Use case: Shared Calendar

A Master’s Thesis
Submitted to the Faculty of the
Escola Tècnica d’Enginyeria de Telecomunicació de Barcelona
Universitat Politècnica de Catalunya
by
Francesc Alonso Belchi

In partial fulfilment
of the requirements for the degree of
MASTER IN TELEMATICS ENGINEERING

Advisor: Juan Luis Gorricho Moreno

Barcelona, September 2014
Abstract

In distributed systems arise several issues that are solved using distributed algorithms. The main purpose of this thesis is to develop a system for academic use that allows to implement distributed algorithms and test them. As a result, a set of tools has been developed, which allow to easily implement distributed algorithms and combine them to build distributed systems. Finally, it was decided to prove their utility by building a system that can be integrated into a final application, called Shared Calendar.
# Contents

**Introduction**  

1 Tools  

1.1 Communication System  

1.1.1 Transport protocol  

1.1.2 Message  

1.1.3 Buffers  

1.1.4 Configuration files  

1.1.4.1 NetConf  

1.1.4.2 Show and PrintMessage  

1.1.5 Stub  

1.1.6 Node  

1.2 Server of Identifiers  

1.3 Timeout and Periodic Task  

1.3.1 TOTask  

1.3.2 PTTask  

1.4 Log  

1.5 Limiting the active nodes in an algorithm  

1.6 Sharing the functionalities  

1.7 Reconciliation  

1.7.1 Log Management  

1.7.2 Reconciler  

---

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1 Tools</td>
<td>5</td>
</tr>
<tr>
<td>1.1 Communication System</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1 Transport protocol</td>
<td>5</td>
</tr>
<tr>
<td>1.1.2 Message</td>
<td>6</td>
</tr>
<tr>
<td>1.1.3 Buffers</td>
<td>7</td>
</tr>
<tr>
<td>1.1.4 Configuration files</td>
<td>8</td>
</tr>
<tr>
<td>1.1.4.1 NetConf</td>
<td>8</td>
</tr>
<tr>
<td>1.1.4.2 Show and PrintMessage</td>
<td>8</td>
</tr>
<tr>
<td>1.1.5 Stub</td>
<td>9</td>
</tr>
<tr>
<td>1.1.6 Node</td>
<td>10</td>
</tr>
<tr>
<td>1.2 Server of Identifiers</td>
<td>11</td>
</tr>
<tr>
<td>1.3 Timeout and Periodic Task</td>
<td>13</td>
</tr>
<tr>
<td>1.3.1 TOTask</td>
<td>13</td>
</tr>
<tr>
<td>1.3.2 PTTask</td>
<td>14</td>
</tr>
<tr>
<td>1.4 Log</td>
<td>14</td>
</tr>
<tr>
<td>1.5 Limiting the active nodes in an algorithm</td>
<td>14</td>
</tr>
<tr>
<td>1.6 Sharing the functionalities</td>
<td>15</td>
</tr>
<tr>
<td>1.7 Reconciliation</td>
<td>15</td>
</tr>
<tr>
<td>1.7.1 Log Management</td>
<td>16</td>
</tr>
<tr>
<td>1.7.2 Reconciler</td>
<td>18</td>
</tr>
</tbody>
</table>
2 Distributed Algorithms

2.1 Synchronization Algorithms ........................................... 19
  2.1.1 Physical time .................................................. 19
     2.1.1.1 Christian ................................................. 19
     2.1.1.2 Berkeley ............................................... 20
  2.1.2 Logical time ................................................... 22
     2.1.2.1 Logical clocks ........................................... 23
     2.1.2.2 Vector clocks ............................................ 24

2.2 Mutual Exclusion Algorithms ....................................... 25
  2.2.1 Centralized .................................................... 26
     2.2.1.1 Centralized coordinator implementation ............... 27
     2.2.1.2 Centralized Stub implementation ....................... 27
  2.2.2 Token Ring .................................................... 27
     2.2.2.1 Token Ring Stub implementation ......................... 28
  2.2.3 Lamport ....................................................... 29
     2.2.3.1 Lamport Stub implementation ........................... 30
  2.2.4 Ricart and Agrawala ......................................... 32
     2.2.4.1 Ricart and Agrawala Stub implementation ............... 33

2.3 Election Algorithms ................................................. 34
  2.3.1 Chang and Roberts Ring ....................................... 35
     2.3.1.1 Ring implementation ..................................... 36
  2.3.2 Bully .......................................................... 36
     2.3.2.1 Bully implementation ..................................... 37

2.4 Consensus Algorithms ............................................... 41
  2.4.1 Paxos .......................................................... 41
     2.4.1.1 Paxos implementation .................................... 44

2.5 Transaction Algorithms ............................................. 54
  2.5.1 Two-phase commit protocol .................................... 55
     2.5.1.1 Two-phase commit protocol implementation ............. 57
CONTENTS

3 Use Case: Shared Calendar 63
  3.1 Selection of the algorithms . . . . . . . . . . . . . . . . . . . . . . . . 64
  3.2 Replicated log . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
  3.3 Calendar GUI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

Conclusions and future development 71

Bibliography 73

Glossary 75
List of Figures

1.1.1 Message data JSON String Structure ................................. 6
1.1.2 Buffers HashMap .................................................. 7
1.1.3 Communication System’s message flow .............................. 11
1.7.1 Elements of the reconciliation mechanism ......................... 16

2.1.1 Christian synchronization ........................................ 20
2.1.2 Logical clocks mechanism ........................................ 24
2.1.3 Vector clocks mechanism .......................................... 25
2.2.1 Ring of processes .................................................. 28
2.3.1 Bully algorithm. Processes in color gray have failed .......... 38
2.3.2 State machine that shows the states and transitions that happen during a Bully election .............................................. 39
2.4.1 Paxos instance execution. Example of a best case scenario ...... 44
2.4.2 Paxos instance execution. Example of a worst case scenario ... 45
2.4.3 State machine that shows the states and transitions that happen in a consensus instance ............................................ 52
2.5.1 2PC (a) Coordinator’s state machine; (b) Participant’s state machine [6] 56

3.2.1 A calendar replica .................................................. 66
3.2.2 Example of execution of the replicated log: The calendar replicas surrounded by red dashes are in the Active set ....................... 68
Introduction

A distributed system is a set of nodes on a network that communicate between them by exchanging messages to coordinate their actions in order to perform a set of tasks.

During the design of distributed systems are found several issues that are handled with distributed algorithms. The most common issues are the following:

- Time: The time is used to know the order of events that occur in a distributed system. The issue is how to timestamp each event, in order to know when an event happened before another.

- Share a resource: The issue is how to coordinate the access of a collection of processes to a shared resource.

- Election: The problem to solve is how a set of processes agree to elect a single process to play a particular role.

- Consensus: How a collection of processes agree to choose a single value among others.

- Transactions: How multiple processes reach to the same decision to eventually complete a transaction successfully.

The main motivation to perform this thesis about distributed systems arose from a proposal made by Dr. Juan Luis Gorricho Moreno, the advisor of this thesis. The proposal was to build a system in Java with which the students can implement and execute distributed algorithms.

During the realization of this thesis, distributed algorithms have been implemented based on their specification, which solve the problems previously presented. The implementation of these algorithms has allowed to develop the necessary tools, in order to the students to be able to implement distributed algorithms, and combine them to build distributed applications. In this way, the implemented tools can be tested in
order to verify that they perform their function correctly, as well as that are useful and easy to use.

Finally, a distributed system that can be integrated in a final distributed application, to be used in the real world, was built with the implemented tools and a set of distributed algorithms.

Note that in this thesis the words node and process are used indistinguishably.

The project is structured in the following chapters. Described in the chapter 1 are the developed tools to be used by the students. The chapter 2 discusses the distributed algorithms. For each algorithm first the theoretical description of the algorithm is presented, then the implementation of the algorithm is described. The chapter 3 describes how the replicated log system that could form part of a final distributed application, called shared calendar, is built. The last section presents the conclusions drawn at the end of the thesis and the future improvements that could be made.
Chapter 1

Tools

This chapter discusses the developed tools and the approaches that have been adopted during the implementation of the distributed algorithms. The students could use these tools to make their own implementations of the distributed algorithms of the next chapter.

1.1 Communication System

The communication system is composed by a set of tools, that the distributed algorithms and some other tools use to communicate with other nodes. The communication system was designed to be able to send and receive messages at any moment. This system is prepared to work locally.

Described next are the components of the communication system.

1.1.1 Transport protocol

To make a simple implementation of the communication system, it was decided to use UDP (User Datagram Protocol) as the transport protocol.

With the UDP protocol it is not necessary to set up a connection between the sender and the receiver to transfer data. In Java, the DatagramSocket is used to send and receive messages via UDP. With one DatagramSocket a node can send and receive packets to and from multiple nodes simultaneously, and therefore it makes the implementation simple.
1.1.2 Message

The Message class (located at Implementations/ToolsAlgo/src/Com) represents a message that is going to be sent or that is received. This class has the following fields:

- **id** and **port**: The id and the port are related. The id is the integer that identifies a node, and the port is the sum of the identifier and the initial port that is set in the NetConf interface. In case that the message is going to be sent, the id is from the receiver; when the message is received is the sender’s id.

- **IPAddress**: The IP address of the node sender or receiver.

- **data**: The message data, that is an object (Data class) with the fields:
  - **BUFFER_ID**: identifier of the buffer, where the message is inserted when is received
  - **TAG**: tag that indicates the type of the message;
  - **CONTENT**: content of the message.

The message data is converted to a JSON string when it is sent, and once received, it is converted to the Java object Data.

JSON (JavaScript Object Notation) is a simple human-readable data format, being suitable to present data on the terminal. JSON can be built in two structures: As a collection of name/value pairs known as object; as an ordered list of values, that is an array. For the message data, the object structure is used (see the figure 1.1.1). The JSON strings are constructed using the GSON (Google JSON) java library [2]. With GSON is simple to convert a Java object to its JSON representation and the other way round. Later on, we will see that GSON is used for other applications.

```
{ BUFFER_ID:<string value>, TAG:<string value>, CONTENT:<string value> }
```

Figure 1.1.1: Message data JSON String Structure

The Message class provides constructors to create sending messages by passing the aforementioned fields as parameters. To create a receiving message, there is a constructor to obtain the fields from the received datagram. Also, it offers methods to get and set all the fields.

It is important to explain how the content can be obtained. The content can be got as a string, and then converted to the corresponding type, if it is necessary. And also, it
can be got directly to the original type, passing the class type as parameter by using GSON. This latter way to get the content is very useful, since allows to the user to create a complex data structure as a Java object, send it as message content as a string to an other node and once this message is received, and recover its content in a Java object, for example.

1.1.3 Buffers

The Buffers class (located at Implementations/ToolsAlgo/src/Com) has been implemented to separate the received messages of the algorithms that are simultaneously running on the node.

Every message that is sent has an identifier (BUFFER_ID), that indicates the buffer to which the receiver’s Stub worker will put the message.

Each algorithm can create as buffers as needed to receive packets.

The Buffers class consist on a concurrent HashMap that represents the buffers and offers a set of methods to manage the buffers.

The Buffers’ HashMap (see the figure 1.1.2) is a hash table that supports concurrent retrieves and updates, and it is formed by

- BufferID: The string that identifies a Buffer.
- Buffer: Buffer formed by a circular queue where the received messages are put.

![Figure 1.1.2: Buffers HashMap](image)

The Buffers class provides methods to:

- Add a new buffer.
- Put a message in a buffer.
• Get a message from a buffer.
• Clear a buffer.
• Remove a buffer.
• Get a buffer.

1.1.4 Configuration files

In this section we describe the files to configure the network settings and the message printing.

1.1.4.1 NetConf

In the NetConf interface (located at Implementations/ToolsAlgo/src/Com) there are the fields to configure the settings and parameters related to the network, which are:

• **BASE_PORT**: The port number used to obtain the port that corresponds to a node, by adding it to the identifier of the node, or subtracting it in order to obtain the identifier of the node from a port.

• **SERVER_PORT**: Port number of the server of identifiers.

• **NETWORK_DELAY**: Time that will take to a packet to be received. This field is used in the Stub (see section 1.1.5).

• **PACKET_LOSS**: Probability of a packet to be lost. Used by the Stub.

• **PACKET_SIZE**: Packet size of the UDP datagram packets. Used by the Stub.

• **LOCALHOST**: The localhost's IP address.

1.1.4.2 Show and Print Message

In the PRINT field of the Show interface (located at Implementations/ToolsAlgo/src/ConfFiles/Print) we can specify the buffers of messages of the algorithms in order to show their messages in the terminal. To make more convenient to set the buffers in the PRINT field, a package called Buffers was created to add to it an interface for every algorithm in which add all the algorithm’s buffers. Also, Show provides two methods that make even more convenient to add buffers.
1.1. COMMUNICATION SYSTEM

To print a message, the type of message (send/received) and the message are passed to the method print of the PrintMessage class (located at Implementations/ToolsAlgo/src/ConfFiles/Print). Then, invoking the print method the message is printed in the terminal, if the PRINT field contains the buffer identifier of the message or if there is no specified buffer in it.

