on round number $k$.

(Base $k=1$) Initially all processes enter the first round. In the beginning of round 1 all processes have the same $G_{\text{delivered}}$ set equal to $\emptyset$. $G_{\text{delivered}}$ is not changed until the first atomic block is executed and processes move to the next round. So, during the first round the $G_{\text{delivered}}$ set is the same for all correct processes.

(Induction step $k \rightarrow k + 1$) The contents of $G_{\text{delivered}}$ set are changed only in the first atomic block at line 20. So, $G_{\text{delivered}}$ sets are not modified during round $k + 1$ — they are changed only when processes enter next round executing first atomic block. To prove the induction step we just need to show two statements:

1. **If one correct process enters round $k + 1$, then all correct processes enter round $k + 1$.** To show this we prove that all correct processes enter the $\text{chk}$ phase and terminate it in round $k$, if one correct enters it. Suppose one correct process $p_x$ entered the $\text{chk}$ phase in round $k$. It means that (1) there are two conflicting messages $m, m' \in (\text{Received}_x \setminus G_{\text{delivered}}_x)$ and (2) it executed line 15 sending $\text{Received}_x \setminus G_{\text{delivered}}_x$ to everyone. Now suppose that there is a correct process $p_y$ that never enters the $\text{chk}$ phase in round $k$. We know that eventually it will receive the message $(k, (\text{Received}_x \setminus G_{\text{delivered}}_x), \text{chk})$ from $p_x$. So, it will handle it in the third entry at line 9, thus executing line 7. So, we know that $m, m' \in \text{Received}_y$ now. By assumption $G_{\text{delivered}}_x = G_{\text{delivered}}_y$ during the round $k$, consequently, $m, m' \in \text{Received}_y \setminus G_{\text{delivered}}_y$ when the check is made by $p_y$ at line 12 and if-condition is false. So, process $p_y$ successfully enters
the CHK phase. In that manner, all correct processes enter the CHK phase and terminate the Check Protocol, a contradiction.

2. **At the beginning of every round** $k + 1$ **the contents of** $G_{\text{delivered}}$ **sets are the same for all correct processes.** In the previous step we have shown that all correct processes participate in the Check Protocol. Since all correct processes decide on the same pair $(\text{NCSet}^k, \text{CSet}^k)$ (due to the specification of the CHK protocol in section 3.3) and they have the same $G_{\text{delivered}}$ sets before the line 20, then after executing the line 20 all correct processes will get the same set $G_{\text{delivered}}$.

\[\square\]

**Lemma 4.2.** All correct processes g-Deliver the same set of messages in each round $k$. This set is $\text{NCSet}_k^k \cup \text{CSet}_k^k$, if a correct process executes line 20 in the round $k$.

**Proof.** There are two cases to consider here:

- **No correct process in round** $k$ **entered the** CHK **phase.** Thus, it means that processes g-Delivered messages only by acquiring enough ACK messages at line 26 of Algorithm 1. So, suppose one correct process $p_x$ receives $n_{\text{ack}}$ pending sets with $m$. Assume $n_{\text{ack}} > f$, then there is some correct process $p_y$ that received $m$ and acknowledged it in the round $k$. Hence, $m \in \text{Received}_y \setminus G_{\text{delivered}}_y$ and since all $G_{\text{delivered}}$ sets are the same at the beginning of the round $k$, then $m$ does not belong to the $G_{\text{delivered}}$ set in round $k$. So, all correct processes will eventually receive it, acknowledge it, and then g-Deliver it at line 26.

- **Some correct process in this round entered the** CHK **phase.** By lemma 4.1 we know that if one correct enters the CHK phase, then all correct processes will enter the CHK phase in this round. Firstly, we will show that:

$$\forall \text{ correct process } p_x \text{ Ack}^k_x \subseteq \text{NCSet}^k_x$$

Consider any $m \in \text{Ack}^k_x$. We know that $p_x$ received $n_{\text{ack}}$ acknowledgments for message $m$. So, there are at least $n_{\text{ack}} - f$ correct processes $p_j$ with $m \in \text{pending}^k_j$. Using the ‘stability’ property of oldest-unconflicted function, we conclude that if a correct process $p_x$ acknowledges the message $m$ at line 13 and enters CHK phase at line 16, then the input to the CHK protocol $\text{pending}^k_x$ at line 16 contains $m$. Hence, there are at least $n_{\text{ack}} - f$ inputs with $m$ from correct processes to the CHK protocol. By the specification of
the \( \text{CHK} \) protocol, \( m \) appears in \( \text{NCSet}^k \). So, we have shown that \( \forall m \in \text{Ack}_x^k \implies m \in \text{NCSet}^k \), so \( \text{Ack}_x^k \subseteq \text{NCSet}^k \). Secondly, we know that all correct processes will execute lines 18 and 19, consequently, all messages from \( \text{NCSet}^k \cup \text{CSet}^k \) will be g-Delivered in the \( \text{CHK} \) phase of the round \( k \) (if not have been g-Delivered before in the \( \text{ACK} \) phase of the round \( k \)).

\( \square \)

**Proposition 4.1.** (Validity) *If a correct process \( p \) g-Broadcasts a message \( m \), then \( p \) eventually g-Delivers \( m \).*

**Proof.** Suppose \( m \) is never g-Delivered by \( p \). Since the links are quasi-reliable, we conclude that all correct processes will receive \( m \) eventually. Now let’s designate a round in which \( m \) was g-Broadcast as \( k \). The fact that \( m \) was never g-Delivered by \( p \) in \( k \) means that:

1. Processes never entered the \( \text{CHK} \) phase, but did not acknowledge \( p \) with pending sets with \( m \), or
2. Processes entered the \( \text{CHK} \) phase and did not decide on the \( (\text{NCSet}^k, \text{CSet}^k) \) s.t. \( m \in \text{NCSet}^k \cup \text{CSet}^k \). Then they proceeded to the next round.

Consider the first case: since all correct processes receive \( m \) in this round, they should have put \( m \) in their pending sets, otherwise, there was already some conflicting message there and they should have entered the \( \text{CHK} \) phase. So, all correct processes should acknowledge \( m \), and thus it will be g-Delivered in the \( \text{ACK} \) phase, a contradiction.

