
# The library of distributed protocols

## Master Thesis

Aleksandar Lazic
from
Crans-Montana VS, Switzerland

Faculty of Science and Medicine
at the University of Fribourg

January 25, 2021

Professor Christian Cachin
Jovana Micic

Cryptology and Data Security Research Group
Institute of Computer Science
University of Bern, Switzerland



# Abstract

Distributed computing aims to solve computational problems using multiple hardware and software components connected over a network and potentially geographically distant. The management of distributed systems is really challenging because processes must achieve some form of cooperation despite the fact that some of them may fail or be disconnected. In this work, we also look at Byzantine processes that can behave in an arbitrary manner, either intentionally or because of a bug, thus interfering with the proper execution of an algorithm. In order to deal with such processes, some cryptographic abstractions are integrated in the design of the algorithms.

We implemented a library of distributed algorithms in a modular way, focusing on the creation of programming abstractions interacting together. The development starts from the ground up with the abstraction of processes and communication links, and then gradually progresses to various broadcast versions or the well-known consensus problem. The technology used is DistAlgo, a high-level programming language very close to pseudo-code. Its ease of use as well as its complex synchronization conditions make it a serious contender for the implementation of distributed algorithms.

# Acknowledgments

I would like to thank my supervisor Jovana Micic for her advice and help throughout this work. I also thank her for her great patience and time. I would also like to thank Professor Christian Cachin for his critical thinking on the different issues that came up during the implementation. In addition, I wish to thank Orestis Alpos for our constructive discussions throughout the development. Finally, I am grateful to my family and friends for their continuous support.

# Contents

# 1 Introduction

Distributed algorithms enable cooperation between multiple processes working together. Several software and hardware components are connected together on a network subject to failures. This could lead to the crash of some processes due to network errors. Meanwhile, the correct processes that are still running need to synchronize their activities consistently. The biggest challenge is therefore to design a robust distributed system tolerating such failures and potentially malicious attacks from faulty processes. The objective of this thesis is to develop a library of distributed algorithms in a modular way for educational purposes.

In order to build a fault-tolerant system, the chosen approach is to implement distributed programming abstractions from the basics by gradually moving towards more advanced problems. We start by defining the underlying physical model with *processes* and communication *links*. Each abstraction works as an independent service accessible through some interface and is associated with a set of properties that it must respect. The different algorithms are built by stacking multiple modules together to meet the defined requirements. Although a monolithic architecture can be more efficient than a modular one, the risk of errors is greater since algorithms tend to rapidly gain in complexity [1]. The whole development process is done with the help of the *DistAlgo* library [2].

DistAlgo is a language very close to pseudocode allowing to write distributed algorithms at a high level. This library allows one to focus on the accuracy and understanding of the algorithm rather than on some low-level mechanics. Indeed, it is possible to easily send messages and manage their reception by respecting certain delivery conditions. Moreover, high-level queries like quantification or list comprehensions can be applied on a set of messages offering complex synchronization conditions [3]. A built-in list containing the history of received messages is also available by default. The DistAlgo compiler compiles code in Python as the target language. It allows students to easily learn this language thanks to the simplicity of Python, making complex algorithms clear and precise. Moreover, the developed programs are directly executable locally or even on several remote nodes.

An earlier study shows that the implementation of multi-Paxos with several optimizations takes about 300 lines of DistAlgo code versus approximately 3000 lines of C++ [3]. DistAlgo was also successfully used in the development of some Byzantine fault-tolerant algorithms in a Bachelor's thesis at the University of Bern [4]. As a result of this work, the idea came up to implement a library of distributed algorithms allowing future students to combine the theoretical aspect of the class with the concrete implementation. This work has realized such a library.

In Chapter 2, the different programming abstractions required to successfully build a distributed system are explained. In addition, a brief description of DistAlgo is presented, explaining the different programming concepts. Chapter 3 presents the incremental development carried out throughout this work, starting with point-to-point communication links and ending with the total-order broadcast built on top of the consensus. A benchmark comparing two different implementations of the total-order broadcast is also presented in Chapter 4. Finally, in Chapter 5, the obtained results are evaluated and possible improvements for further work are assessed.