1.1.5 Stub

The Stub (located at Implementations/ToolsAlgo/src/Com) is a Runnable class that has the logic to send and receive messages through the datagram socket. When a Stub instance is created receives the DatagramSocket and the Buffers objects in the constructor.

The run method implementation waits to receive datagrams (packets of data) from other nodes via the DatagramSocket. A DatagramPacket is constructed for receiving packets of a length that is specified in the field PACKET_SIZE in NetConf interface. When a datagram is received, the following process is executed:

1. A Message is constructed from the parameters of the DatagramPacket instance.

2. Creates an instance of a thread, called Worker by passing the message and the Buffers object, and starts the Worker thread.

3. The Worker first sleeps for an amount of time set in NETWORK_DELAY, in order to simulate the network delay.

4. The Worker calls the method to print of the PrintMessage class, in order to print the received message in the terminal.

5. The Worker puts the message in the corresponding buffer.

Note that the Worker class mentioned in the receiving process is necessary, because otherwise the Stub would be blocked in the case that the message’s corresponding buffer was full. And also, because it is used to simulate the network delay.

The Stub provides the send method that is called passing the message to send as a parameter. When the method is invoked, it is determined if the message is lost using the value of the PACKET_LOSS field in NetConf interface, in order to simulate the packet loss. Then it constructs a DatagramPacket including: the message’s data, the IP address, and the port. Finally it sends the datagram through the DatagramSocket, and prints the sent message in the terminal.
CHAPTER 1. TOOLS

1.1.6 Node

The Node class (located at Implementations/ToolsAlgo/src/Com) sets up all the necessary to send and receive messages to and from other nodes. This class is used by the algorithms that need to communicate with other instances of them in other nodes, and by some tools.

The Node class provides constructors to set up a node:

- requesting a new identifier.
- passing a specific identifier as a parameter and also a boolean to indicate if the passed identifier must be requested to the server of identifiers.
- requesting an specific identifier passing the identifier from the command line. The identifier is requested to the server of identifiers by default. This constructor is useful in case that we want to be able recover a node by passing its identifier as an argument from the terminal.
- passing a DatagramSocket directly.

When a Node’s instance is created using a constructor to request a new or a specific identifier, this process is followed:

1. Requests the identifier to the server of identifiers.
2. Creates an instance of DatagramSocket passing the port, that is the sum of the BASE_PORT in NetConf and the received identifier.
3. Creates an instance of Buffers.
4. Creates an instance of Stub passing the DatagramSocket and Buffers instances as parameters.
5. Executes the Stub Runnable.

When the identifier is not requested to the server of identifiers, the steps 2 to 5 are executed. The constructor to which is passed the DatagramSocket, just executes the steps 3 to 5.

Node is thread and when it starts first creates an instance of the Helper thread, which will respond the alive messages from the failure detector of the server of identifiers to indicate that the node is alive. Then waits for the updates from the server of identifiers of the alive nodes list and the total number of nodes.

The Node provides the public methods to
1.2 Server of Identifiers

The server of identifiers (ServerIDs class located at Implementations/ToolsAlgo/src/Server) assigns the identifiers to the nodes, and reports to the nodes: the total number of nodes, and the nodes that are alive. It is assumed that the server of identifiers is always available, and that the communication between the server and the nodes is reliable.

The server of identifiers maintains the following variables:
• id: Integer that is increased monotonically when a new identifier is assigned.

• assignedIDs: List of the identifiers that are currently assigned.

• vacantIDs: Queue of the vacant identifiers that have been assigned previously, that the server can reassign.

The list and the queue described previously are managed by the server (the main thread) and the failure detector.

The failure detector is a thread that its main function is to detect the nodes that have failed, so that the server can know which identifiers are available to be assigned. For this purpose the failure detector follows these steps:

1. Sends to the nodes of assignedIDs list an ALIVE_REQUEST message.

2. Waits an amount of time to receive the ALIVE_RESPONSE messages of the nodes.

3. Removes the identifiers of the nodes that have not responded the request from the assignedIDs list, as these nodes are considered failed.

4. Add the identifiers of the failed nodes into the vacantIDs queue.

5. Inform to the alive nodes which of them are alive by sending an ALIVE_NODES_RES message, that contains the assignedIDs list, to the nodes that are in assignedIDs.

The nodes can request to the server of identifiers

• a new identifier.

• a specific identifier that has been assigned previously and it is vacant. This is useful in case that a node recovers.

• the list of alive nodes.

• the total number of nodes.

Every time that the server assigns a new identifier, it reports to the nodes: the total number of nodes, and a list of the nodes that are alive.

It is important to note that the server only assigns a specific identifier, if it has been assigned previously and it is vacant. To assign a specific identifier to a node, the server of identifiers follows these steps:
1. Receives an SPECIFIC_ID_REQ message with an identifier $i$ from a node.

2. Checks if $i$ is in vacantIDs

   (a) if $i$ is vacant, then
      i. removes $i$ from vacantIDs.
      ii. sends an ID_RESPONSE message with an “OK” along with $i$.
      iii. adds $i$ to assignedIDs.
      iv. sends an ALIVE_NODES_RES message with the list of alive nodes (assignedIDs) to the nodes that are in the assignedIDs list.

   (b) if $i$ is not vacant, then sends an ID_RESPONSE message with a “KO” as content.

Some algorithm implementations may need to initiate the nodes at the same instant. The server has the option to assign the node identifiers at the same time, passing the number of nodes that will be initialized as an argument in the command line.

1.3 Timeout and Periodic Task

The following classes can be used by the algorithms to execute a task at the end of a period of time or periodically.

1.3.1 TOTask

TOTask class (located at Implementations/ToolsAlgo/src/Utils) used to set a timeout, that is a period of time that once it has elapsed, a task is triggered.

When it is instantiated we pass to the constructor the

- task that is going to be performed when the timeout expires.
- period of time that elapses before the task is triggered.

This class offers the methods to

- start and stop the timeout. If a timeout has been started and the start method is invoked; the timeout is interrupted and a new one is started.
- know if a the timeout expired.
1.3.2 PTask

PTask class (located at Implementations/ToolsAlgo/src/Utils) is used to set a task that will be executed periodically. An instance of this class is constructed by passing the same parameters as the TOTask plus the initial period that must elapse before executing the task for the first time.

This class provides the methods to start and stop the periodic task.

1.4 Log

The Log class (located at Implementations/ToolsAlgo/src/Utils/Log) is used to write and read entries to and from a log in JSON format by using the GSON library described before (see the section 1.1.2). Log is designed to create logs that are mainly handled in memory and human-readable.

To create and manage a log we must pass to the constructor the path and the name of the file to write/read.

Every entry that is going to be written to the log must be a subclass of LogEntry. And each subclass must implement the method to convert itself to its JSON representation.

The Log class offers methods to

- check if the log file exists.
- get all the log's entries, passing the subclass of LogEntry.
- write an entry to the log.
- write a collection of entries to the log.
- overwrite the log's content with a collection of entries.
- remove the first entry of the log.
- delete the log file.

1.5 Limiting the active nodes in an algorithm

In some algorithms or distributed applications could be useful to limit the number of active nodes. The active nodes, are the ones that participate actively in the algorithm.
The nodes that do not participate actively, are the passive nodes. The passive nodes only receive information, if it is required by the algorithm. To implement this, the algorithms use the ActiveSet class (located at Implementations/ToolsAlgo/src/ConfFiles).

In the ActiveSet, it can be manually set the number of nodes that will be active in the public field MEMBERS. For example, if MEMBERS is set to 3, then the nodes with the identifiers 0 to 2 will be active nodes. If MEMBERS is null, then that all the nodes are active. Also, ActiveSet provides to the algorithms the methods:

- isMember(int identifier): It returns true if the node with this identifier is a member of the active set.
- isLimited(): It returns true if the number of members is limited, that is, if not all the nodes are active.

1.6 Sharing the functionalities

There are distributed algorithms that require functionalities from other algorithms to perform their tasks, therefore the algorithms need to share their functionalities between them. A tool was not implemented for this purpose; instead a simple approach is followed.

First we create a main class where we write the logic to initiate the node and create an instance of each algorithm. In the creation of the instances, we pass the instance of one algorithm to the constructor of other/s, so that the latter/s can use the functionalities of the former. Also, it is necessary to pass the instance of Node to the constructor of the algorithms that need to send and/or receive messages. Finally, we add the methods to run the algorithms.

1.7 Reconciliation

During the development of a system called replicated log of the shared calendar (see section 3.2), it was implemented a reconciliation mechanism. The replicated log consist in a collection of nodes, called replicas, which write operations in the same order on their local log, so that each replica eventually has a log that is identical to that of the other replicas. So, it was noted that it was necessary to implement a reconciliation mechanism to put up-to-date the replicas that are new or have been recovered and have missed operations. For this purpose, it was determined to develop a mechanism
to be used together with any algorithm that can be used for replication, with which, in a reconciliation, is not necessary to exchange the entire log, and simple to implement.

After doing some research, it was decided that the most suitable is to develop an anti-entropy protocol that is a class of epidemic algorithm [12]. Assume that there is a collection of replicas that have a log with written ordered operations, called writes. With the anti-entropy protocol, two replicas catch up between them, where one of them sends writes to the other, which are unknown to the other. The implementation of the anti-entropy protocol is called reconciler.

Described below are the elements that form part of the reconciliation mechanism that are depicted in the figure 1.7.1.

![Figure 1.7.1: Elements of the reconciliation mechanism](image)

**1.7.1 Log Management**

The log manager (LogManager class, located at Implementations/ToolsAlgo/src/Reconciliation/Log) is the responsible to ensure that all the operations are written in order to the log. As we can see in the figure 1.7.1, the log manager is used by the replication algorithm and the reconciler. The replication algorithm adds new committed operations. The reconciler writes unknown writes, which are obtained in a reconciliation.

The Log class (section ) is used to create the log, where each operation is finally written. Each write contains these fields:
1.7. RECONCILIATION

- **SN**: The sequence number that indicates in which instance of the replication algorithm the operation was committed. The writes are ordered by their sequence numbers.

- **stamp**: It identifies the operation. It is composed of the identifier of the node that issued the operation, and the logical time in which the operation was issued.

- **operation**: The committed operation.

The log manager creates a temporary log called *unwritten*, that contains the operations that are just been committed and/or can not yet be written to the log, because their SN is not the next one that must be written. For example, if in the log we have three writes with the SNs 0, 1 and 2, then if a new operation is added an has as SN 4, then it cannot be written to the log, because is not 3, so this write is written to the *unwritten*.

The log manager offers the following methods to:

- get the last write.
- get the SN of the last write.
- add a new operation. It is invoked by the replication algorithm.
- write a set of ordered writes to the log. It is invoked by the the reconciler.
- get the writes from a SN to the end of the log.
- get the writes from a SN to another SN.
- get the last n written writes. n is the number of writes to get.
- get the last n unwritten and/or written writes.
- get a list of all the writes in the log.
- register threads that want to be notified about:
  - The last written write in the log
  - A new write that was unknown.

When the replication algorithm commits an operation, then it must be immediately written in stable storage, since the node may fail. Therefore, when the replication algorithm adds a new operation, first it is written to *unwritten*. Later, when the node gets unknown writes from a reconciliation, and *unwritten* contains the next write, then
the log manager writes to the log the operation/s of unwritten, and removes them from unwritten.

When the reconciler gets a set of writes, it writes them directly to the log, since these writes are the next to be written.

1.7.2 Reconciler

The reconciler (Reconciler class, located at Implementations/ToolsAlgo/src/Reconciliation) is the responsible to get the missing writes, and provide to other replicas the writes that they do not know. The reconciler is composed by a server and a client (Server class and Client class, located at Implementations/ToolsAlgo/src/Reconciliation). The idea of how to propagate the operations was obtained from [12].

The operation of the algorithm is the following:

1. A server $i$ periodically sends a MySN message that contains the $SN$ of the last write of the log, $lSN_i$, to a random alive node. The alive nodes are obtained by invoking the method to get a list of the alive nodes of the Node instance.

2. A client $j$ receives a MySN message from $i$ with $lSN_i$ that is greater than $lSN_j$. First, it waits some time, to get updated by its own. Then, if $lSN_j$ still being smaller than $lSN_i$, it sends a GET_WRITES message with $lSN_j$ to the server, starts a timeout $T$, and waits to receive the writes from the server $i$. When $T$ expires, the client executes the step 5.

3. The server $i$ receives a GET_WRITES message from $j$, then passes the $lSN_j$ to a thread, which gets the writes from $lSN_j$ (exclusive) to the end of the log, and sends them in WRITE messages, one by one, to $j$.