Now consider the second case: suppose all processes progress round by round and \( m \) never gets g-Delivered. Eventually all correct processes receive \( m \), and since it never gets g-Delivered in the \( \text{CHK} \) phase, then there is always some correct process \( p_x \) that never puts \( m \) in its pending set in all rounds \( l > k \) at line 11. Otherwise, if all correct processes put \( m \) in their pending sets, then, due to the specification of the \( \text{CHK} \) protocol, \( m \) will appear in \( \text{NCSet}^k \cup \text{CSet}^k \), and hence it will be g-Delivered. If process \( p_x \) never puts \( m \) in its pending set, then \( p_x \) always has some message \( m'_l \) in its \((\text{Received} \setminus \text{G\_delivered}) \) set, s.t. \( m \sim m'_l \) and \( \text{age}(m'_l) < \text{age}(m) \). So, there are two cases:

- There is an infinite set of different messages \( \{m'_l\}_{l=1}^{\infty} \), s.t. \( \text{age}(m'_l) < \text{age}(m) \), which contradicts the way messages are assigned an age.
• There is a message $m'$ that is always in $(\text{Received}_x \setminus G_{\text{delivered}}_x)$ and is never g-Delivered as $m$. In this case we can repeat all our reasoning for $m'$ having in mind that $\text{age}(m') < \text{age}(m)$. But this still leads us to the contradiction, since we can build an infinitely big chain of messages with decreasing age that are never g-Delivered.

We get that both cases are impossible, so we get a contradiction with the main assumption. □

**Proposition 4.2.** (Agreement) If a correct process $p$ g-Delivers a message $m$, then every correct process $q$ eventually g-Delivers $m$.

**Proof.** Follows immediately from lemma 4.2, since $m$ is g-Delivered in the same round by all correct processes. □

**Proposition 4.3.** (Integrity) For any message $m$, every correct process $p$ g-Delivers $m$ at most once.

**Proof.** Suppose some message $m$ was g-Delivered twice. It could be done in the following ways:

• **Message $m$ was g-Delivered twice in the ACK phase at line 26 in the same round.** In that case after the first g-Delivery it was put in $\text{Ack}^k$ set, so it couldn’t be g-Delivered later because condition at line 23 checks that $m \notin \text{Ack}^k$.

• **Message $m$ was g-Delivered twice in the ACK phase at line 26 in different rounds.** Suppose it was g-Delivered in rounds $k$ and $l$ with $k < l$. So $m \in \text{Ack}^k$ and due to the second item in lemma 4.2 $m \in G_{\text{delivered}}$ starting from round $k + 1$. The fact that $m \in G_{\text{delivered}}$ makes it impossible for $m$ to be put in the pending set in rounds after the round $k$, since it is built from the set $\text{Received} \setminus G_{\text{delivered}}$. If $m$ is g-Delivered in round $l$ at line 26, then condition at line 24 holds, so $m \in \text{pending}^l$, a contradiction.

• **Message $m$ was g-Delivered twice in the CHK phase in the same round.** This is impossible since $\text{NCS} \cap \text{CSet} = \emptyset$.

• **Message $m$ was g-Delivered twice in the CHK phase in different rounds.** This is impossible since after first g-Delivery of $m$ it is put in $G_{\text{delivered}}$ set at line 20, and later we g-Deliver messages that do not belong to the $G_{\text{delivered}}$ set.
• Message $m$ was g-Delivered in the $\text{ACK}$ phase, then in the $\text{CHK}$ phase in the same round $k$. If $m$ was g-Delivered in the $\text{ACK}$ phase, then it belongs to $\text{Ack}^k$ set. So, this case is impossible since we g-Deliver only messages that do not belong to the $\text{Ack}^k$ set in the $\text{CHK}$ phase.

• Message $m$ was g-Delivered in the $\text{ACK}$ phase in the round $k$, then in the $\text{CHK}$ phase in the round $l$, $k < l$. If $m$ was g-Delivered in the $\text{ACK}$ phase in the round $k$, then by lemma 4.2 $m$ appears in $G_{\text{delivered}}$ set in the $\text{CHK}$ phase in the round $l$. So, this case is impossible since we g-Deliver only messages that do not belong to the $G_{\text{delivered}}$ set in the $\text{CHK}$ phase.

• Message $m$ was g-Delivered in the $\text{CHK}$ phase in the round $l$, then in the $\text{ACK}$ phase in the round $k$, $k > l$. If $m$ was g-Delivered in the $\text{CHK}$ phase in the round $l$, then $m$ appears in the $G_{\text{delivered}}$ set in the round $k$ due to the lemma 4.2. If $m$ was g-Delivered in the $\text{ACK}$ phase, then it appeared in the set $\text{pending}^k$ due to the condition at the line 24. So, $\text{pending}^k \subseteq \text{Received} \setminus G_{\text{delivered}}$ (by the specification of the $\text{oldest-unconflicted function}$) and $m \in G_{\text{delivered}}$, a contradiction.

□

**Proposition 4.4.** (Order) If correct processes $p$ and $q$ both g-Deliver conflicting messages $m$ and $m'$ ($m \sim m'$), then $p$ and $q$ g-Deliver them in the same order.

In the following proof we assume that $2n_{\text{ack}} - n > f$.

**Proof.** Consider four cases:

• **If $p$ and $q$ g-Deliver $m$ and $m'$ in different rounds**, then the proposition is ensured by lemma 4.2 and proposition 4.3.

• **If $p$ and $q$ g-Deliver $m$ and $m'$ in the $\text{CHK}$ phase**, it means that $m$ and $m'$ have not been g-Delivered by $p$ and $q$ in the $\text{ACK}$ phase. Consequently, either $m$ and $m'$ will be g-Delivered by $p$ and $q$ in the id order at the line 19, or since $m \sim m'$, one of this messages will be g-Delivered at the line 18 and another will be g-Delivered at the line 19.

• **If $p$ and $q$ g-Deliver $m$ and $m'$ in the $\text{ACK}$ phase**, then if an intersection of two quorums of $n_{\text{ack}}$ processes has at least one correct process ($2n_{\text{ack}} - n > f$), then these messages do not conflict, so it is impossible that $p$ g-Delivers $m$ and $q$ g-Delivers $m'$ in the $\text{ACK}$ phase.
• If \( p \) g-Delivers \( m \) in the \textit{ACK} phase, and \( q \) g-Delivers \( m' \) in the \textit{CHK} phase.

Using the second item of the lemma 4.2, we see that \( m \) will appear in the \( NCS{\scriptsize et}^k \) variable decided by all correct processes in the \( \text{CHK} \) protocol. Since \( m \sim m' \), by specification of the \textit{Check Protocol}:

\[
m' \notin NCS{\scriptsize et}
\]

If \( m' \) was g-Delivered, then \( m' \) appears in \( C{\scriptsize set} \). So, all processes will g-Deliver \( m \) in the \textit{ACK} phase or from the \( NCS{\scriptsize et} \) in the \textit{CHK} phase, and will g-Deliver \( m' \) from the \( C{\scriptsize et} \) set.

\[\square\]

\textbf{Theorem 4.1.} \( \mathcal{PGB} \) requires that \( 2n_{\text{ack}} - n > f \) (condition for the right order of conflicting messages) and a valid \textit{Check Protocol}.