# 2 Background

## 2.1 Distributed Systems

A distributed system is defined as a system in which components located on different networked computers seek to achieve some form of cooperation and coordinate their actions. It is composed of a set of processes where each element works simultaneously and communicates with each other through the exchange of messages. Therefore, care must be taken to anticipate the potential issues that may arise following a process crash or network errors. Leslie Lamport's famous definition of a distributed system clearly illustrates the complexity of the problem: *"A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable."* [1].

Some of the processes may stop working due to a failure or a disconnection from the network. The main challenge is to ensure that the processes that are still working remain consistent and tolerate such failures. In addition, the system should also be able to tolerate malicious adversaries that may deliberately behave to disrupt communication between processes and affect the proper execution of algorithms.

The algorithms implemented in this thesis are mostly based on the book "Introduction to Reliable and Secure Distributed Programming" [1]. It explains how to build a robust distributed system and different approaches to solve problems such as process failures or how to prevent faulty processes from interfering with the proper functioning of the system. However, it is a very challenging task to develop such systems with correctness and efficiency.

## 2.2 Distributed Programming Abstractions

The first step in the creation of a distributed system is to define certain property-based abstractions that can be applied to a wide range of systems, regardless of the type of machines or their operating systems. A clear definition of these abstractions is necessary in order to have a good overview of the fundamental components of a distributed system. Indeed, each algorithm implemented in this thesis is built using these abstractions.

**System Model.** The underlying physical system of a distributed system can be represented by two components: *processes* and *links*. Processes are active entities that perform computations such as a computer, and links represent the physical and logical network used by processes to communicate over the network. It is not possible to represent these two components with a unique abstraction because of all the possible properties of a distributed system. Indeed, different process abstractions will exist according to the type of faults they may have. Communication links will differ according to the delivery property they provide.

**Composition Model.** The composition model refers to the description of distributed algorithms. Since the book [1] aims at a general implementation, pseudocode was chosen for the description because of its simplicity. Additionally, it allows the reader to focus only on the understanding of the algorithm and omit implementation-related details. Each algorithm is then accessible as an independent service through the use of *events* defined in its Application Programming Interface (API).

Every process in the system runs a set of components composing the software stack. At the top is the application layer, whereas the networking layer is usually at the bottom. The modules inside the same stack will communicate through the exchange of events. As shown in Fig. 2.1, components deal with two types of events: *requests* and *indications*.

*Request* events are used by a module to invoke a service to another module. Given the software stack, a module can send a *request* event to the module below it. For example, the application layer can trigger a request event to the component in charge of the broadcast.

*Indication* events are used by one module to provide information to another module. Continuing with the broadcast example, the message is transmitted to each process using the *indication* event.

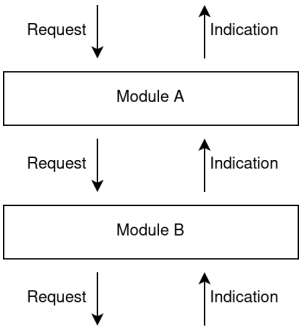


**Figure 2.1.** Composition model.

**Abstracting Processes.** A *process* represents the computational unit in a distributed system. The system is composed of $N$ different processes executing an instance of an algorithm. The term *failure* is used when a process deviates from the correct execution of the algorithm. There are multiple types of possible failures such as a simple crash or malicious behavior.

Considering the *crash-stop* abstraction, a process is considered *faulty* if it crashes at some point during its execution. On the contrary, it is assumed to be *correct* if it never stops and follows the algorithm specification.

Arbitrary faults, also called *Byzantine* processes, occur when a process arbitrarily deviates from the current algorithm, intentionally or unintentionally. In order to deal with them, no assumptions must be made about the behavior of these faulty processes allowing them to send any type of message. In addition, cryptographic abstractions whose security properties cannot be broken are used.