4. When the client receives a WRITE message from $i$, stores the write, and starts a timeout $T$ again.

5. When the client’s $T$ is expired, it is assumed that the server has sent all the writes or it has failed. Then, the client writes the stored writes to the log. First, it selects the adjacent writes in order to avoid gaps in case that the messages are not received in order, and the sender has failed before sending all the writes. For example, if $T$ is expired and the client received the writes with the SNs 3, 4, 6, 7, it selects the writes 3 and 4. Finally, it writes the selected writes to the log, and waits for MySN messages.
Chapter 2

Distributed Algorithms

Described in the following sections are the distributed algorithms that have been implemented using the tools described previously (chapter 1).

First of all, an issue to solve is presented. Then, the distributed algorithms that solve the issue are described.

Each algorithm is described theoretically and then its implementation. In the implementation part, it is described the organization of the algorithm, the operation and what is considered important to highlight.

2.1 Synchronization Algorithms

In a distributed system the time is used to know the order of events that occurred in it. The order of events is useful for concurrent access control to shared resources, distributed debugging, etc.

2.1.1 Physical time

The time is usually measured with physical clocks, but physical clocks tend to deviate. To solve this problem in a distributed system, algorithms to synchronize physical clocks were designed.

2.1.1.1 Christian

The Christian algorithm consist in the server that its clock is used as a reference because it is the most accurate, and the clients that request the time to the server to correct their clocks.
The algorithm process is depicted in the figure 2.1.1. First a client requests the server’s time at time $T_0$, and receives the server’s response containing the correct time $T_{server}$ at $T_1$. Then the client estimates the elapsed time since the server sent the response, assuming that the time that takes to send the request is the same that takes to receive the response, that is $(T_1 - T_0)/2$. Finally the client sets the new time that is $T_{server} + \text{elapsed time}$ [1].

![Figure 2.1.1: Christian synchronization](image)

2.1.1.1 Christian Implementation

In the Christian implementation (located at Implementations/ToolsAlgo/src/Synchronization/Christian), when the server initiates waits to receive the TIME_REQUEST messages from the clients. When the server receives a TIME_REQUEST message, it gets the current time in milliseconds, and then sends the TIME_RESPONSE message with the time to the client.

The client first generates a random integer that is into a specific range, which is added each time that the client’s thread requests the current time to the system, in order to simulate the clock’s deviation. Then the client stores the current time in milliseconds ($T_0$), sends the TIME_REQUEST message to the server, and waits to receive the TIME_RESPONSE of the server. When the client receives the server’s response, which contains the time of the server, the client stores the current time ($T_1$), and computes the corrected time with the three "times" that has been obtained.

The node that is going to be the server is fixed before running the algorithm.

2.1.1.2 Berkeley

Unlike the Christian algorithm the Berkeley considers that there is not only one node with a more accurate clock. Berkeley considers that the most correct clock time is the average of the clock times.
2.1. SYNCHRONIZATION ALGORITHMS

In the Berkeley algorithm [3] is the master, that is the elected coordinator, which ask to the other nodes, called the slaves, for their clock times. The master estimates the local clock times of the slaves, using the round-trip time (as the Christian algorithm), and calculates the differences between its clock and the clocks of the slaves. Then the master takes a fault-tolerant average to eliminate the faulty clocks, which are the clocks with a large deviation. First it selects the largest subset of clocks in which the difference between the clocks is not more than a determined amount. Then the master takes the average of the clocks from the selected subset, and sends the amount of time, which is the difference between the average and the previously calculated differences, by which the clocks of the slaves must be adjusted. The master also adjusts its clock.

2.1.1.2.1 Berkeley’s master Implementation

In the Berkeley implementation (located at Implementations/ToolsAlgo/src/Synchronization/Berkeley), the Berkeley’s master is the node with the biggest identifier, and it is assumed that is the last node to initiate, in order to get the total number of nodes, which it is used for the request of the clock times.

When the master starts, the first phase is to get a list of the clocks of the master and the slaves, where a Clock is an object with the node identifier and the difference with the master’s clock. The master begins sending the TIME_REQUEST messages to the slaves and storing the current time in a variable \( T_0 \), and waits to receive the TIME_RESPONSE messages of the slaves. For each TIME_RESPONSE received, the master gets the current time in milliseconds \( T_1 \); calculates difference between the clocks of the master and slave, taking in account the round-trip time \( RTT = T_1 - T_0 \); adds the node id and the difference in the clocks list. The master also adds its Clock in the list.

The second phase is to make the selection of clocks of the fault-tolerant average. The authors [3] do not describe the algorithm to select the clocks that belong to the largest subset, where the difference between them is not more than a determined amount, and also it should be noted that the number of clocks in the selected subset must be more than \( (N - 1)/2 \), where \( N \) is the total number of clocks, because otherwise most of the clocks would be faulty and thus the fault-tolerant average would not be meaningful.

The steps to select the clocks are the following:

1. Get a sorted list of the clock differences.

2. Try to find the clock differences that belong to a largest set. First the master stores in a variable \( m \) the total number of clock differences, which is the largest
possible number of clocks in a set. Then the master tries to find if there are clock
differences that can belong to a set where the number of members is \( m \). If there
are not enough clock differences that can form a set of \( m \), then \( m \) is decreased and
the process starts over. This process continues until the master finds the clock
differences that can belong to a set with more than \((N - 1)/2\) clock differences, or
in the case that are not found, this means that most of the clocks are considered
faulty, then the process finishes with no clock differences.

3. Get a list with one of the largest subsets of clock differences, or an empty list if
most of the clocks are faulty.

In the third phase the master takes the average of the clock differences of the largest
subset. Then it sends to each slave an OFFSET message with the difference between
the average and the calculated difference in the clocks list of the first phase, which
the slave will add to its clock time. The master also adjusts its clock using the same
method.

2.1.1.2.2 Berkeley’s slave Implementation

The slave, as the client of the Christian algorithm, to simulate the clock’s deviation,
it first generates a random number in a specific range that is added when the slave’s
thread requests the current time of the system. Then the slave waits to receive the
TIME_REQUEST and OFFSET messages from the master. When it receives the
TIME_REQUEST message it responds with a TIME_RESPONSE message with its
time. Then when the slave receives the OFFSET message, it adjusts its clock by adding
the offset to its current simulated time.

2.1.2 Logical time

Lamport pointed out that it is not necessary to know the exact time to know the order
of the events that occurred in a distributed system [4]. To order events, we just need to
know if an event happened before an other event. An event is an action that executed
by a process. An event could be internal, for example an execution of an instruction,
or a send/receive message event.

Happened before relation

The happened before relation is denoted by →, and it is defined by the following [4]:
(1) If an event $a$ occurred before an event $b$ in the same process, then $a \rightarrow b$.

(2) If $a$ is the event of sending a message $m$ and $b$ is the event of receiving $m$, then $a \rightarrow b$.

(3) If $a$, $b$, $c$ are events and $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Note that if two events $a$ and $b$ occur in different processes and there is not an exchange of messages between these processes, then $a \not\rightarrow b$ and $b \not\rightarrow a$, therefore these events are concurrent.

2.1.2.1 Logical clocks

Lamport developed the logical clocks [4] to capture the happened before relation ($\rightarrow$). A logical clock is a counter that is monotonically incremented. Each process maintains its own logical clock, $L$.

When an event $a$ occurs in a process $P_i$, the process assigns a logical clock value to the event, $L_i(a)$, known as timestamp. If we have two events $a$ and $b$, if $a \rightarrow b$ then $L(a) < L(b)$.

To capture the happened before relation, the processes update their logical clocks following these rules:

LC1: A process $P_i$ increments its logical clock before executing an event:

$$L_i := L_i + 1$$

LC2: (a) If a process $P_i$ sends a message $m$, then it includes the timestamp in $m$:

$$T := L_i$$

(b) On receiving the message $m$, the process $P_j$ sets to $L_j$ the greater value of $L_j$ and $T$:

$$L_j := \max(L_j, T)$$

In the figure 2.1.2 we can see that the events $d$ and $e$, $d \rightarrow e$ and $L_2(d) < L_1(e)$.

2.1.2.1.1 Logical clock Implementation

The LogicalClock class (located at Implementations/ToolsAlgo/src/Synchronization/LogicalClocks) has an integer field that is the logical clock value.

The class provides constructors to create an instance with the initial logical clock value to 0, or with an initial value passed as a parameter.

The LogicalClock provides the following methods:
2.1.2 Vector clocks

Looking at the figure 2.1.2 $L_1(a) < L_2(c)$ but $a \rightarrow c$, because $a$ and $c$ are concurrent events, therefore with logical clocks we cannot conclude that an event happened before another. To solve this issue Fidge, Mattern and Schmuck developed vector clocks independently [5]. A vector clock consist in an array of $N$ integers, where $N$ is the number of processes in the system.

Each process maintains its vector clock, and updates it following these rules:

VC1: Before executing an event, a process $p_i$ increments its index:
$$V_i[i] := V_i[i] + 1$$

VC2: (a) When a process $p_i$ sends a message $m$, then it includes the vector timestamp in $m$:
$$VT := V_i$$

(b) Upon receiving a message $m$, the process $P_i$ merges $V_i$ with the $VT$ of $m$ by taking the maximum for each index:
$$1 \leq k \leq N : V_i[k] := max(V_i[k], VT[k])$$

We can compare two vector timestamps $V_i$ and $V_j$ as follows:
2.2. MUTUAL EXCLUSION ALGORITHMS

- \( V_i \leq V_j \iff \forall k : V_i[k] \leq V_j[k] \)
- \( V_i < V_j \iff \forall k : V_i[k] \leq V_j[k] \land \exists x : V_i[x] < V_j[x] \)

Looking at the figure 2.1.3, just by comparing the vector time stamps \( V_3(b) \) and \( V_1(e) \) we can conclude that \( V_3(b) < V_1(e) \) then \( b \rightarrow e \). On the other hand, if we compare the vector timestamps \( V_1(a) \) and \( V_2(c) \) we see that they are concurrent, because neither \( V_1(a) \leq V_2(c) \) nor \( V_1(a) \geq V_2(c) \).

Figure 2.1.3: Vector clocks mechanism

2.1.2.2.1 Vector clock Implementation

In the VectorClock class (located at Implementations/ToolsAlgo/src/Synchronization/VectorClocks) vector clock is represented as an array of integers. To instantiate a VectorClock we need to pass the total number of processes to initialize the vector clock array and the id of the local process.

The methods that this class offers are the following:

- Methods that apply the rule VC1 for internal and send events
- The method to receive an event, which receives a vector timestamp as a string. First an array is created from the received string, and then the rules VC2 (b) and VC1 are applied.

2.2 Mutual Exclusion Algorithms

The mutual exclusion algorithms are used to coordinate the processes to give exclusive access to one process at time to a shared resource, in order to avoid interferences and
ensure consistency. We assume that each process executes a block of code called critical section (CS), in which it accesses to a shared resource.

The mutual exclusion algorithms must meet these requirements [1]:

- **Fairness:** The access to the CS must be granted in the order that the processes have requested it.
- **Liveness:** The access to the CS is eventually granted to each process.
- **Safety:** Just one process at at time can execute in the CS.

Next are described four algorithms for distributed mutual exclusion. In each implementation of these algorithms there is a class called Stub that extends an abstract class called Mutex, which extends Thread and has the abstract methods to exit and enter the CS that must be implemented. These two methods of each Stub are used by another thread to enter and exit the CS.

### 2.2.1 Centralized

The centralized algorithm consist simply in a process that is elected as a coordinator that coordinates the access of the other processes to a shared resource.

A process that wants to access to the CS, it sends a request to the coordinator, and waits for the coordinator’s response. When the coordinator receives the request, if an other process is in the CS, then the coordinator does not respond the request and adds it to a queue. If no other process is in the CS, then the coordinator responds granting access to the CS. When a process exits from the CS, it sends a message to the coordinator, which indicates that it has released the resource. When the coordinator receives the release message, if there are requests in the queue, extracts the oldest and grants the access to the CS to the corresponding process. If the queue is empty, the coordinator waits for new requests from the processes [6].

Assuming that no failures occur the algorithm satisfies the mutual exclusion requirements. The fairness requirement is met because the requests are processed in order. On the other hand, the coordinator eventually grants the access to each process to the CS, therefore satisfying the liveness. The safety is also met, because if there is a process in the CS, the coordinator does not grant access to other processes.
2.2. MUTUAL EXCLUSION ALGORITHMS

2.2.1 Centralized coordinator implementation

In the centralized coordinator implementation (located at Implementations/ToolsAlgo/src/MutEx/Centralized), the coordinator is a thread that keeps a boolean initially set to false, which indicates if a process is in the CS, called mutex. Also, it keeps a queue of the identifiers of the processes that are waiting to access to the CS. When the coordinator starts, it waits to receive messages from the processes:

- When the coordinator receives a REQUEST message from a process \( P_i \):
  - If \( \text{mutex} \) is false, then it sends a message GRANT to process \( P_i \).
  - If \( \text{mutex} \) is true, then it puts the \( P_i \)'s identifier into the queue.