\textbf{Proof.} The theorem holds from Propositions 4.1, 4.2, 4.3 and 4.4. \[\square\]
Chapter 5

BGB with Interactive Consistency and $n > 4f$

In this chapter, we provide an implementation of the Check Protocol and denote it as $\mathcal{C}_{ic}$ (Algorithm 2). We prove that $\mathcal{C}_{ic}$ satisfies the Check Protocol specification (for specification see Section 3.3). In order to get a complete algorithm for Byzantine Generic Broadcast, we just need to plug $\mathcal{C}_{ic}$ in $\mathcal{PGB}$. $\mathcal{C}_{ic}$ uses Interactive Consistency [PSL80, BOEY03].

5.1 Interactive Consistency definition

Roughly speaking, Interactive Consistency is a consensus protocol that decides on the subset of the proposed values. Suppose that, like in consensus, each process $p_i$ has some value $v_i$. The Interactive Consistency abstraction provides two functions: proposeIC($v_i$) and decideIC($\vec{V}$). To execute the protocol processes invoke function proposeIC($v_i$) and then wait for a callback decideIC($\vec{V}$), where $\vec{V}$ is the vector of $n$ values. The protocol guarantees the following properties:

- (Termination) All correct processes eventually decide on some value $\vec{V}$.
- (Agreement) No two correct processes decide differently.
- (Validity) If a correct process decides on a value $\vec{V}$, then for all correct processes $p_i$ and their proposed values $v_i : \vec{V}[i] = v_i$.

We will tag the instance of Interactive Consistency algorithm with the round number $k$, thus using modified functions: proposeIC($k, v_i$) and decideIC($k, \vec{V}$).
5.2 Algorithm \( C_{ic} \)

Before presenting the algorithm we define one concept:

**Definition 5.1.** pending

\[ k \] is said to be correct iff:

\[ \forall m_1, m_2 \in \text{pending}^k_j : m_1 \not= m_2 \]

Algorithm 2 works as follows:

1. Correct processes acquire a global state of the system (the \( GS^k \) variable) using Interactive Consistency at lines 2, 3. Interactive Consistency guarantees that all correct processes put their pending sets in \( GS^k \).

2. We filter \( GS^k \) in a way that it contains only correct pending sets at line 4. This is essential for the next step at line 5, so we do not put conflicting messages in \( NCSet^k \) (this could happen if some byzantine process proposed pending set with conflicting messages at line 2).

3. We put in the \( CSet^k \) variable all the messages from \( GS^k \) that we have not put in \( NCSet^k \). Hence, we guarantee the property Validity-4 of the Check Protocol.

4. At line 5 we put in \( NCSet^k \) messages that belong to a majority of the pending sets of \( GS^k \). We do this to ensure the property Validity-2 of the Check Protocol. In this case, we need that if a message belongs to the \( n_{\text{ack}} - f \) correct pending sets, then it belongs to a majority of the pending sets of \( GS^k \).

**Algorithm 2:** A Check Protocol with the Interactive Consistency (\( C_{ic} \))

```
1 Procedure proposeCheck(k, pending^k)
2 \quad proposeIC(k, pending^k)
3 \quad wait until decideIC(k, V)
4 \quad GS^k \leftarrow \text{select correct pending}_j^k \text{ from } V
5 \quad NCSet^k \leftarrow \{ m \mid \exists \lceil \frac{|GS^k|+1}{2} \rceil \text{ sets pending}_j^k \in GS^k : m \in \text{pending}_j^k \}
6 \quad CSet^k \leftarrow \bigcup_{pending_j^k \in GS^k} \text{pending}_j^k \setminus NCSet^k
7 \quad decideCheck(k, NCSet^k, CSet^k)
```
5.3 A proof of correctness

**Proposition 5.1. (Agreement and termination)** If all correct processes execute `proposeCheck(pending_j)`, then all correct processes eventually decide on the same pair of message sets \((\text{NCSet}^k, \text{CSet}^k)\).

*Proof.* Termination holds as long as Interactive Consistency algorithm terminates. Interactive Consistency decides on the same vector of messages \(\vec{V}\). Since \(\text{GS}^k\), \(\text{NCSet}^k\) and \(\text{CSet}^k\) are just deterministic functions of \(\vec{V}\), all the correct processes will decide on the same pair of message sets \((\text{NCSet}^k, \text{CSet}^k)\). 

**Proposition 5.2. (Validity-1)** \(\text{NCSet}^k \cap \text{CSet}^k = \emptyset\).

*Proof.* Follows immediately from the definition of \(\text{CSet}^k\) at line 6. 

To prove the following proposition we assume that \(n_{\text{ack}} - f \geq \frac{n+1}{2}\).

**Proposition 5.3. (Validity-2)** If a message \(m\) belongs to \(n_{\text{ack}} - f\) pending sets of correct processes put to the input of the protocol, then \(m \in \text{NCSet}\).

*Proof.* If a message \(m\) belongs to \(n_{\text{ack}} - f\) pending sets of correct processes put to the input of the protocol, then all these \(n_{\text{ack}} - f\) pending sets will belong to the \(\vec{V}\) variable. Moreover, all these \(n_{\text{ack}} - f\) pending sets will go to the \(\text{GS}^k\) variable, since they are all correct. The maximum cardinality of the built \(\text{GS}^k\) variable is \(n\). So, we need that:

\[ n_{\text{ack}} - f \geq \frac{n + 1}{2} \]

And in that case we are guaranteed to put message \(m\) in the \(\text{NCSet}^k\). 

**Proposition 5.4. (Validity-3)** No two messages from \(\text{NCSet}^k\) conflict.

*Proof.* Suppose that the lemma is false and there are two conflicting messages \(m\) and \(m'\) that are present in the majority of the pending sets from \(\text{GS}^k\) variable. It means that there is a \(\text{pending}_j^k \in \text{GS}^k\), s.t:

\[ m, m' \in \text{pending}_j^k \]

So, we get a contradiction with line 4 — there was a \(\text{pending}_j^k \in \vec{V}\), which is not correct. 

**Proposition 5.5. (Validity-4)** There exists \(\text{pending}_j^k\) set of some correct process put to the input of the protocol, s.t. \(\text{pending}_j^k \subseteq \text{NCSet}^k \cup \text{CSet}^k\).
Proof. From the lines 5, 6 we can conclude that:

\[ NCS{\text{et}}^k \cup C{\text{Set}}^k = \bigcup_{\text{pending}^k \in GS^k} \text{pending}^k \]

We just need that in \( GS^k \) there is at least one pending set put as input to the protocol by some correct process. Since \( n > f \), the claim holds. \( \Box \)

**Proposition 5.6.** (Number of processes) \( \mathcal{PGB} \ (\mathcal{C}_{ic}) \) requires at least \( 4f + 1 \) processes with \( n_{ack} = 3f + 1 \).