**Abstracting Communication.** The network components of the distributed system are represented with the *link* abstraction. The processes are fully connected and able to communicate with any other in the system. Each message sent via a link is unique and contains enough information for the recipient to recognize the sender and respond if necessary. It is important to notice that messages can get lost as they pass through the network and consider that in the abstractions.

**Classes of Algorithms.** Different classes of algorithms are designed to implement the distributed programming abstractions. These can affect the failure assumptions, the environment or the system parameters. Classes of algorithms can be of the type: [1]

1. *fail-stop*, designed under the assumption that processes can fail by crashing but the crashes can be reliably detected by all the other processes;
2. *fail-silent*, where process crashes can never be reliably detected;
3. *fail-noisy*, where processes can fail by crashing and the crashes can be detected, but not always in an accurate manner (accuracy is only eventual);
4. *fail-recovery*, where processes can crash and later recover and still participate in the algorithm;
5. *fail-arbitrary*, where processes can deviate arbitrarily from the protocol specification and act in malicious, adversarial ways;
6. *randomized*, where in addition to the classes presented so far, processes may make probabilistic choices by using a source of randomness.

In the context of this thesis, we focus on *fail-stop* and *fail-silent* classes of algorithms. However, two algorithms belonging to the *fail-arbitrary* model have also been implemented.

**Cryptographic Abstractions.** Messages are protected from malicious adversaries using three cryptographic primitives: *hash functions*, *Message-Authentication Codes (MACs)* and *digital signatures*.

A cryptographic hash function maps a bit string of arbitrary length to a m-bit unique representation. A hash function is said *collision-free* if there are no two distinct values $x$ and $x'$ such that $H(x) = H(x')$. Cryptographically secure hash functions make it impossible for an attacker to find collisions.

A MAC authenticates data between two entities using a shared symmetric key, known only to the sender and receiver of a message. A sender can compute an authenticator for a given message and send it to the receiver along with the message itself. The recipient can then verify that the message has really been authenticated by the sender. In practice, MACs are based on fast symmetric cryptographic primitives such as hash functions.

Digital signatures associate a public/private key for each entity in the distributed system. The private key is secret and cannot be known by any entity other than the one for which it is intended. When a process sends a message, he uses its own private key to sign the message. The receiver process can then verify the validity of the signature using the sender's public key.

**Quorums.** *Quorums* are used to design fault-tolerant algorithms. Assuming a system with $N$ crash-fault processes, a quorum is any set of more than $N/2$ processes ensuring that every two quorums overlap in at least one process. However, in a system that includes Byzantine processes it may happen that two quorums don't intersect in a correct process. Therefore, a *Byzantine quorum* tolerating $f$ faults is defined as a set of more than $\frac{N+f}{2}$ processes. Thus, each Byzantine quorum contains more than $\frac{N-f}{2}$ correct processes.

$$\frac{N+f}{2} - f = \frac{N-f}{2}$$

As a result, two disjoint Byzantine quorums would together have more than $N - f$ of correct processes, which is not possible because there are only $N - f$ correct processes. Therefore, one correct process is

contained in both quorums.

$$\frac{N-f}{2} + \frac{N-f}{2} = N - f$$

Finally, considering algorithms in the fail-arbitrary model, up to $f$ faulty processes may not follow the algorithm specifications. It is then necessary that at least a Byzantine quorum of correct processes exist in the system to make progress. This can only happen when $N > 3f$.

$$N - f > \frac{N+f}{2} \iff N > 3f$$

## 2.3 DistAlgo

*DistAlgo* is a very high-level language similar to pseudocode allowing the user to focus mainly on the algorithm logic compared to lower-level programming languages. The library offers a simple way to send messages between processes and manage their delivery under certain conditions. This project uses a DistAlgo compiler with Python as the target language. In the following part, the most important DistAlgo concepts are presented based on the language description [3].

### 2.3.1 Processes and Messages

**Process definition.** A process in DistAlgo is the instance that executes some predefined code such as threads in Java. The code snippet below describes a new class $p$ extending the class $process$. The $process\_body$ is the code that will be run by the process.

```
class p extends process:
    process_body
```

A *setup* method can be defined to initialize the required variables before the process execution.