- When the coordinator receives a RELEASE message:
  - If the queue is not empty, it gets and removes the oldest id in the queue and sends a GRANT message to the corresponding process.
  - If the queue is empty, sets \( \text{mutex} \) to false, in order to indicate that the resource is free.

2.2.1.2 Centralized Stub implementation

The Stub class (located at Implementations/ToolsAlgo/src/MutEx/Centralized) implements the methods to enter and exit from the CS. On initialization is set the Node instance, the buffer identifier, and the coordinator’s identifier. When the method to enter the CS is invoked. First the REQUEST message is sent to the coordinator, and then waits for the GRANT message. When the method to exit from the CS is invoked, the RELEASE message is sent to the coordinator.

2.2.2 Token Ring

A simple way to organize the mutual exclusion between \( N \) processes without needing a process to play the role of the coordinator, is to arrange the processes in a logical ring [6]. The processes are arranged in a ring, using their identifiers as it is shown in the figure 2.2.1. In order for a process to be able to enter to the CS, it must have what is called a token. There is only one token that circulates around the ring in a single direction, therefore each process only needs to know the identifier of the neighbor to contact it.
When a process $P_i$ gets the token, if it requires entering the CS, then holds the token until it exits from the CS. If $P_i$ does not require accessing the CS it passes the token to the process $P_{(i+1) \mod N}$.

Then the algorithm satisfies the safety requirement, because just the process that has the token can access to the CS. The token circulates by passing by each process, so the liveness requirement is also met, assuming that no failures occur. But fairness is not met, because for example, if the number of processes is 4 and the process $P_1$ requires to enter to the CS, but now $P_2$ has the token and later $P_0$ requires the token, then $P_0$ will acquire the token before $P_1$, and therefore $P_0$ will enter to the CS before $P_1$.

### 2.2.2.1 Token Ring Stub implementation

The Stub (located at Implementations/ToolsAlgo/src/MutEx/TokenRing) is a thread that executes the token ring algorithm, and offers to another thread the methods to enter and exit the CS.

On initialization, the stub sets a boolean called request to false. The request boolean is used to indicate if there is a request to enter to the CS.

When the stub starts, if it is the process 0, (the node identifier is 0), then puts the token to circulate in the ring by sending the token to the process 1. The token passing is performed with a TOKEN message. Note that the ring must be formed before the token begins to circulate, otherwise the token would be lost. For this purpose, we can initialize the server of identifiers with the option to assign the node identifiers at the same instant, in order to initialize the nodes at the same instant, and therefore the
ring is formed before the node 0 passes the token.

When a process \( P_i \) has started it enters in a loop and waits to receive the token. When \( P_i \) receives the token, if there is not a request to enter the CS (\( \text{request is false} \)), then it sends the token to the neighbor \( P_{i+1 \mod N} \). The following are described the steps when a thread requests to enter to the CS:

1. The method to enter to the CS is invoked. The thread requests to enter to CS by setting \( \text{request} \) to \( \text{true} \). Then, it waits until the stub thread wakes it.
2. The stub thread receives the token and there is a request. Then the stub wakes the other thread, allowing it to enter to the CS, and waits.
3. The thread invokes the method to exit the CS, then sets \( \text{request} \) to \( \text{false} \), and wakes the stub.
4. The stub wakes up and passes the token to the neighbor.

The number of nodes (\( N \)) is obtained invoking the method to get the total number of nodes from the Node's instance. In this way, we can add new members in the ring without restarting all the process.

### 2.2.3 Lamport

In the Lamport’s distributed mutual exclusion algorithm each process keeps an ordered queue of requests. The queue is ordered using logical clocks, but we cannot order concurrent events solely with logical clock timestamps (see the section 2.1.2.2), then an extended version of the happened-before relation \( \Rightarrow \), for total ordering, is defined: if \( a \) is an event in \( P_i \) and \( b \) is an event in \( P_j \), then \( a \Rightarrow b \) if and only if either \( L_i(a) < L_j(b) \) or \( L_i(a) = L_j(b) \) and \( i < j \). It is assumed that the number of the process of the system is fixed, and the messages are received in the order that they are sent (FIFO order) [4].

The processes follow the following rules:

**L1:** To enter in the CS, process \( P_i \) sends a request message with timestamp \( T_i \) and its identifier \( i \).

**L2:** When process \( P_j \) receives the request from \( P_i \), it puts the \( P_i \)'s request to its request queue, and it sends a timestamped acknowledgment to \( P_i \).
L3: When $P_i$ exits from the CS removes its request from the request queue, and sends a release message to the other processes.

L4: When $P_j$ receives a release message from $P_i$, removes the $P_i$'s request from its request queue.

L5: $P_i$ enters in the CS when both of these conditions are met:

(a) $P_i$'s request is at the top of the request queue (the requests are ordered by $\Rightarrow$)

(b) $P_i$ has received an acknowledgment from every other process with a timestamp greater than the timestamp of its request.

The Lamport’s algorithm satisfies the three mutual exclusion requirements. The fairness is satisfied from the fact that the requests are ordered by the relation $\Rightarrow$. The rule L2 ensures that the condition (b) of rule L5 is met. The rules L3 and L4 imply that the condition (a) of rule L5 is eventually met. Therefore the Liveness requirement is also satisfied. The Safety is met because giving the rules L3 and L4, if a process $P_i$ is in the CS, the condition (a) of the rule L5 cannot be met in any other process until $P_i$ exits the CS.

2.2.3.1 Lamport Stub implementation

The Stub (located at Implementations/ToolsAlgo/src/MutEx/Lamport) maintains:

- l: A logical clock that is an instance of LogicalClock (see 2.1.2.1)
- requestQueue: The queue of requests, which is a PriorityQueue that orders its elements of type Request.
- myRequest: The request of the node to enter in the CS.
- N: The number of nodes. It is assumed that the server of identifiers option to assign the node identifiers at the same time is used.
- nACKs: The number of ACKs received.

The Request class is a data structure that represents a request, which has as fields the identifier and the timestamp. In order to be ordered this class implements the interface Comparable and therefore it implements the method compareTo, which the PriorityQueue uses to compare this request with other requests. As it is expressed in
2.2. MUTUAL EXCLUSION ALGORITHMS

⇒ Definition: first, the time stamps are compared, then if time stamps are equal the identifiers are compared.

It is assumed that an event is sending or receiving a message. Therefore, before a message is sent the 1 value is incremented by invoking the corresponding method, and for every received message the corresponding 1’s method is invoked passing the received timestamp.

The algorithm is executed as follows:

- Request the CS
  - When the thread invokes the method to enter in the CS:
    1. sends message with the timestamp in its content to the other nodes.
       The identifier is not added in the content because it can be obtained in the destination.
    2. sets to myRequest a new Request passing the id and the time stamp.
    3. adds myRequest to the requestQueue.
    4. waits until it is wakened by the stub, to enter in the CS.
  - When the stub receives the REQUEST message:
    1. adds a new Request passing the sender’s id (got form the message) and the received timestamp to its requestQueue.
    2. sends an ACK (acknowledgment) message with the timestamp to the sender of the request.

- Exit the CS
  - When the thread invokes the method to exit the CS:
    1. removes the request at the top of the requestQueue.
    2. sends a timestamped RELEASE message to the other nodes.
  - When the stub receives a RELEASE message:
    1. removes the request at the top of the requestQueue.

- Enter in the CS
  - When the stub receives an ACK message:
    1. compares the received timestamp rT with the myRequest timestamp mT, and if rT is greater than mT it increments nACKs.
- For every received ACK and RELEASE message:
  1. it checks if it has received \( N - 1 \) ACK and if myRequest is at the top of requestQueue.
  2. if so, it wakes the thread to let it enter to the CS and sets nACKs to 0.

### 2.2.4 Ricart and Agrawala

Ricart and Agrawala developed an algorithm that improves the Lamport algorithm requiring less messages [7]. The idea of this algorithm is that a process can only access the CS, if it has received a reply to its request from all other processes. In this algorithm, when the processes receive the requests, they take decisions based on the relation \( \Rightarrow \) (defined in the previous section), to totally order the access of the processes in the CS.

Each process keeps:

- A sequence number that can be considered as a logical clock. Then it is managed applying the logical clock’s rules (section 2.1.2.1), when a message is sent or received.
- A queue of the identifiers of the processes, whose replies are deferred.
- A boolean called requestingCS, which when is true the process is requesting or executing the CS. Initially it is set to false.

The algorithm works as follows:

- Requesting the CS
  - When a process \( P_i \) decides to enter in the CS, sets requestingCS to true, sends a request to all the other processes, and waits to receive the replies.
- Taking decisions to totally order
  - When a process \( P_i \) receives a request from another process \( P_j \):
    1. if \( P_i \)'s requestingCS is false, \( P_i \) replies to \( P_j \).
    2. if \( P_i \)'s requestingCS is true, following the relation \( \Rightarrow \): if the request’s timestamp of \( P_j \) is greater than the request’s timestamp of \( P_i \), or if the timestamps are equal and \( P_j \)'s identifier is greater than \( P_i \)'s identifier, then the \( P_j \)'s reply is deferred and its identifier is put in the queue. Otherwise, \( P_i \) replies to \( P_j \).
2.2. MUTUAL EXCLUSION ALGORITHMS

- Entering the CS
  
  - When all the other nodes have replied to process \( P_i \), then \( P_i \) enters the CS.

- Exiting the CS
  
  - When \( P_i \) exits the CS, sets requestingCS to false and it replies to all the processes of the queue.

The mutual exclusion requirements are satisfied. When a process \( P_i \) receives a message from \( P_j \), decides if the one that enters to the CS is itself or \( P_j \). Since this decision is taken following the definition of the relation \( \Rightarrow \), in the end the requests are totally ordered, so fairness is met. Liveness is satisfied because every process that exits the CS replies all the deferred requests, and therefore every node that has requested to enter the CS, eventually receives the replies from all other nodes and enters the CS. Giving two process \( P_i \) and \( P_j \) as \( i \neq j \), then to enter the CS at the same time, the processes would have to have replied each other and this is impossible, because the processes take decisions based on \( \Rightarrow \) relation. Therefore, Safety is also met.

2.2.4.1 Ricart and Agrawala Stub implementation

The Stub implementation of the Ricart and Agrawala (located at Implementations/ToolsAlgo/src/MutEx/RicartAgrawala) is organized as the Lamport’s. Ricart and Agrawala’s implementation is more simple, since it uses less message types and data structures.

The Stub maintains the following:

- \( l \): A logical clock that is an instance of LogicalClock.

- \( N \): The number of nodes. It is assumed that the server of identifiers option to assign the node identifiers at the same time is used.

- \( n\text{REPLYs} \): The number of \text{REPLY} messages received.

- \( \text{requestingCS} \), mentioned before.

- \( \text{deferredQueue} \): The queue of the identifiers of the nodes, whose replies are deferred.

- \( \text{myRequestT} \): The request’s timestamp of the node.
Unlike Lamport's in this algorithm is not necessary to keep Request objects. And only two types of messages are used: REQUEST and REPLY. The REPLY message is an equivalent to the combination of Lamport's ACK and RELEASE, because the nodes cannot enter the CS, until they have not received $N - 1$ REPLY messages (ACK messages). When a node exits the CS, sends the deferred REPLY messages to the nodes of the queue, therefore it lets to other nodes to enter in the CS, which is the same function of the RELEASE message.

The main difference regarding to Lamport's implementation is on receiving a REQUEST message. When a REQUEST message is received, the algorithm's description is followed. If requestingCS is true and the requesting node has less priority to access the CS, then the REPLY message to the requesting node is delayed, so its id is added to the deferredQueue. Otherwise, if the requesting node has priority or the requestingCS is false, the REPLY message is sent to the requesting node.

2.3 Election Algorithms

In a distributed system an election algorithm is used to elect an unique process to play a particular role. Usually, this role is to be the coordinator, for example to choose the process that would be the coordinator in the centralized mutual exclusion algorithm (section 2.2.1), we need an election algorithm. It is assumed that every process has a unique identifier, and the identifiers processes are ordered. It is also assumed that the elected process is the one with the largest identifier.

The requirements for an election algorithm are the following [1]:

- Safety: Only one process can be elected as a coordinator.
- Liveness: All the processes participate and eventually must have elected a coordinator.

Any process can initiate an election. A process can only participate in one election at time, and can be participant if it is involved in an election, or non-participant if it is not involved in an election [1].

The next election algorithm implementations implement the Election interface, which it extends the Runnable class. The Election interface has the methods to get the coordinator, and to get the new coordinator, which must be implemented by the algorithm implementations. These implemented methods are provided to other algorithms that
need to elect an unique node to perform a particular task. The implementation of Election’s methods work in the same way in each implementation:

- **getCoordinator():** While no coordinator has been elected, each thread waits until the coordinator is elected. If the coordinator has been elected, then it returns the identifier of the node that is the coordinator.