Proof. We mentioned the following requirements to satisfy the properties of the \( \text{CHK} \) protocol:

\[ n_{ack} - f \geq \frac{n + 1}{2} \]

and

\[ n > f \]

Moreover, we need that the whole \( \mathcal{PGB} \) is correct (see proposition 4.1):

\[ 2n_{ack} > n + f \]

Processes can wait for at most \( n - f \) processes in each phase, so:

\[ n_{ack} \leq n - f \]

Finally, we get the following system:

\[ n_{ack} \leq n - f \]
\[ 2n_{ack} \geq n + 2f + 1 \]

Now we can obtain a bound for \( n \) as follows:

\[ n + 2f + 1 \leq 2n_{ack} \leq 2(n - f) \Rightarrow n \geq 4f + 1 \]

And we can achieve this bound only when \( n_{ack} \geq 3f + 1 \). \( \Box \)
Chapter 6

\[ B \mathcal{G} \mathcal{B} \text{ with Atomic Broadcast, signatures and } n > 5f \]

The algorithm we present next uses the parameterized Generic Broadcast from Chapter 4. In this chapter we provide an implementation of the Check Protocol denoted as \( \mathcal{C}_{\text{absign}} \) (Algorithm 3) and prove that it satisfies the Check Protocol specification (for specification see section 3.3). In order to get a complete algorithm for Byzantine Generic Broadcast, we just need to plug \( \mathcal{C}_{\text{absign}} \) in \( P \mathcal{G} \mathcal{B} \). \( \mathcal{C}_{\text{absign}} \) uses digital signatures [RSA83] and Atomic Broadcast [BT85].

6.1 Algorithm \( \mathcal{C}_{\text{absign}} \)

We will denote message \( m \) signed by process \( p_j \) as \( \langle m \rangle_{\delta_j} \). The signed messages satisfy the following properties:

- If a correct process \( p_j \) signed message \( m \), then all other correct processes can verify that \( \langle m \rangle_{\delta_j} \) was signed by \( p_j \).

- If a correct process \( p_i \) verified that message \( \langle m \rangle_{\delta_j} \) was signed by \( p_j \), then all other correct processes will verify that \( \langle m \rangle_{\delta_j} \) was signed by \( p_j \).

- If a correct process \( p_i \) verified that message \( \langle m \rangle_{\delta_j} \) was signed by \( p_j \), then \( p_i \) cannot verify that the same message \( \langle m \rangle_{\delta_j} \) was signed by another process \( p_k \), s.t. \( k \neq j \)

- It is impossible to ‘forge’ signatures of correct processes by providing signed message \( \langle m \rangle_{\delta_j} \), s.t. a correct process \( p_j \) has never signed \( m \).
Definition 6.1. If a correct process \( p \) ADelivers messages of the form \( \langle k, \text{pending}^k_j, \text{CHK} \rangle \delta \) from other processes in round \( k \), then specific message \( \langle k, S_1, \text{CHK} \rangle \delta \) received by \( p \) is said to be unique iff \( p \) has not ADelivered messages \( \langle k, S_2, \text{CHK} \rangle \delta \) in the round \( k \), s.t. \( S_1 \neq S_2 \).

A correct message is defined as in definition 5.1. Algorithm 3 is similar to Algorithm 2 except that the properties of the \( G^k \) variable have changed. In Algorithm 2 all pending sets of correct processes are put to \( G^k \), while in Algorithm 3 we are sure to put pending sets of \( n_{\text{chk}} \) different processes (including correct and possibly byzantine) to \( G^k \). Hence, only \( n_{\text{chk}} - f \) correct pending sets appear in \( G^k \). Later, we will select \( n_{\text{chk}} \), s.t. if a message \( m \) belongs to \( n_{\text{ack}} - f \) pending sets of correct processes, then \( m \) will appear in a majority of the pending sets of \( G^k \).

**Algorithm 3: A Check Protocol with Atomic Broadcast and signatures (\( \mathcal{C}^\text{absign} \))**

1. Procedure proposeCheck\((k, \text{pending}^k)\)
2. \[\text{ABcast} \langle k, \text{pending}^k \rangle \delta \] to all
3. \[\text{wait until} \ |G^k| = n_{\text{chk}} \triangleq \{\text{pending}^k_j \mid \text{ADelivered unique correct} \langle k, \text{pending}^k_j \rangle \delta \text{ from } p_j \}\]
4. \(N\text{CSet}^k \leftarrow \{m \mid \exists \lceil \frac{n_{\text{chk}}+1}{2} \rceil \text{ sets } \text{pending}^k_j \in G^k : m \in \text{pending}^k_j \}\)
5. \(C\text{Set}^k \leftarrow \bigcup \text{pending}^k_j \in G^k \text{pending}^k_j \setminus N\text{CSet}^k\)
6. decideCheck\((k, N\text{CSet}^k, C\text{Set}^k)\)

6.2 A proof of correctness

**Lemma 6.1.** A \( \text{pending}^k_j \in G^k \) with conflicting messages.

*Proof.* Follows directly from the fact that we wait for correct messages at line 3. \( \square \)

**Lemma 6.2.** \( G^k \) contains pending sets from \( n_{\text{chk}} \) different processes.

*Proof.* Follows directly from the fact that we wait for unique messages at line 3. \( \square \)

**Proposition 6.1.** (Agreement and termination) If all correct processes execute proposeCheck\((\text{pending}_j)\), then all correct processes eventually decide on the same pair of messages sets \( (N\text{CSet}^k, C\text{Set}^k) \).

*Proof.* To show termination we need to verify that condition at line 3 is eventually satisfied. Suppose there is a correct process \( p \) that never terminates and is stuck at line 3. If \( n_{\text{chk}} \leq n - f \), then there is a moment when \( p \) ADelivered \( n_{\text{chk}} \) messages
from correct processes. All these messages are unique and correct, so $p$ would be able to form the $GS^k$ variable, s.t. $|GS^k| \geq n_{chk}$, so there is a moment when $|GS^k| = n_{chk}$ and $p$ has to proceed to the next line 4, a contradiction.

Since Atomic Broadcast delivers messages in the same order on all correct processes, and properties correct and unique are checked deterministically and uniformly on all correct processes, they all will form the same $GS^k$ variable at the line 3. Since $NCS^k$ and $CSet^k$ are just deterministic functions of $GS^k$, all the correct processes will decide on the same pair of message sets $(NCS^k, CSet^k)$. \hfill \square

**Proposition 6.2.** (Validity-1) $NCS^k \cap CSet^k = \emptyset$.

**Proof.** Follows immediately from the definition of $CSet^k$ at line 5. \hfill \square

To prove the following proposition we assume that $n_{chk} + n_{ack} - f - n \geq \left\lceil \frac{n_{chk}+1}{2} \right\rceil$.