**Process creation.** The creation of a process is done using the keyword *new*. Two more optional parameters can be defined: *n* as the number of instances of that process to create, and *at* which represents the hostname of the node running the process. The statement below creates *n* instances of type *p* on the host represented by the expression *node_exp*.

```
n new p at node_exp
```

**Sending messages.** One can send a message using the following statement. It sends a message of the form *mexp* that can be of any type to the process or set of processes represented by *pexp*.

```
send mexp to pexp
```

**Deliver messages.** A process can define a *receive* clause to handle message delivery. The following delivers a message of the form *mexp* from process *pexp*. The statements corresponding to the *handler_body* are then executed.

```
receive mexp from pexp:
    handler_body
```

**Pattern matching.** A pattern can be used to match a message in a receive handler. By convention, messages are expressed using tuples. The following example matches every triple where the first two elements match *'ack'* and the value of the variable *n*; the underscore matches anything. Finally, the sender must also bind to *a*.

```
receive ('ack', =n, _) from a
```

**Synchronization.** An non-blocking *await* statement is used to wait until the Boolean-valued expression *bexp* becomes true.

```
await bexp
```

### 2.3.2 High-level Queries

High-level queries can also be used over a set of messages. There are three main types of high-level queries: *comprehension*, *aggregation* and *quantification*. It is possible to combine them to create complex synchronization conditions.

**Comprehension.** A comprehension query returns a set of processes matching some Boolean expression. The following Python example returns the list of processes from which the running process received a *Hello* message.

```
setof(p, received(('Hello',), from_=p))
```

**Aggregation.** An aggregation query uses an operator such as *len*, *sum*, *min* or *max*. The previous example is extended to get the process with the smallest identifier that sent *Hello*.

```
min(setof(p, received(('Hello',), from_=p)))
```

**Quantification.** A quantification query can either be in the universal or existential form. An universal quantification returns true if and only if *each* element of the set matches some Boolean expression. On the other hand, the existential form returns true if *some* member satisfy the condition.

```
each(p in procs, has=received(('Hello',), from_=p))
some(p in procs, has=received(('Hello',), from_=p))
```

### 2.3.3 Configuration

It is also possible to define some configuration items such as the delivery of messages using the first-in first-out ordering (FIFO). Lamport's logical clock can also be used, unlike a vector clock which is not yet built-in the library.

```
configure channel = {fifo, reliable}
configure clock = Lamport
```

### 2.3.4 Logging

A logging module can also be used to print some information alongside with the process identifier. Each print statement comes with a level whose default value is *Logging.INFO* [5]. The default printing level can then be set at runtime where printing will only be displayed if the level is equal or higher. By default, everything is displayed on the console line but can also be redirected to a file.

```
output(exp1, ..., expk, level=l)
```

# 3 Library of Algorithms

This section presents the library of algorithms implemented using DistAlgo. The goal of this implementation is to keep the modularity and the pedagogical structure presented in the book [1]. The focus is more on the properties of the algorithms to be developed than on performance and optimizations.

## 3.1 Modular approach

The distributed programming abstractions described in Section 2.2 are built in a modular way and used together to solve complex distributed computing problems. Implementation starts from the bottom with the creation of communication link abstractions required in the message-passing model. These abstractions will, among others, be used for the design of the broadcast abstractions. As the complexity gradually increases, it is necessary to make sure that the implemented modules work properly in order to be able to rely on them for subsequent abstractions. Firstly, simpler classes of algorithms are considered such as the crash-stop model before adapting the abstractions for some more complex system models.

## 3.2 Architecture Design

When started, a DistAlgo process executes the code in its *run* method and waits for incoming messages to be handled. Each module implementing an algorithm is represented as a DistAlgo process that is running its own code with its own receive handlers. The first step is therefore to set up an internal mechanism to create the necessary modules according to the algorithm chosen by the user.