- **getNewCoordinator(int previous):** The previous integer is the identifier of the previous coordinator. While no coordinator has been elected or the current coordinator is the same as the previous one, each thread waits until a new coordinator is elected. If a new coordinator has been elected and is different than the previous one, then it returns the identifier of the node that is the coordinator.

### 2.3.1 Chang and Roberts Ring

In the Chang and Roberts Ring algorithm the processes are arranged in a ring, where the messages circulate clockwise [1]. Each process $P_i$ communicates with its neighbor $P_{(i+1) \mod N}$. It is assumed that no failures occur.

Initially all the processes are *non-participant*. When a process begins an election, marks itself as *participant*, and sends an election message with its identifier to the neighbor.

When a process receives an election message compares the message’s identifier with its own identifier: if the message’s identifier is larger than its own, it forwards the received message to the neighbor and marks itself as a *participant*; if the message’s identifier is smaller than its own and it is *non-participant*, then the process discards the received message and sends an election message with its identifier to the neighbor, and marks itself as a *participant*; but if the process is *participant*, then it just discards the received message.

Eventually, a process will receive an election message with its identifier. This means that this process has the largest identifier, therefore declares that it is the *coordinator*: marks itself as a *non-participant* and sends an elected message with its identifier to the neighbor. When a process receives an elected message, if the message’s identifier is not its identifier, it marks itself as *non-participant*, accepts that the elected is the *coordinator* and forwards the elected message. When the *coordinator* receives the elected message the election finishes.

Note that by marking the processes as *participant* when they send a message, ensures that the processes are only involved in one election.
The safety requirement is satisfied. Every identifier in an election message is compared by all the processes in the ring, and a process must receive an election message with its identifier to send an elected message. Therefore, it is impossible for any process other than the process with the largest identifier, to receive its identifier in an election message. The liveness is met as there are no failures.

2.3.1.1 Ring implementation

The Ring class (located at Implementations/ToolsAlgo/src/Election/Ring) implements Election and therefore provides the Election's methods, to other algorithms. The Ring maintains the following:

- **myId**: Identifier of the node.
- **participant**: Boolean, which is set to true when the node is participant. Initially is set to false (non-participant).
- **elected**: Keeps the identifier of the elected node. Initially is null, because no node was elected.

When the Ring starts, first it begins an election, so sets the participant to true, and sends an ELECTION message to the neighbour. Then the Ring waits to receive ELECTION and ELECTED messages.

When an ELECTION or an ELECTED message is received, in the logic of the implementation has been followed the description of the algorithm. Once the coordinator has been elected, that is when the identifier of the coordinator is set in elected, the Ring wakes up all the threads of other algorithms that are waiting to get the coordinator.

Every time that a message is sent to the neighbor, when the identifier of the neighbor is computed, the number of nodes \( N \) is obtained from the Node's instance. In this way, new nodes can be added to the ring.

2.3.2 Bully

In the Bully algorithm the process with the largest identifier is the coordinator[1]. The name Bully comes from the fact that for example, when a new process enters in the system with a larger identifier than the current coordinator, then this new process proclaims itself as coordinator without initiating an election. It is assumed that: each process knows the identifier of the other processes; the message delivery is reliable; it
allows process failures. The process failures are detected using timeouts, that is, if a process doesn’t respond a message in a determined amount of time, it is assumed that it has failed. In the figure 2.3.1 are depicted a sequence of actions that show the operation of the algorithm.

If a process has the largest identifier, then sends a coordinator message to all the other processes, proclaiming that is the coordinator (figure 2.3.1a).

A process that does not have the largest identifier, begins an election when: enters in the system; recovers from a failure; detects that the current coordinator has failed. To begin an election, a process sends an election message to the processes with a larger identifier than its own, changes its state to participant, and start a timeout TO1 and waits for answer messages (P2 of figure 2.3.1b). If TO1 expires, that is, the process has not received any answer message, then becomes the coordinator, so it sends a coordinator message to the processes with a smaller identifier, and changes its state to non-participant (P2 of figure 2.3.1e). Otherwise, if the process receives an answer message, then starts a timeout TO2 and waits for a coordinator message (P2 of figure 2.3.1c). If TO2 expires begins a new election (P2 of figure 2.3.1d).

When a process receives an election message, responds with an answer message and if it is non-participant, then begins an election.

When a process receives a coordinator message, accepts that the sender is elected as the coordinator.

The Liveness requirement is met, because all the processes will receive a coordinator message, as the message delivery is reliable. The safety is also met, because two processes could not determine that they are the coordinator at the same time, as one of these processes would notice that there is another process with a larger identifier than its own identifier.

2.3.2.1 Bully implementation

In the Bully algorithm’s description is not specified how to detect the failure of the coordinator. In the implementation of Bully (located at Implementations/ToolsAlgo/src/Election/Bully) in order to detect the coordinator’s failure, once the coordinator has been elected, then each node that is not the coordinator, sends an ALIVE message periodically to the coordinator and starts a timeout for each message. When the coordinator receive an ALIVE message, sends back an ALIVE_ANSWER message. If a node does not receive an ALIVE_ANSWER in the timeout period, then begins an election.

A Bully process keeps the following:
(a) $P_3$ has the largest identifier, so it sends a coordinator message and becomes the coordinator.

(b) $P_2$ recovers and begins an election.

(c) $P_3$ sends an answer message to $P_2$.

(d) $P_3$ fails and then TO2 of $P_2$ expires, so it begins an election.

(e) TO1 of $P_2$ expires, so it sends a coordinator message, and becomes the coordinator.

Figure 2.3.1: Bully algorithm. Processes in color gray have failed.

- **myId**: The id of the node.

- **participant**: Boolean that is true when the process is participating in an election. Initially is set to false.

- **elected**: Keeps the identifier of the elected process. Initially is null, because no process was elected.

- **electionTO**: Timeout that expires, if no ELECTION\_ANSWER or COORDINATOR message is received in a determined period of time.

- **answerTO**: Timeout that expires, if no COORDINATOR message is received in a determined period of time.

- **alivePTask**: Periodic task that checks if the coordinator is alive.

- **aliveTO**: Timeout that expires, if no ALIVE\_ANSWER message is received in a determined period of time.
2.3. ELECTION ALGORITHMS

The period of time of the timeouts is set with the NETWORK_DELAY of NetConf, the number messages to send and receive, and also the time that is considered enough to process each message: \( T = T_{\text{NETWORK}} \times 2 + T_{\text{process}} \)

In the figure 2.3.2 we can see a state machine that models the operation of the implementation. This state machine shows the states and transitions that happen during an election. An election starts in the Beginning state, and ends in the Coordinator elected state. The Beginning state is before than an election begins, that is, before the process sends an ELECTION or COORDINATOR messages. The Coordinator elected state is when the coordinator has been elected. We have a tuple \([\text{participant}, \text{elected}, \text{state}]\) that is updated in each transition, which contains the variables that are updated during an election. Initially the tuple's values are \([\text{false}, \text{null}, \text{beginning}]\). Below are described the transitions to the states. Note that every time that the coordinator has been elected, the process wakes up all the threads of other algorithms that invoked the Election's methods. Also, every time that an election begins, elected is set to null, as the previous coordinator is no longer valid.

![Figure 2.3.2: State machine that shows the states and transitions that happen during a Bully election](image)

C1: The process gets the total number of processes and checks that it has the largest identifier, so it sets participant to false, sets elected to myId, and sends a
COORDINATOR message to the processes with a smaller identifier. The tuple is updated to [false, myId, coordinator_elected].

WE: The process gets the total number of processes and checks that it does not have the largest identifier, so it sets participant to true, sends an ELECTION message to the nodes with a larger identifier, and starts a electionTO. The tuple is updated to [true, null, waiting_ELECTION_ANSWER].

TO1: electionTO is expired, then a task is triggered, which sets participant to false, sets elected to myId, and sends a COORDINATOR message to the processes with a smaller identifier. The tuple is updated to [false, myId, coordinator_elected].

WC: The process receives an ELECTION_ANSWER message. Then the process checks if it is participant, because it may have received a COORDINATOR message before receiving the ELECTION_ANSWER, and therefore participant would be false and then nothing is executed. If participant is true, then the process stops the electionTO, and starts answerTO. The tuple is updated to [true, null, waitingCOORDINATOR].

C: A COORDINATOR message is received from a process $P_j$, so the process stops all the timeouts, sets participant to false, sets elected to the $P_j$’s identifier ($j$), and starts alivePTask. The tuple is updated to [false, $j$, coordinator_elected].

TO2: answerTO is expired, then the triggered task sets participant to false, and begins a new election. The tuple is updated to [false, null, beginning].

TO3: aliveTO is expired, then the task begins an election. The tuple is updated to [false, myId, beginning].

B: The process receives an ELECTION message, so it sends back an ELECTION_ANSWER, and begins an election. The tuple is updated to [false, myId or $j$, beginning].

It is important to mention that few modifications have been made in the implementation of Bully, to use the ActiveSet class (see 1.5), in order to have the option to limit the number of nodes that can participate in an election, and become the coordinator. The nodes that are going to participate in an election are members of the active set, and only one of them can be the coordinator. The nodes that are not members just wait until the election is finished, to finally know who is the elected coordinator. The main reason why the ActiveSet has been added, is because later Bully will be used along with the Paxos algorithm, and in the Paxos implementation that has been made, it is convenient to limit the number of active nodes.
2.4 Consensus Algorithms

Consensus consists in a collection of processes that propose values. Then, the processes agree to choose a single value among the proposed values. Consensus algorithms can be used for mutual exclusion, to agree in which process will access to a resource, also for election to elect a node to be the coordinator. But, as we have seen there are dedicated algorithms for these tasks. Consensus algorithms are mainly used for replication, where a collection of replicas agree in the next operation that they will execute, so the replicas execute the same operations in the same order, thereby ensuring that eventually they will reach the same state.

A consensus algorithms must met the following requirements [11]:

- **Safety:**
  - Only a single value must be chosen.
  - Only a proposed value must be chosen.
  - A process never learns a value that has not been chosen.

- **Liveness:** A proposed value is eventually chosen, and if a value is chosen, a process eventually learns it.

Next, it is described the Paxos consensus algorithm, which has been widely used for replication in real systems [8, 9, 10].

2.4.1 Paxos

Paxos is a fault-tolerant consensus algorithm where the processes play the roles: proposer, acceptor, learner. A process may play one or more of these roles. In Paxos we assume that:

- processes may fail and recover.
- messages can be lost and may be delivered with delay
- messages are not corrupted.

The idea of Paxos is that a proposer proposes a value $v$ from a client along with a proposal number $n$ to a set of acceptors; then the acceptors decide if they accept proposal. If a majority of the acceptors accept the proposal with the value $v$, then
v is chosen. When an acceptor accepts a proposal notifies to the learners that they
must learn the accepted value v. Finally, when a learner receives the response from a
majority of acceptors, then sends a response to the client that issued the value v.

The proposal number is used to distinguish the proposals of the proposers. Each
proposal number is unique and is increased monotonically. Also these proposal numbers
must be totally ordered.

The operation of the algorithm for a consensus instance is the following [11]:

**Phase 1**
(a) A proposer increments its proposal number, and then sends a PREPARE
message with the proposal number n to a set of acceptors.

(b) When an acceptor receives a PREPARE message with a proposal num-
ber n that is greater than the proposal number of any PREPARE message
that it has responded, then an acceptor responds with a PROMISE mes-
gee that indicates that it will not accept proposals with a proposal number
less than n, and the proposal with the highest proposal number that it has
accepted (if any).

**Phase 2**
(a) When a proposer receives a PROMISE message from a majority of the
acceptors, then it sends an ACCEPT message with a proposal, which is
composed with the proposal number n used in phase 1 and a value v, where
v is the value of the proposal with the greatest proposal number among the
PROMISE messages, or a new value selected by the proposer.

(b) When an acceptor receives an ACCEPT message with a proposal number
n, accepts it if it has not acknowledged a proposal number of a PREPARE
message greater than n. Finally, the acceptor sends a LEARN message with
the accepted proposal to all the learners.

(c) When a learner receives a LEARN message with a proposal from a
majority of the acceptors, then it learns the proposal’s value v, and responds
to the client that issued v.

Note that, since the processes may fail, each acceptor must keep in stable storage, at
least, the highest acknowledged proposal number of a PREPARE message, and the
proposal with the highest proposal number accepted.

We can apply the next performance optimization. When an acceptor receives a PRE-
PARE or ACCEPT message with a proposal number less than the highest proposal
number stored, instead of ignoring the message, the acceptor may send a REJECT message with the current proposal number in order to inform that the proposer may abandon its proposal. When the proposer receives the REJECT message, it increments its proposal number and begins the phase 1.