**Proposition 6.3.** (Validity-2) If a message $m$ belongs to $n_{ack} - f$ pending sets of correct processes put to the input of the protocol, then $m \in NCS$.

**Proof.** At line 3 processes wait to receive pending sets from $n_{chk}$ different processes. The size of an intersection of $n_{chk}$ and $n_{ack} - f$ processes is at least $n_{chk} + n_{ack} - f - n$. So, if $m$ belongs to $n_{ack} - f$ pending sets of correct processes, it will appear in $n_{chk} + n_{ack} - f - n$ pending sets in $GS^k$. Now we need that $n_{chk} + n_{ack} - f - n$ pending sets form a majority among all pending sets in $GS^k$:

$$n_{chk} + n_{ack} - f - n \geq \left\lceil \frac{n_{chk} + 1}{2} \right\rceil$$

And in that case we are guaranteed to put message $m$ in $NCS^k$. \hfill \square

**Proposition 6.4.** (Validity-3) No two messages from $NCS^k$ conflict.

**Proof.** Suppose that the lemma is false and there are two conflicting messages $m$ and $m'$ that are present in the majority of the pending sets from $GS^k$ variable. It means that there is a $pending^k_j \in GS^k$, s.t:

$$m, m' \in pending^k_j$$

So, we get a contradiction with lemma 6.1 — there is a $pending^k_j \in GS^k$ with conflicting messages. \hfill \square

**Proposition 6.5.** (Validity-4) There exists a $pending^k_j$ set of some correct process put to the input of the protocol, s.t. $pending^k_j \subseteq NCS^k \cup CSet^k$. 
Proof. From lines 4, 5 we can conclude that:

\[ N\text{CSet}^k \cup C\text{Set}^k = \bigcup_{\text{pending}_j^k \in \text{GS}^k} \text{pending}_j^k \]

We just need that in \( GS^k \) there is at least one pending set put as input to the protocol by some correct process. Due to the lemma 6.2 this means that we require \( n_{chk} > f \).

Proposition 6.6. (Number of processes) \( P \& B \) \((C_{absign})\) requires at least \( 5f + 1 \) processes with \( n_{ack} = n_{chk} = 4f + 1 \).

Proof. We mentioned the following requirements to satisfy the properties of the CHK protocol:

\[ n_{chk} + n_{ack} - f - n \geq \left\lceil \frac{n_{chk} + 1}{2} \right\rceil \iff n_{chk} + 2n_{ack} \geq 2n + 2f + 1 \]

and

\[ n_{chk} > f \]

Moreover, we need that the whole \( P \& B \) is correct (see proposition 4.2):

\[ 2n_{ack} > n + f \]

Processes can wait for at most \( n - f \) processes in each phase, so:

\[ n_{ack} \leq n - f \]
\[ n_{chk} \leq n - f \]

We thus get the following equivalent system:

\[ n_{ack} \leq n - f \]
\[ n_{chk} \leq n - f \]
\[ n_{chk} + 2n_{ack} \geq 2n + 2f + 1 \]

Now we can obtain a bound for \( n \) as follows:

\[ 2n + 2f + 1 \leq n_{chk} + 2n_{ack} \leq 3(n - f) \implies n \geq 5f + 1 \]

And we can achieve this bound only when \( n_{ack} = n_{chk} = n - f = 4f + 1 \).
6.3 On the usage of signatures

Unfortunately, the usage of digital signatures can be expensive in terms of the time the processes need to sign messages and verify the signatures. We could get rid of the signatures using the witness quorums that verify if a message was correctly signed [ST87]. However, it is said in [AABC08] that this method requires infinite storage and can be inefficient. From a theoretical point of view, it is important to notice that using the method [ST87], we can transform all algorithms with signatures to algorithms without signatures if $n > 3f$ and adding a constant amount of network delays for each signing/verification procedure. Moreover, usage of the classical approach [RSA83] requires system to distribute the public keys among the processors, which is a complicated problem by itself (actually, it is exactly Interactive Consistency [PSL80]), while [ST87] does not need any special initialization.

So far, the problem with the classical implementation of digital signatures [RSA83] is the expensive operation of signing messages, while verification is considerably fast if the algorithm uses a small public key exponent [Rab79]. One way to overcome the high cost of producing the signature was proposed in [AABC08], where the authors build signatures as matrices of MACs [BCK96]. In [AABC08], the approach to sign/verify messages needs a constant number of network delays combined with very fast operation of issuing MAC for a message. Another solution is using elliptic curves cryptography for digital signatures ECDSA [Sol00]. The advantage is that the procedure of signing a message becomes faster compared to normal RSA [RSA83], while verification of message signatures is slow.

All in all, it looks like there is a trade-off in the usage of digital signatures and there is no common rule on how to implement them. In that sense, for a practical implementation of the proposed algorithm a careful analysis should be done, finding out what implementation of digital signatures suits the algorithm better.
6.3 On the usage of signatures
Chapter 7

BG B with Selection Protocol and $n > 9f$

In this chapter, we provide an implementation of the Check Protocol $C_{select}$ (algorithm 4) and prove that it satisfies the Check Protocol specification (for specification see section 3.3). In order to get a complete algorithm for Byzantine Generic Broadcast, we just need to plug $C_{select}$ in $P G B$. $C_{select}$ uses Selection Protocol [BOEY03].

The removal of signatures introduces a fundamental problem in the design of the algorithm — processes cannot retransmit messages anymore, since messages authenticity cannot be verified. In order to be able to overcome this problem, we introduce a special distributed primitive called the Selection Protocol. The Selection Protocol was first introduced in [BOEY91] to solve the Interactive Consistency problem [PSL80]. However, [BOEY91] was never published and later the same authors published new paper [BOEY03] where we can find the definition of the Selection Protocol. The major disadvantage of the Selection Protocol presented in [BOEY03] is that this algorithm is randomized, we present a deterministic version of the Selection Protocol in Chapter 8.

7.1 The Selection Protocol specification

Roughly speaking, the Selection Protocol is a consensus protocol that additionally requires the decided value to be meaningful. One simple setting in which such an algorithm is useful is a problem of time synchronization among the processes in the presence of byzantine failures. Suppose we solve this problem by using a traditional consensus protocol [Lam01] — we put processes times as inputs to consensus, so all correct processes will decide on the same time. However, if there
is a byzantine process that proposes time 0 (something completely useless), we can end up by deciding on this useless time.

Suppose that all processes have some initial value \( v_i \) to propose. Additionally, all correct processes have predicates \( P_i \) that define if any value \( v \) is meaningful or not from the viewpoint of process \( p_i \). The abstraction requires that for all correct processes \( p_i \) and \( p_j \), predicate \( P_i(v_j) \) holds (also denoted as \( \models P_i(v_j) \)). The Selection Protocol is defined by primitives \( \text{proposeSelect}(v_i, P_i) \) and \( \text{decideSelect}(v) \). Processes invoke \( \text{proposeSelect}(v_i, P_i) \) to propose a value \( v_i \) and then decide on a value \( v \) with \( \text{decideSelect}(v) \). The Selection Protocol guarantees the following properties:

- (Termination) All correct processes eventually decide on some value \( v \).
- (Agreement) No two correct processes decide differently.
- (Selection) If some correct process decides on value \( v \), then there exists correct process \( p_i \) such that \( \models P_i(v) \).

We tag instances of the Selection Protocol with the round number \( k \), thus invoking \( \text{proposeSelect}(k, v_i, P_i) \) and \( \text{decideSelect}(k, v) \).

### 7.2 Algorithm \( \mathcal{C}_{\text{select}} \)

The pseudocode of \( \mathcal{C}_{\text{select}} \) is presented in Algorithm 4. There we use definitions of correct (see definition 5.1) and unique (see definition 6.1) messages. \( \mathcal{C}_{\text{select}} \) as \( \mathcal{C}_{ic} \) and \( \mathcal{C}_{absign} \) builds \( GS^k \) variable with the pending sets from the processes. In this case every process \( p_i \) builds its own variable \( GS_i^k \) representing partial global state of the system at lines 1, 2. Afterwards, processes run the Selection Protocol in order to decide on \( GS^k \), which is used to build \( NCS^k \) and \( CSet^k \). Here we use the Selection Protocol in order to prevent byzantine processes from proposing \( GS^k \) composed maliciously, e.g. when all pending sets of \( GS^k \) contain conflicting messages.

### 7.3 A proof of correctness

**Proposition 7.1.** (Agreement and termination) If all correct processes execute \( \text{proposeCheck}(\text{pending}_j) \), then all correct processes eventually decide on the same pair of messages sets \( (NCS^k, CSet^k) \).
Algorithm 4: A Check Protocol with the Selection Protocol ($\mathcal{G}_{select}$)

1. send $(k, \text{pending}^k)$ to all
2. wait until $|\mathcal{G}^k| = n_{chk}$ : $\mathcal{G}^k \overset{\text{def}}{=} \{\text{pending}^k| \text{Received unique correct } (k, \text{pending}^k) \text{ from } p_j\}$
3. Predicate $Pred(X)$ is defined as following:
4. \[ \text{if } (|X| = n_{chk}) \text{ and } (|\mathcal{G}^k \cap X| \geq (2n_{chk} - n - f)) \text{ and } (\text{all pending } \in X \text{ are correct}) \text{ then} \]
5. \[ \text{return } \text{true} \]
6. \[ \text{else} \]
7. \[ \text{return } \text{false} \]
8. \[ \text{end} \]
9. proposeSelect($k, \mathcal{G}^k, Pred$)
10. wait until decideSelect($k, \mathcal{G}^k$)
11. $N CSet^k \leftarrow \{m \mid \exists [\frac{2n_{chk} + 1}{2}] \text{ sets } \text{pending}^k \in \mathcal{G}^k : m \in \text{pending}^k\}$
12. $CSet^k \leftarrow \bigcup_{\text{pending}^k \in \mathcal{G}^k} \text{pending}^k \setminus N CSet^k$
13. decideCheck($k, N CSet^k, CSet^k$)

Proof. To show termination we need to verify that condition at line 2 is eventually satisfied. Suppose there is a correct process $p$ that never terminates and is stuck at line 2. If $n_{chk} \leq n - f$, then there is a moment then $p$ receives $n_{chk}$ messages from all correct processes. All these messages are unique and correct, so $p$ will be able to form $\mathcal{G}^k$ variable s.t. $|\mathcal{G}^k| \geq n_{chk}$, so there is a time when $|\mathcal{G}^k| = n_{chk}$ and $p$ proceeds to the next line 9, a contradiction! Now we need to show that requirements of the Selection Protocol are satisfied, so it terminates — for all correct processes $p_i$, and $p_j$, predicate $P_i(\mathcal{G}^k)$ holds. We will check all three conditions from the line 4:

- $|\mathcal{G}^k| = n_{chk}$ by construction.

- $|\mathcal{G}^k \cap \mathcal{G}^k| \geq (2n_{chk} - n - f))$. Since this is just as an intersection of two sets of size $n_{chk}$ and up to the $f$ pending sets in intersection can be sent by byzantine.

- (All pending $\in \mathcal{G}^k$ are correct) by construction.

By the agreement property of the Selection Protocol all correct processes will decide on the same value $\mathcal{G}^k$ at the line 10, so they will all construct the same pair of message sets $(N CSet^k, CSet^k)$.

Proposition 7.2. \textit{(Validity-1)} $N CSet^k \cap CSet^k = \emptyset$.

Proof. Follows immediately from the definition of $CSet^k$ at line 12.
**Proposition 7.3.** (Validity-2) If a message \( m \) belongs to \( n_{\text{ack}} - f \) pending sets of correct processes put to the input of the protocol, then \( m \in N\text{CSet} \).

**Proof.** Let’s denote the set of pending sets of correct processes with \( m \) as \( A \). The Selection Protocol guarantees that at line 10 processes decide on the value \( GS^k_y \) that is verified by some correct process \( p_x \). Suppose process \( p_x \) proposed some value \( GS^k_x \). We know that \( |GS^k_x \cap GS^k_y| \geq 2n_{\text{chk}} - n - f \). We know that \( |GS^k_x \cap A| \geq n_{\text{chk}} + (n_{\text{ack}} - f) - n \), because \( p_x \) is a correct process, and \( p_x \) forms its \( GS^k_x \) variable from \( n_{\text{chk}} \) pending sets of different processes. Furthermore, the intersection \( GS^k_x \cap A \) consists of pending sets of correct processes, since \( A \) is composed of pending sets of correct processes. We need that \( |GS^k_y \cap A| \geq \left\lceil \frac{n_{\text{chk}} + 1}{2} \right\rceil \) for message \( m \) to be included in \( N\text{CSet}^k \). Let’s estimate \( |GS^k_y \cap A| \):

\[
|GS^k_y \cap A| \geq |GS^k_y \cap A \cap GS^k_x| \geq |GS^k_x \cap A| + |GS^k_y \cap GS^k_x| - |GS^k_x|
\]

So, we can rewrite this inequality using \( n_{\text{ack}}, n_{\text{chk}}, n, \) and \( f \):

\[
\text{int} := |GS^k_y \cap A| \geq \left[(n_{\text{ack}} + n_{\text{chk}} - n - f) + (2n_{\text{chk}} - n - f) - n_{\text{chk}}\right]
\]