**Progress**

Imagine an scenario where two proposers $P_0$ and $P_1$ are active. First $P_0$ completes the phase 1 with a proposal number $n$, later $P_1$ completes the phase 1 with a proposal number $n + 1$, so in the phase 2 $P_0$'s proposal is rejected because $n$ is less than $n + 1$. Later, $P_0$ completes the phase 1 with $n + 2$, so $P_1$'s proposal is rejected as $n + 1$ is less than $n + 2$, and so forth. To avoid the previous situation, and ensure progress, that is, when the system passes by all the phases and terminates, it is necessary to elect a proposer to be the coordinator. The proposer that is the coordinator will issue the proposals and the other nodes will remain passive, prepared to relieve the coordinator, in case that it fails.

**Examples**

The figure 2.4.1 depicts the best case scenario, where value has been chosen and no failures have occurred. In this scenario there are three nodes $N_0$, $N_1$, $N_2$, where each node plays the three roles: proposer, $P$; acceptor, $A$; learner, $L$. For example $N_0$ internally has $P_0$, $A_0$, $L_0$, and therefore when a node $N_i$ fails its $P_i$, $A_i$, $L_i$ also fail. A proposer $P_i$ uses the learner $L_i$ to know if a value is learned. A client $C$ sends values to the proposer that is the coordinator, which will be the one with the largest identifier that is alive. Note that, since the total number of nodes is 3, then a majority is more than $3/2$. This scenario is easy to follow by reading the description of the algorithm’s operation as there are no failures.

In the figure 2.4.2 we can see a worst case scenario. First, $C$ sends a new value $v$ to the coordinator $P_1$, because $P_2$ failed. Then $P_1$ increments its proposal number, $n + 1$, then sends it in a PREPARE message to the acceptors. The phase 1 is completed, but in the phase 2 the acceptors $A_0$ and $A_1$ fail just after they have accepted $v$ and sent a LEARN message to the learners. This means that $N_0$ and $N_1$ have failed, so $L_0$ and $L_1$ have also failed and they have not processed the LEARN messages. Therefore, $v$ is chosen, but not learned. Then, $N_2$ ($P_2$, $A_2$, $L_2$) recovers, increments its proposal number, $n + 1$, and sends a PREPARE message to the acceptors. $P_2$ does not receive a majority of PROMISE messages as only $A_2$ is alive, so it increments its proposal number, $n + 2$,
and begins the phase 1. Now, $P_2$ receives a majority of the PROMISE messages, as $N_0(P_0, A_0, L_0)$ has recovered. $A_0$'s PROMISE message contains the chosen value $v$, so $P_2$ sends an ACCEPT message with $n + 2$ and $v$ to the acceptors. Once $A_0$ and $A_2$ receive the ACCEPT message, they send a LEARN message with the $P_2$'s proposal to the learners. Finally, $L_0$ and $L_2$ learn $v$, and then they send a RESPONSE message with $v$ to $C$.

**Consensus requirements**

The liveness requirement is satisfied, since a proposer is elected as a coordinator in order to ensure progress (see the section 2.4.1).

Once a value is chosen for an instance of the consensus algorithm, this value is associated permanently to this consensus instance, because if a proposer tries to propose a new value for this instance, it will always receive the chosen value in a PROMISE message. Therefore, we can see that safety is met.

2.4.1.1 **Paxos implementation**

In the description of the Paxos algorithm is described how a single value is chosen for an instance of the consensus algorithm. Since more than one value will be proposed
2.4. CONSENSUS ALGORITHMS

Figure 2.4.2: Paxos instance execution. Example of a worst case scenario

and each one should be eventually chosen, then we need to execute different consensus instances. In the implementation (located at Implementations/ToolsAlgo/src/Consensus/Paxos) each consensus instance is identified by an integer \( i \), the instance number, that is monotonically increased. When a chosen value for an instance is learned, we say that the instance is closed.

Each node internally has a proposer, an acceptor and a learner. Each role is a thread, and has the corresponding Buffer to receive to messages that are addressed to it.

The Bully algorithm (see section 2.3.2) is used to elect the proposer that will be the coordinator. It is the most suitable, because it detects the node failures.

The Paxos implementation uses the reconciliation mechanism (see section 1.7) to put up-to-date the nodes that failed and missed chosen values. Later, in the learner imple-
The reconciliation mechanism is not strictly necessary, since just following the description of the Paxos algorithm, the implementation works correctly without it. But, later we will see that Paxos is used in the shared calendar application (see chapter 3). So, this mechanism was added during the development of the shared calendar, because it was necessary.

The nodes that are the acceptors are fixed, because it is necessary to know which are the acceptors that are going to participate. Otherwise, if the acceptors are not fixed, a set of nodes could choose a value $v$ for the instance $i$, then another completely different set of nodes could choose a value $v'$ for the instance $i$. Therefore we can see that there is an inconsistency, since two values have been chosen for the same instance. To fix the nodes that are acceptors, we set the number of acceptors in the ACCEPTORS field of the interface Conf. In this way, the proposer sends the messages addressed to the acceptors to the nodes with an identifier from 0 to ACCEPTORS − 1. We also decided that only the nodes that are acceptors can be elected as a coordinator in a Bully election. This later decision was made taking in account that it is assumed that the nodes that are acceptors are the most stable, moreover, since if there is not a majority of alive nodes that are acceptors, then no value can be chosen, therefore it is not necessary that other nodes that are not acceptors to be able to be the coordinator. This decision has been applied using the ActiveSet class (see 1.5). So we set the MEMBERS field in ActiveSet to the number of nodes that will be able to be elected as a coordinator, and also we assigned MEMBERS to ACCEPTORS, thereby fixing the nodes that will be acceptors. Therefore, the nodes that will be active will be the nodes with an identifier from 0 to MEMBERS − 1; the rest of the nodes will be passive, and only will receive LEARN messages with the values to be learned.

The following assumptions are made:

- The number of nodes is dynamic.
- The active nodes are alive long enough to choose a value.
- For practical reasons, the message delivery is reliable.

### 2.4.1.1.1 Data structures

The following are described the main implemented data structures.

**Value**

This class represents a value. Has the following fields:
2.4. CONSENSUS ALGORITHMS

- **clientID**: Identifier of the node that issued the value.
- **time**: Logical time at which the value was issued.
- **operation**: Operation to be executed.

The `clientID` along with `time` are used to identify the value. The method equals is implemented in order to check if the `clientID` and `time` are equal to the other’s ones.

**ProposalNumber**

It represents a proposal number. To make each proposal number unique and to be able to be totally ordered (from larger to smaller) with other proposal numbers, each proposal number has the fields:

- **l**: Logical clock that is incremented each time that the proposer prepares to propose a value.
- **id**: Identifier of the proposer.

It provides the methods to:

- increment the l’s value.
- check if a proposal number is greater than another one.
- make a proposal number equivalent to another one. When a REJECT message is received, then the proposer increments l before beginning the phase 1 in order that its proposal be accepted. Instead of just incrementing, the proposer uses this method before, to set its l’s value to the l’s value of the other proposal number, so it saves possible future increments.
- check if the proposal number is equal to another one. it returns true if l’s value and id are equal to the other’s ones.

**Proposal**

This class represents a proposal. It has the fields:

- **i**: The instance number.
- **n**: The proposal number.
• \( v \): The value.

It offers the methods to:

• increment \( n \). It uses the \( n \)'s method to increment. For example, if the proposer's id is 0 then if \( n \) is incremented for the first time, \( n \)'s \( l \) is incremented so (1,0).

• prepare a new proposal. It sets \( v \) to null, increments \( n \), and gets the next opened instance, so it gets the instance number of the last closed instance from the learner and increments it.

• set the value of \( v \).

• make \( n \) equivalent to another \( n \), if the other \( n \) is greater than \( n \).

• check if the proposal is equal to another one, by comparing their \( i \) and \( n \).

2.4.1.1.2 Learner implementation

The learner (located at Implementations/ToolsAlgo/src/Consensus/Paxos) maintains the following variables:

• \texttt{learned}: Array of learned values.

• \texttt{closed}: Array of closed instances.

• \texttt{cp}: capacity of the \texttt{learned} and \texttt{closed} arrays.

• \texttt{lastClosedInstance}: The last closed instance.

• \texttt{lm}: Instance of the log manager of the reconciliation mechanism (see section 1.7).

When the learner learns a value for an instance, uses the log manager to add the instance number and the value's fields to a log file.

When the learner initiates, it starts a thread called \texttt{Updater}, which gets \texttt{cp} closed instances and its associated values from the log using the log manager, adds them to \texttt{learned} and \texttt{closed} arrays, and sets the \texttt{lastClosedInstance}.

When the learner receives a LEARN message, which contains a Proposal with and instance number \( i \), a proposal number \( n \), a value \( v \)
2.4. CONSENSUS ALGORITHMS

1. Checks if \( i \) is closed.

2. Stores \( n \) along with a counter for \( i \), if it does not exits. The counter keeps the number of times that \( n \) has been received in a LEARN message.

3. Increments the \( n \)'s counter.

4. If it has received the proposals from a majority of acceptors, that is, if the counter of \( n \) is greater than \( \text{ACCEPTORS}/2 \), then:
   
   (a) Remov es all the \( i \)'s proposal numbers received and their counters.
   
   (b) Sets \text{lastClosedInstance} to \( i \).
   
   (c) Uses the log manager to add \( i \), the clientID, time and operation of \( v \) to the log file.
   
   (d) Adds \( i \) to closed and \( v \) to learned.
   
   (e) Sends a RESPONSE message with \( v \) to the node with the identifier clientID of \( v \).
   
   (f) Invokes a method from the proposer to close \( i \), if the proposer has proposed or is trying to propose a value for \( i \).
   
   (g) Invokes the proposer's method to begin the phase 1.

When the node recovers, multiple instances may have been closed in its absence, therefore it is necessary to update the node. The node must obtain all the closed instances and their associated values, that it has not learned, from other nodes. For this purpose, the \text{Updater} is registered to the log manager to be notified when a value of a closed instance is written in the log. When the \text{Updater} receives a notification from the log manager, it gets \( \text{cp} \) closed instances and their values, and, if it is necessary, updates learned and closed arrays, and the \text{lastClosedInstance}.

The learner provides the following methods to:

- get the last closed instance.
- notify the client.
- check if a value is learned.
- wait until the learner is updated to an opened instance equal to or greater than the instance number passed as a parameter.
2.4.1.1.3 Acceptor implementation

The acceptor (located at Implementations/ToolsAlgo/src/Consensus/Paxos) keeps the following variables:

- **highestN**: the highest acknowledged proposal number of a PREPARE message.
- **lastAcceptedProposal**: the proposal with the highest proposal number accepted.

Initially the acceptor recovers the **highestProposedN** and **lastAcceptedProposal** from a file.

When the acceptor receives a PREPARE message with a instance number *i* a proposal number *n*:

1. if it has not acknowledged any proposal number before, or *n* is greater than **highestN** and has not accepted any proposal with an instance number greater than *i*; then:
   
   (a) it sets **highestN** to *n*.
   
   (b) stores **highestN** in a file.
   
   (c) sends a PROMISE message with *i*, *n*, and if there is an accepted value for *i*, that is, if *i* and the instance number of **lastAcceptedProposal** are equal, then it also adds the **lastAcceptedProposal**.

2. else, it sends a REJECT message with *i*, *n*, the instance number of **lastAcceptedProposal** (if any), and **highestN**.

The acceptor sends a REJECT message to indicate to the proposer that **highestN** is greater than the received proposal number, but also if the received instance number is is smaller than the instance number of **lastAcceptedProposal**, then it is sent to indicate that the proposer must get the instance number of the last closed instance, or the instance number of the next opened instance.

When the acceptor receives an ACCEPT message with a Proposal *p*:

1. if it has not acknowledged any proposal number before, or if the *p*'s proposal number is equal to or greater than **highestN**.
   
   (a) it sets **lastAcceptedProposal** to *p*. 
2.4. CONSENSUS ALGORITHMS

(b) stores lastAcceptedProposal in a file.
(c) sends a LEARN message to all the nodes.

2. else, it sends a REJECT message with the same content as described above.

2.4.1.1.4 Proposer implementation

The proposer (located at Implementations/ToolsAlgo/src/Consensus/Paxos) maintains the following variables:

- **election**: instance of the Election (Bully) (see 2.3).
- **myId**: Proposer's identifier.
- **myP**: The proposal.
- **vQueue**: Queue of values that are pending to be proposed.
- **nPROMISES**: Number of PROMISE messages received.
- **p1P**: Proposal from a PROMISE message with the highest proposal number accepted in phase 1.
- **p1TO**: Timeout that expires, if the promises from a majority of the acceptors are not received in a determined period of time.
- **p2TO**: Timeout that expires, if the current instance is not closed in a determined period of time.
- **p1**: Boolean that when it is true indicates that the proposer is executing the phase 1.
- **p2**: Boolean that when it is true indicates that the proposer is executing the phase 2.