For message \( m \) to be included in \( N\text{CSet}^k \), we need that:

\[
|GS^k_y \cap A| \geq \left\lceil \frac{n_{\text{chk}} + 1}{2} \right\rceil \iff \text{int} \geq \left\lceil \frac{n_{\text{chk}} + 1}{2} \right\rceil \iff 2\text{int} \geq n_{\text{chk}} + 1 \iff 2((n_{\text{ack}} + n_{\text{chk}} - n - f) + (2n_{\text{chk}} - n - f) - n_{\text{chk}}) \geq n_{\text{chk}} + 1 \iff 3n_{\text{chk}} + 2n_{\text{ack}} \geq 4n + 4f + 1
\]

Later, we derive exact bounds on \( n_{\text{chk}} \) and \( n_{\text{ack}} \) considering the last inequality. The important point is that here we considered the intersection of set \( A \) possibly received by a correct process in the ACK phase, the CHK set built by another correct process \( (GS^k_x) \), and the CHK set possibly proposed by a byzantine process \( (GS^k_y) \). The illustration of the intersections between \( A, GS^k_x \) and \( GS^k_y \) from this proof is presented in the figure 7.1. The tricky part is to consider that \( GS^k_y \) could be proposed by a byzantine process. If this possibility is not taken into account, the reasoning is flawed (c.f. Appendix A).

**Proposition 7.4.** (Validity-3) No two messages from \( N\text{CSet}^k \) conflict.
Proof. Suppose that there are two conflicting messages $m$ and $m'$ that are present in the majority of the pending sets from $GS^k_y$ variable verified by correct process $p_x$. It means that there is a $\text{pending}^k_j \in GS^k_y$, s.t:

$$m, m' \in \text{pending}^k_j$$

So, we get a contradiction with the predicate $P_x$ — there is a $\text{pending}^k_j \in GS^k_y$ with conflicting messages. \hfill \square

**Proposition 7.5.** (Validity-4) There exists $\text{pending}^k_j$ set of some correct process put to the input of the protocol, s.t. $\text{pending}^k_j \subseteq NCSek \cup CSek$.

**Proof.** From lines 11, 12 we conclude that:

$$NCSek \cup CSek = \bigcup_{\text{pending}^k_j \in GS^k_y} \text{pending}^k_j$$

We require $GS^k_y$ to contain at least one pending put as input to the protocol by some correct process. Suppose $GS^k_y$ was verified by a correct process $p_x$. Since
$P_x(GS^k_y)$ is true, then we need that $2n_{chk} - n - f > f$. In that case, at least one pending set from the intersection $GS^k_y \cap GS^k_x$ is guaranteed to be sent by a correct process, thus this pending set belongs to $GS^k_y$. So, we require that:

$$2n_{chk} > n + 2f$$

Later, we derive exact bounds on $n_{chk}$ and $n_{ack}$ considering the last inequality.

**Proposition 7.6.** *(Number of processes)* To make the protocol work we need at least $9f + 1$ processes with $n_{ack} = n_{chk} = 8f + 1$.

**Proof.** We stated the following requirements:

$$2n_{ack} + 3n_{chk} \geq 4n + 4f + 1$$
$$2n_{chk} > n + 2f$$

and we need the whole $\mathcal{P} \mathcal{G} \mathcal{B}$ to be correct:

$$2n_{ack} > n + f$$

We know that $n_{ack}$ and $n_{chk}$ are both less or equal than $n - f$, so, we get the following inequality that bounds $n$:

$$5(n - f) \geq 4n + 4f + 1 \iff n \geq 9f + 1$$

And it becomes equality when $n_{ack} = n_{chk} = n - f = 8f + 1$. 

$\square$
Chapter 8

A deterministic Selection Protocol

The definition of the Selection Protocol is presented in the section 7.1. Here we present two deterministic implementations of the Selection Protocol. In both implementations we employ Byzantine Binary Consensus (functions proposeStrongValid and decideStrongValid). All correct processes start the protocol by proposing their initial value $v_i \in \{0, 1\}$, Byzantine Binary Consensus ensures the following properties:

- (Termination) All correct processes will terminate by deciding on some value.
- (Agreement) No two correct processes decide on the different value.
- (Strong validity) If all correct processes have the same initial value $v$, then the decided value is $v$.

Algorithms that satisfy these properties are presented in [MHS09, SR08].

8.1 A Selection Protocol with leader-based consensus

Assume we have leader-based Byzantine Multivalued Consensus (functions proposeLeader, decideLeader) with opportunity to change the leader (function changeLeader) like [CL02, MA06]. The algorithm’s pseudocode is presented in Algorithm 5. This algorithm needs a correct leader to be chosen eventually and that this leader’s proposal is the decision. The algorithm’s principal scheme is devised in three steps:

- Using rotating coordinator correct processes decide on some value $v$.  

• Processes verify \( v \) using Byzantine Binary Consensus.

• If \( v \) is verified, then algorithm decides. Otherwise, coordinator is changed and the whole procedure is repeated.

\begin{algorithm}
\begin{algorithmic}
\STATE Initialization:
\STATE \( x_p \leftarrow v_p \in V \) /* \( v_p \) is \( p \)'s initial value*/
\STATE To decide process executes these commands until \textit{decideCorrect} is executed:
\STATE \quad proposeLeader(\( v_p \))
\STATE \quad decideLeader(\( x \))
\STATE \quad \( \text{flag} \leftarrow \text{correct}(x, v_p) \)
\STATE \quad proposeStrongValid(\( \text{flag} \))
\STATE \quad decideStrongValid(\( \text{decided-flag} \))
\STATE \quad \textbf{if} (\( \text{decided-flag} \)) \textbf{then}
\STATE \quad \quad \text{decideCorrect}(x)
\STATE \quad \textbf{else}
\STATE \quad \quad \text{changeLeader()}
\STATE \textbf{end}
\end{algorithmic}
\end{algorithm}

### 8.2 A Selection Protocol with many consensus instances

Suppose we have Byzantine Multivalued Consensus algorithm that needs at least \( x \) processes to tolerate \( f \) faults. We try to run simultaneously many consensus instances, s.t. one instance is composed only of correct processes. In that case, we need \( n \geq x + f \) and we run the following number of consensus instances for different subset of processes:

\[
\binom{n}{x} - \binom{n-f}{x} + 1
\]

where \( \binom{n}{x} \) is the total number of consensuses composed of different quorums we can run, \( \binom{n-f}{x} \) is the total number of quorums that do not contain byzantine processes. Hence, \( \binom{n}{x} - \binom{n-f}{x} \) is the total number of quorums with at least one byzantine process in each quorum. Finally, we just take one more quorum to be sure that
at least one quorum among \( \binom{n}{k} - \binom{n-f}{k} + 1 \) is composed of correct processes only. After all \( \binom{n}{k} - \binom{n-f}{k} + 1 \) consensus instances terminate, we run again the same amount of \textit{Byzantine Binary Consensus} instances with \textit{strong validity} to verify each of the decided values. Afterwards, correct processes just select deterministically one value that was decided to be correct.
8.2 A Selection Protocol with many consensus instances
Chapter 9