The period of time of each timeout is set with the NETWORK_DELAY of NetConf, the number messages to send and receive in each phase, the number of messages that the proposer will process using ACCEPTORS of Conf, and the time that is considered enough to process a message: 

\[ T = T_{\text{NETWORK}} \times 2 + ACCEPTORS \times T_{\text{process}} \]

p1 and p2 are used to indicate in which phase is executing the proposer. This is useful, when the proposer receives a REJECT message, because the proposer will process the message differently depending on which phase is executing. Since there are
different threads that may invoke the same methods. It is also useful to do not let a thread to execute the statements of a method.

When initiating, the proposer starts a thread called Helper, which checks continuously if the proposer is the new coordinator by using election's method `getNewCoordinator` passing `myId` (see 2.3). If the proposer is the coordinator it must take the role as it, so `Helper` begins the phase 1.

Since the proposer operation is more complex, it is better to model it in a state machine. In the figure 2.4.3 we can see the states and transitions that happen in a consensus instance. The algorithm starts at the `Beginning` state, and finishes at the `Closed` state that is when the instance is closed. We have a tuple `[myP: [i, n, v], nPromises, ph1P, state]` that is updated in each transition. Initially the tuple values are `[myP: [0, null, null], 0, null, beginning]`. Described below are the transitions to the states. To indicate that a value of a variable is incremented we add `++`, for example `n++` represents that `n` is incremented.

![State machine diagram](image)

Figure 2.4.3: State machine that shows the states and transitions that happen in a consensus instance

**B**: The proposer begins the phase 1 for the first time. It checks if it is the coordinator, if so, sets `nPromises` to 0 and `ph1P` to `null`, invokes the `myP`'s method to prepare
a new proposal: sets $v$ to null; increments $n$, sets $i$ to the instance number of the next opened instance. Finally, it sends a PREPARE message which contains $i$ and $n$ to the acceptors, and starts ph1T0. The tuple is updated to $[\text{myP}:[i, \, n++, \, \text{null}], \, 0, \, \text{null}, \, \text{p1\_waiting}].$

\text{P:} The proposer has received a PROMISE message $m$. A PROMISE is processed if the received instance number $i_m$ and proposal number $n_m$ are equal to $i$ and $n$ respectively. Then, it increments $n$Promises. Next, if the received promise has a proposal $p_m$ greater than ph1P’s, then it sets ph1P to $p_m$. The tuple is updated to $[\text{myP}:[i, \, n, \, \text{null}], \, \text{nPromises++}, \, \text{ph1P or null}, \, \text{p1\_waiting}].$ If the proposer have received the PROMISE messages from a majority of the acceptors, then stops ph1T0, and now depending if there is a value to be proposed or not, will trigger V or N.

\text{R1:} The proposer has received a REJECT message $m$ from an acceptor in the phase 1, which contains: an instance number $i_m$, a proposal number $n_m$, an instance number from an accepted proposal $a_i_m$, the highest accepted proposal number $a_n_m$. If $i_m$ and $n_m$ match with $i$ and $n$, then the REJECT is accepted. If $a_n_m$ is greater than $n$, then the proposer makes $n$ equivalent to $a_n_m$. Also, If $i$ is smaller than $a_i_m$, then it waits until the learner is updated, in order to get the instance number of an opened instance equal to or greater than $a_i_m$. At the end, the proposer begins the phase 1 executing the same procedure as in B. The tuple is updated to $[\text{myP}:[i, \, n++, \, \text{null}], \, 0, \, \text{null}, \, \text{p1\_waiting}].$

\text{TO1:} ph1T0 is expired, so the proposer begins the phase 1. Now, the tuple is $[\text{myP}:[i, \, n++, \, \text{null}], \, 0, \, \text{null}, \, \text{p1\_waiting}].$

\text{N:} If there is not a value neither in the PROMISE messages nor in vQueue, then the proposer waits to receive a value from a client. When the proposer receives a value, begins the phase 1. The tuple is updated to $[\text{myP}:[i, \, n++, \, \text{null}], \, 0, \, \text{null}, \, \text{p1\_waiting}].$

\text{V:} If at least one of the PROMISE messages had a proposal with a value, then the proposer sets $v$ to the ph1P’s value. Otherwise, it sets $v$ to a value of vQueue, if this value has not been learned. Finally, it sends the ACCEPT messages, that contain myP, and starts ph2T0. The tuple is updated to $[\text{myP}:[i, \, n, \, v], \, \text{nPromises}, \, \text{ph1P or null}, \, \text{p2\_waiting}].$

\text{R2:} The proposer receives a REJECT message $m$ in the phase 2, with $i_m$ and $n_m$ equals to $i$ and $v$ respectively. If $a_n_m$ is greater than $n$, then it makes $n$ equivalent
to \( an_m \). Finally, it stops \( \text{ph2T0} \), and begins the ph1. The tuple is updated to 
\([\text{myP}:[i, \text{n}++, \text{null}], 0, \text{null}, \text{p1\_waiting}]\).

**TO2:** \( \text{ph2T0} \) is expired, so the proposer begins the phase 1. The tuple is updated to 
\([\text{myP}:[i, \text{n}++, \text{null}], 0, \text{null}, \text{p1\_waiting}]\).

**L:** The learner invoked the method to close the instance passing as parameters: the instance number, \( i \) and the chosen value, \( v \). If \( i \) matches with \( i \), then it stops \( \text{ph1T0} \) and \( \text{ph2T0} \). Since \( v \) may be in \( v\text{Queue} \), then it tries to remove \( v \). The tuple is updated to \([\text{myP}:[i, \text{n}, \text{null}], 0, \text{null}, \text{closed}]\). Note that \( i \) may be closed when the proposer is in any state, since may be another proposer has proposed a value for \( i \).

When the proposer receives a value from a client, if the value is not into \( v\text{Queue} \) and has not been learned, and the proposer is the coordinator, then it puts the value in \( v\text{Queue} \), and begins the phase 1. If the proposer is not the coordinator, then it sends the value in a REQUEST message to the coordinator.

Each time that the proposer decides to begin the phase 1, checks if it is the coordinator, if so continues. If it is not the coordinator then it forwards the client values of \( v\text{Queue} \) (if any), to the current coordinator.

### 2.5 Transaction Algorithms

A distributed transaction consists in a set of operations that are executed in multiple nodes. At the end, of a transaction all the nodes must commit or must abort the transaction, because a transaction is completed only if all the operations are executed. For this purpose, a coordinator is necessary to communicate to all the nodes that participate in the transaction, if the final decision commit or abort.

A transaction has the following properties [1]:

- **Atomic:** A transaction can be viewed as a single action. All operations are executed successfully or must have no effect at all.

- **Durability:** If a transaction is completed successfully, the effects must be permanent.

- **Isolation:** Transactions cannot interfere with each other.

Described below is the the Two-phase commit protocol that is used to perform distributed transactions correctly.
2.5. TRANSACTION ALGORITHMS

2.5.1 Two-phase commit protocol

A simple approach to complete a transaction is that the coordinator to request to the participants of the transaction to perform an operation, and then repeat the request until all of them acknowledge that have successfully executed the transaction. The drawback of this approach is that a participant cannot report that it has not been able perform the operation. The two-phase commit protocol allows to the participants of the transaction to abort the transaction, when they have not been able to execute the operation.

Since the node failures can occur, the coordinator and each participant store their state in a log in stable storage. The participant also it saves the modified objects. In this way, when a node recovers can take the corresponding actions based on its state.

As its name indicates, the Two-phase commit protocol is executed in two phases [6]:

Phase 1 (Voting phase)

1. The coordinator writes WAIT in its log and sends a VOTE_REQUEST message to the participants.

2. When a participant receives a VOTE_REQUEST message from the coordinator, it may:
   
   (a) indicate that it is prepared to commit. So, it stores the modified objects and the state READY in its log, and responds with a VOTE_COMMIT message.
   
   (b) or abort. Then, it writes ABORT in the log, and responds with a VOTE_ABORT message.

Phase 2 (Decision phase)

3. The coordinator receives the votes from the participants:
   
   (a) If there no failures and all the participants voted to commit the transaction, then it records COMMIT in the log, and sends a GLOBAL_COMMIT message to all the participants.
   
   (b) Otherwise, aborts the transaction. So, it writes ABORT in the log, and sends a GLOBAL_ABORT message to the participants that voted to commit.

4. If the participants that voted to commit receive a:
(a) GLOBAL_COMMIT message, then they write COMMIT in their log.
(b) GLOBAL_ABORT message, then they record ABORT in their log, and undo the modifications of their stored objects.

The figure 2.5.1 shows the states of the coordinator and the participant, and the messages involved in each transition [6].

![State Machine Diagram](image)

**Figure 2.5.1**: 2PC (a) Coordinator’s state machine; (b) Participant’s state machine [6]

**Timeout actions**

The coordinator and each participant start a timeout before transitioning to a state in which they are going to be waiting to receive a response. When the timeout expires, they act according to their state.

As we can see in the figure 2.5.1, initially the participant is in the INIT state waiting to receive the VOTE_REQUEST message from the coordinator. If the timeout is expired, then the participant can decide to abort. Thereby, the object that would have been changed, is released, and it can be used in other transactions.

If the coordinator is in the WAIT state, waiting for the votes from the participants. Then, it decides to abort, so performs the actions of the step 3 (b).

When the participant is waiting to receive the decision from the coordinator, in state READY. In this case, it cannot take a decision by itself, as the coordinator may have reached a decision (commit or abort), that the participant does not know. So, it asks which is the decision, by sending a GET_DECISION message to the coordinator [1].
Recovery

When the coordinator or a participant recovers, takes the following actions based on their state in its log:

- Coordinator:
  - WAIT: It aborts unilaterally, so it writes ABORT in the log.

- Participant:
  - INIT: It decides to abort. It writes ABORT in the log.
  - READY: It performs the same actions as if the timeout had expired. Therefore, it asks the coordinator which is the final decision.

Conclusions

We can see that independently of the state of the coordinator and the participants, the two-phase commit protocol ensures that, at the end, all the participants will reach the same decision; all of them commit or none of them. No participant takes a decision, if there is the possibility that the coordinator and may be other participants have reached a decision that it does not know.

This algorithm is prone to blocking. Since, if the coordinator fails and a participant is in the READY state, then the participant blocks, because it must ask to the coordinator if it must commit or abort, and therefore it must wait until the coordinator recovers.

Although the two-phase commit protocol is prone to blocking, it guarantees that eventually the transaction is completed, even if failures occur.

A solution to avoid blocking is the three-phase commit protocol. But, this solution is not used in practice, since the blocking situations of the two-phase commit do not occur frequently. Instead, the real systems use a version of the two-phase commit protocol [13].

2.5.1.1 Two-phase commit protocol implementation

In the implementation of the two-phase commit protocol (2PC protocol) (located at Implementations/ToolsAlgo/src/Transactions), the nodes are both coordinator and participant. The node that is the coordinator is fixed before the execution. There are two types of coordinators and participants:
• *Transaction coordinator* (*TransCoordinator* class): It interacts with the clients that want to perform a transaction.

• *Transaction participant* (*TransParticipant* class): It receives and executes the operations of the transactions.

• *2PC coordinator* (*TwoPCCoordinator* class): It is the coordinator of the two-phase commit protocol.

• *2PC participant* (*TwoPCParticipant* class): It is the participant of the two-phase commit protocol.

The two types of coordinators and participants need to exchange parameters between them. For this purpose, two threads, called *coordinator thread* (*NodeCoordinator* class) and *participant thread* (*NodeParticipant* class), were created.

**Identifier of the transaction**

Each message contains the identifier of the transaction, transID, which is composed by:

• **clientID**: the client’s identifier.

• **coordinatorID**: the identifier of the coordinator.

• **seq**: the sequence number. Incremented by the *transaction coordinator* for each new transaction.

When a message is received, first the transID is checked, in case that a delayed message from a closed transaction is received. If the received transID does not match with the current transID, then the message is ignored. In the case of the GET_DECISION messages, the transID is not checked.

**Interaction with the client**

The process that is followed since the client begins a transaction, until it receives the result is the following.

1. The client sends an OPEN_REQUEST message to the *transaction coordinator*, in order to open a transaction.
2. When the transaction coordinator receives an OPEN_REQUEST message, it responds with an OPEN_RESPONSE message with a transID.

3. The client sends a TRANSACTION message that contains transID and a list of the transaction’s operations to the transaction coordinator.

4. The transaction coordinator receives the TRANSACTION message and sends each operation along with transID to the transaction participants. The coordinator thread invokes a method of the 2PC coordinator to start the voting phase of the 2PC protocol, passing a list of the participants and the transID as parameters.

5. When a transaction participant receives the operation tries to execute it. Then, the participant thread passes the result of the execution, the operation and the transID to the 2PC participant.

6. When the 2PC coordinator and participants of reached a decision (commit or abort). Then, The coordinator thread invokes the corresponding method of the transaction coordinator , to send a CLOSE_RESPONSE message with the final decision and the transID to the client.