Conclusion

To the best of the author’s knowledge this thesis has provided the first implementations of Byzantine Generic Broadcast. The provided algorithms (\text{PGB}(c_i), \text{PGB}(c_{absign}), \text{PGB}(c_{select})) share the same architecture of the parameterized \text{Generic Broadcast}. This parameterized algorithm is a generalization of the \text{GB} algorithm from [PS99]. The main advantage of \text{PGB} is that it provides a modular approach for building \text{Generic Broadcast} protocols — instead of building the whole algorithm from scratch, it is enough to provide a valid \text{Check Protocol}. Moreover, a proof of correctness of the resulting algorithm is simplified by splitting it into two disjoint parts: a correctness proof of \text{PGB}, which is done once and for all, and a correctness proof of the \text{Check Protocol}.

Another contribution of this thesis is a presentation of the first deterministic Selection Protocol. In general there is no problem of using its randomized version from [BOEY03], however, in safety-critical services it would be more natural to use the deterministic version of the protocol due to the simplicity of its assumptions. In that case, when proving the correctness of the whole system, it is not required to reason about the probabilistic termination or expected number of rounds to decide.

Byzantine Generic Broadcast can be used to implement State Machine Replication. Clients just need to g-Broadcast their requests to replicas, replicas will g-Deliver all requests in the order that takes into account the conflict relation \(\sim\), execute requests in the order of their g-Delivery, and send results back to the client. The first practical algorithm for State Machine Replication that tolerates byzantine faults was proposed in [CL02], more work has been done in recent years [KAD+09, CML+06, AEMGG+05]. The contribution of State Machine Replication with Byzantine Generic Broadcast is that it allows a usage of \(\sim\) conflict relation.
benefiting from requests semantics, while other algorithms use total ordering. The comparison of State Machine Replication algorithms for a Byzantine model is presented in the table 9.1. In this table ‘Best-case delay’ refers to the number of the network delays required to execute one client request using special assumptions (failure-free or contention-free runs). A run is failure-free iff no failures of replica or client occur. A run is contention-free iff only one client sends requests to replicas.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type</th>
<th>n</th>
<th>Best-case delay</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBFT [CL02]</td>
<td>total order</td>
<td>$3f + 1$</td>
<td>4</td>
<td>FF</td>
</tr>
<tr>
<td>Zyzzyva [KAD+09]</td>
<td>total order</td>
<td>$3f + 1$</td>
<td>3</td>
<td>FF</td>
</tr>
<tr>
<td>HQ [CML+06]</td>
<td>total order</td>
<td>$3f + 1$</td>
<td>4</td>
<td>FF, CF</td>
</tr>
<tr>
<td>Q/U [AEMGG+05]</td>
<td>total order</td>
<td>$5f + 1$</td>
<td>2</td>
<td>FF, CF</td>
</tr>
<tr>
<td>$\mathcal{PB} (C_{ic})$</td>
<td>~ order</td>
<td>$4f + 1$</td>
<td></td>
<td>FF, $\exists m_1, m_2: m_1 \sim m_2$</td>
</tr>
<tr>
<td>$\mathcal{PB} (C_{absgn})$</td>
<td>~ order</td>
<td>$5f + 1$</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$\mathcal{PB} (C_{select})$</td>
<td>~ order</td>
<td>$9f + 1$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 9.1: The costs of byzantine fault-tolerant replication protocols (based on the table from [KAD+09]). Here FF means failure-free and CF means contention-free.

In this table we see that the algorithm with the smallest possible best-case delay is Q/U. The point is that it uses speculative [KAD+09] execution (also called the optimistic approach [PS03], or tentative execution). The idea of speculative execution is that the replicas can start executing the commands right away after receiving them, assuming that all replicas receive them in the same order. The advantage of tentative execution is that we improve the best-case network delay, while disadvantage is that sometimes we have to perform a rollback of the executed requests. In this thesis no optimistic Byzantine Generic Broadcast has been delivered, however, the author believes that one can combine the $\mathcal{PB}$ with tentative execution to get State Machine Replication with a best-case latency of two network delays.

The real costs of the State Machine Replication could be different in practical implementations [CMW+08] as opposed to the table 9.1. All in all, implementing byzantine fault-tolerant State Machine Replication is a challenging task, which involves not only a theoretical construction of the algorithms but a careful engineering as well. This work provided a distributive primitive called Byzantine Generic Broadcast, which is valuable for implementing a concrete State Machine Replication algorithms and as a theoretical work generalized and simplified previous results on Generic Broadcast [PS99].
Appendix A

An example of incorrect reasoning for Algorithm 4 in Chapter 7

It can be not immediately clear, why the following reasoning does not work (we use notation from the proof of the proposition 7.3):

The intersection of the set \( A \) and \( GS^k \) should have at least \( (n_{\text{ack}} + n_{\text{chk}}) - n - f \) correct pending sets. The set \( (A \cap GS^k_x) \) and \( GS^k_y \) should have \( n_{\text{ack}} + 2n_{\text{chk}} - 2n - f \) correct pending sets in the intersection. This holds if \( GS^k_y \) was proposed by correct process, which is an essential assumption to make this statement but does not necessarily holds. Now we just want the number of correct processes in this intersection \( A \cap GS^k_x \cap GS^k_y \) to be a majority in any \( n_{\text{chk}} \) set:

\[
n_{\text{ack}} + 2n_{\text{chk}} - 2n - f \geq \frac{n_{\text{chk}} + 1}{2} \quad \text{(when } n_{\text{chk}} = n_{\text{ack}} = n - f \text{)} \quad n \geq 7f + 1
\]

So, in that way we could have obtained a new but incorrect bound for \( n \).

The counterexample for this reasoning is presented in the figure A.1. In this figure we built \( GS^k_y \) variable in a way s.t. \( GS^k_y \) is verified by a correct process but formed maliciously. \( GS^k_y \) does not have sufficiently many correct pending sets with the message that was \( g \)-Delivered before (belongs to \( n_{\text{ack}} \) pending sets).
Figure A.1: The counterexample for possibility of straightforward message verification when \( n = 7f + 1 \). Message \( m \) is g-Delivered, the value decided \( GS^k_y \) is successfully verified by some correct but does not have \( m \) in the majority of the pending sets (only in \( 2f + 1 \) instead of \( 3f + 1 \)).
Bibliography