7. When the client receives the CLOSE_RESPONSE message, checks if the transaction is completed. If the decision is commit then the transaction is completed. Otherwise, the client requests to open a new transaction (step 1).

The transaction coordinator only accepts one transaction per time. So, if it receives a OPEN_REQUEST message, when there is a transaction in course, then this message is dropped.

Execution of the operations

Note that the management of the objects is not implemented as is not necessary for the operation of the 2PC protocol. Therefore, a transaction participant instead of executing the received operation, it returns a random result of the execution of the operation.

Log management

The 2PC coordinator and each 2PC participant use a log manager (TransLogManager class) to manage its log.

An entry of the log may contain the following fields:
• **transactionID**: The transID.

• **state**: The state of the coordinator/participant.

• **operation**: The executed operation.

The `TransLogManager` class offers the following methods to:

(a) add a new entry.

(b) get the last entry.

(c) get the last entry with the transID passed as a parameter.

**2PC implementation**

The implementation of the 2PC **coordinator** and the 2PC **participant** (located at `Implementations/ToolsAlgo/src/Transactions/TwoPC`) operate in the same way as it is described in the description of the algorithm.

The 2PC **coordinator** and 2PC **participant** have separate logs, that is, each one has its own log. Both the 2PC **coordinator** and 2PC **participant** initially check the state that contains the last entry of log (if any), and take the corresponding action as it is described in the description.

After checking the log, the 2PC **coordinator** just waits to receive the messages from the participants. When there is a new transaction, the **coordinator thread** invokes the method to execute the step 1 of the voting phase, and also sets the list of participants and the transID.

When the 2PC **coordinator** receives a GET_DECISION message with a transID, invokes the method (c) of the log manager passing the received transID, in order to get the final decision. Finally, it sends the decision to the corresponding participant.

Once the 2PC **participant** has checked the log, if it got the decision of the transaction before the failure (if any), and it is not prepared to vote, then waits until the **participant thread** passes the result of the operation's execution, the operation and the transID. The result is a boolean; it is true if the execution has been successful. When the mentioned parameters are passed, then the 2PC **participant** is in the INIT state, waiting to receive the VOTE_REQUEST message from the coordinator.
Execution

As mentioned, the coordinator is fixed before the execution in the TransactionStub class. It is fixed to 0, because the idea is to initiate first the servers (coordinators and participants) before the client. Note that since the transaction coordinator sends an operation to each participant, then we must initiate at least as many servers as operations has a transaction.

In the interface Conf (located at Implementations/ToolsAlgo/src/Transactions), there is a field where we can set the probability of an operation to be executed successfully.
Chapter 3

Use Case: Shared Calendar

It was decided to develop part of a distributed application, in order to prove that with the tools and a set of distributed algorithms that have been developed, it can be build a system that can be integrated in a final application, which can be used in the real world. Furthermore, developing an application, offers new conditions that may imply to develop new tools, and to improve the existing ones and the distributed algorithms.

The distributed application is a shared calendar. A shared calendar consist in a collection of devices, where each one of them has a local calendar, when one of the devices performs a modification, for example to add a new event, this modification is applied to the calendars of all the devices.

The distributed system must be implemented taking in account the following conditions:

SC1: Eventually all the local calendars of the devices must be identical.

SC2: Any device can add or delete events on the calendar.

SC3: The devices may be offline and online at any moment.

SC4: The number of devices is dynamic. New devices could join to the collection at any moment.

In the following sections, first a comparison of various of the implemented algorithms is made to select the most suitable to build the system. Finally, the replicated log system is described.
3.1 Selection of the algorithms

Replication

Looking at SC1, we can see that a calendar must be replicated in multiple nodes. Therefore, we need an algorithm that can be used for replication.

The first algorithm to come in mind is the Paxos consensus algorithm (see section 2.4.1). The implementation of the Paxos algorithm can be used for replication, since the nodes in the network agree in the next operation to be chosen, among a set of operations, and finally they write it in their local log. Therefore, all the nodes will have an identical log of ordered operations. The next section discuss how the local log in each node is used.

The Two-phase commit protocol (see section 2.5.1) is another algorithm that can be used for replication. The Two-phase commit is intended to be used for transactions, but this protocol can be adapted to finally follow the same approach as in Paxos. Each client instead of sending a transaction, that is a set of operations, sends just one operation, and each server instead of trying to execute an operation, writes it in a local log.

Now, it must be decided which of the two algorithms is the most suitable to form part of the shared calendar application. Taking in account the condition SC3, the 2PC has the drawback that a failure of a single node leads to an abort. Instead, Paxos tolerates more failures, since only needs a majority of the nodes to chose an operation. Also, 2PC blocks if the coordinator fails just when it has requested the votes of the participants. But, in Paxos if the coordinator fails, another proposer can take the role of the coordinator and propose operations, without violating the correctness of the algorithm, as we can see in the second example of the section 2.4.1. Therefore, it is clear that the most suitable algorithm is Paxos.

Election

Since Paxos has been selected, we need an election algorithm to elect the proposer that will be the coordinator. If we look at the section 2.3 there are two choices, the Chang and Roberts ring ("Ring") and Bully.

Ring has the advantage of requiring less messages than Bully. But bully is faster, which is more convenient. On the other hand, Ring blocks if a single node fails, since the ring is broken. On the contrary, Bully never blocks, because detects the node failures. Therefore, Bully is the most appropriate of the two algorithms.
3.2 Replicated log

In the previous section, it has been explained that Paxos can be used to replicate a log of ordered operations, and also that it tolerates failures. Also, it is explained that Bully is used as the election algorithm. In addition, we have seen that the Paxos implementation uses the reconciliation mechanism tool (section 1.7), which allows to a replica that has recovered from a failure to obtain the operations that have missed, and finally write them in the log. Therefore, a system called replicated log using Paxos, the reconciliation mechanism, and Bully is build. In the literature is published the description of another system that also use Paxos to replicate a log [8].

Looking at the description of the shared calendar in the introduction of this chapter. For the replicated log, a device is called calendar replica. The modifications on the calendar performed by the calendar replicas are operations, which are the input of the replicated log. Then, when the replicated log system receives an operation, all the calendar replicas agree to write the operation in their local log. Since, all the calendar replicas write the operations in their local log in the same order; if each calendar replica applies the operations of the log to a local database, then eventually all the replicas have identical databases. Finally, the content of the database (the events) can be shown in the calendar’s user interface. Therefore, it is clear that the condition SC1 is satisfied.

In the figure 3.2.1 we can see the replicated log system architecture, which is contained in a calendar replica. The new added elements are the ones with the word “Calendar” at the beginning of their name.

Calendar learner

LearnerC class (located at Implementations/SharedCalendar/src/Paxos) overrides the method to notify to the client that a value is learned of the the Learner class (see section 2.4.1.1.2). Instead of sending a RESPONSE message, if the calendar learner has learned a value, and this value contains an operation that has been issued by the calendar client that is built on top of it, then notifies it. Note that this is an optimization, since each time that a value is learned, it is not necessary to send a message to the client that issued the value.
CHAPTER 3. USE CASE: SHARED CALENDAR

Figure 3.2.1: A calendar replica

**Calendar client**

Each calendar replica has a calendar client that sends the operations introduced from the calendar GUI, contained in values, to the current coordinator. The calendar GUI writes the operations in a log, called log of pending operations, so that they can be recovered in case that the replica fails.

The calendar client (ClientC class, located at Implementations/SharedCalendar/src/Paxos) maintains the following variables:

- **myId**: The identifier of the calendar replica.
- **pendingQueue**: Queue of pending operations.
- **operationsLog**: Log of pending operations, which is an instance of Log (see section 1.4).
- **currentValue**: The value to be learned that contains a pending operation.
- **requestTO**: Time out that when expires, if currentValue is not learned, then it sends currentValue again.
- **l**: The next logical time to be associated to an operation.
3.2. REPLICA TED LOG

- **learner**: The calendar learner.

The calendar client allows to other threads to be registered, in order to receive notifications when an operation is added or removed to and from `operationsLog`.

A pending operation is a data structure (`OperationsLogEntry` class, located at `Implementations/SharedCalendar/src/Log`) that contains the fields:

- **time**: The time when the operation was written in the log.
- **operation**: operation introduced through the calendar GUI

The calendar client offers methods to:

- add a new pending operation. It is invoked by the calendar GUI.
- receive a notification from the calendar learner indicating the a value issued by this calendar client has been learned.

The calendar client increments `l` and stores it in a file, after a new pending operation is added.

When the calendar client initiates, first it recovers `l`'s value from a file. Next, if the calendar replica has been recovered from a failure, then it may obtain a value that was issued before its failure, so the calendar client invokes the method of `learner` to know if a value is learned, in order to check if the next pending operation to be issued is learned. If so, then it is removed from the `pendingQueue` and the `operationsLog`.

When there is a pending operation, the calendar client operates in the following way:

1. Prepares the value to send. So, it creates new value that contains the pending operation along with its associated time and `myId`, and stores it in `currentValue`.
2. Sends a REQUEST message that contains `currentValue`.
3. Starts the `requestTO` and waits to be notified.
4. Receives a notification from the calendar learner indicating that `currentValue` is learned.
5. Removes the operation from the `pendingQueue` and the `operationsLog`.

Note that if the `requestTO` expires, then if `currentValue` is learned, notifies it to the calendar client. Otherwise, it sends `currentValue` again.
Replicated log operation

The figure 3.2.2 depicts a basic example of the execution of the replicated log system. In the figure we can see that the calendar replicas 0, 1, 2 are in the Active set. This organization of active and passive nodes is explained in the description of the Paxos implementation (see section 2.4.1.1). The active calendar replicas are the ones that chose the next operation to write, and the passive calendar replicas wait to receive the next operation to write. The total number of members of the active set is 3, and the Paxos’ instances until $i - 1$ are closed, remember that to close an instance is to write an operation. It is assumed that a majority of the calendar replicas in the active set are alive enough time to close an instance of Paxos.

The sequence of actions of the example are the following:

(a) The calendar replica 3 sends an operation $o$ contained in a value $v_3$ to the calendar replica 2 that is the current coordinator elected in a Bully election.

(b) A majority of the active calendar replicas $(0, 2)$ have executed a Paxos’ consensus instance $i$ and have chosen the $v_3$, therefore each one sends a LEARN message to
all the other calendar replicas. Finally, the calendar learner of each replica writes the received operation in its local log. The learner of the replica 3 also notifies to the calendar client that \( v_3 \) has been learned.

(c) A new calendar replica 5 joins to the collection. Later, replica 4 sends a MySN message with the SN of the last written operation, \( \bar{i} \), to the replica 5.

(d) Then replica 5 sends a GET_WRITES message with SN 0 to replica 4.

(e) Then replica 4 sends the written operations from 1 to \( \bar{i} \), each one in a WRITE message, to 5.

In the example, we have seen that a new replica can join to the collection at any moment, so the condition \( SC_4 \) is met. And each calendar replica is able to add and remove events to the calendar through its calendar client, so \( SC_2 \) is also met.

### 3.3 Calendar GUI

The calendar GUI (Graphical User Interface) (CalendarGUI class, located at Implementations/SharedCalendar/src/Calendar) creates a window, with which the shared calendar’s user can interact, to enter operations that will be applied to the all the calendar replicas.

The calendar GUI shows the pending operations introduced by the user, and the written operations in the local log. For this purpose, the calendar GUI is registered to the calendar client and the log manager in order to obtain updates from them (see the figure 3.2.1).
CHAPTER 3. USE CASE: SHARED CALENDAR
Conclusions and future development

In this thesis, we see that the developed tools make easier the implementation of the algorithms, for instance, Message's class (see section 1.1.2) that allows complex structures to be easily send and retrieved in a human-readable format as JSON is.

In the second chapter we can see that during the process of the description of the algorithms they became more complicated. This is due to the fact that the algorithms should be able to solve more complex tasks.

Furthermore, we can see that when an algorithm is put from the theory to the practice, that is, when an algorithm is implemented from its specification, some adaptations and/or tools could be needed in order to make it able to interact with other algorithms, use it in a real system or in a distributed application.

Also, it has been proved that the developed tools could be used to easily implement distributed algorithms, and this ones could be finally combined to build distributed systems that could form part of a final application that may be used in the real world.

In a future project, first of all, a system to avoid the need to use the identification server could be developed and, this way, build a totally distributed system. A distributed system able to generate unique identifiers could be developed, such as UUID (Universally Unique Identifier). Furthermore, a distributed failure detector should be implemented.

Secondly, as we have seen the communication system is just working locally. Then, a communication system working in a local network could be developed. This one could be implemented with the current system, making adaptations to the identifier server. And even, across multiple networks, which would mean to use a new reliable transport protocol.
Bibliography


Glossary

2PC  Two-phase commit
CS   Critical Section
GSON Google JSON
GUI  Graphical User Interface
JSON JavaScript Object Notation
UDP  User Datagram Protocol
UUID Universally Unique Identifier