**Process Initialization.** The creation of each abstraction is delegated to a special process called the *initiator*. It takes care of creating, configuring and starting each process. In order to perform these different tasks, the initiator needs several parameters to be initialized depending on the type of algorithm or for the remote execution.

```
class Initiator(process):
    def setup(data:dict, algo:str, n:int, f:int, sender:str, receiver:str, m:str,
              props:list, crash:bool, ips:list, remote:bool):
```

The first step is to create the module stack by reading the *algorithms.json* file. This file describes the required components according to the desired algorithm and needs to be updated each time a new component is added. The stack is then built recursively by going through each abstraction and creating its underlying components. Suppose we want to run the perfect failure detector which relies on perfect point-to-point links. Perfect point-to-point links, in turn, use stubborn point-to-point links. Thus, the stack of modules will consist of three layers running on each node. You can refer to Fig. 3.1

In order to ensure communication between DistAlgo processes, the receiver's process identifier is required when you attempt to send a message. For this reason, the initiator must send all the required modules to each abstraction to ensure successful communication. Assuming the example of a perfect failure detector, a system with 3 nodes $\{p, q, r\}$ would look like this after the initialization phase.

```
{
    "p": {"pfd": <distalgo_pid>, "pl": <distalgo_pid>, "sl": <distalgo_pid>},
    "q": {"pfd": <distalgo_pid>, "pl": <distalgo_pid>, "sl": <distalgo_pid>},
    "r": {"pfd": <distalgo_pid>, "pl": <distalgo_pid>, "sl": <distalgo_pid>}
}
```

The system is represented by a dictionary of dictionaries where each node contains a DistAlgo process identifier for each module. Once created, each module is given a name based on the node it is running on. It is then relatively easy to communicate with any abstraction using the node's name and the module. The initiator finally sends each abstraction the software stack to which it belongs along with the list of nodes in the system.

Each implemented abstraction must also add a receive clause, handling the delivery of its required modules from the initiator. This marks the end of the initialization phase.

```
def receive(msg=('REQUIREMENTS', requirements), from_=p):
    self.stack = requirements['stack']
    self.procs = requirements['procs']
```

**Cryptographic abstractions.** When the algorithm chosen by the user requires certain cryptographic abstractions, the initiator also generates and distributes the symmetric or private/public keys to the different modules of the system.

**Inter-modules communication.** The message-passing model is used to ensure communication between DistAlgo processes and to trigger certain events in modules. Unlike conventional programming, the DistAlgo library does not allow direct invoking of functions of certain modules. For instance, the *Send* event of the stubborn link abstraction can be triggered as follows.

```
    send(('Send', path, tag, m), to=node_sl)
```

The first parameter of the tuple represents the type of event triggered. The *path* corresponds to an array containing the whole stack of modules through which the message has passed, it is used to know which module is the final recipient of a message. The third parameter *tag* is used in more complex abstractions to differentiate messages of the same type. Finally, the last parameter is the message $m$ itself. It is a class with multiple parameters such as the original sender of the message or the recipient which are required for the proper functioning of the different algorithms.

Fig. 3.1 shows an example of how a message is processed in the module stack. The perfect failure detector abstraction (PFD) on node $p$ wants to send a message to the PFD on node $q$. First, the PFD forwards the message to its own perfect link (PL) abstraction, taking care to add "pfd" to the $path$. The PL then adds its own module "pl" to the $path$ and transmits the message to its own stubborn link (SL). Then, the SL which is at the bottom of the stack sends the message to the SL on the node $q$. Upon arriving to

the destination module, the message is transmitted to the first module in the path until it becomes empty meaning that the desired destination is reached.

In summary, the *path* variable is modified each time the message visits a new module in the stack. Once the last module of the stack is reached (*SLp*), the *path* looks like this: *[pfd, pl]*. Then, *SLp* sends the message to *SLq*. *SLq* receives the message and reverses the path as follows: *[pl, pfd]*. Now, *SLq* knows that the next module to receive the message is *pl*. This operation is performed until the last module in the array is reached.

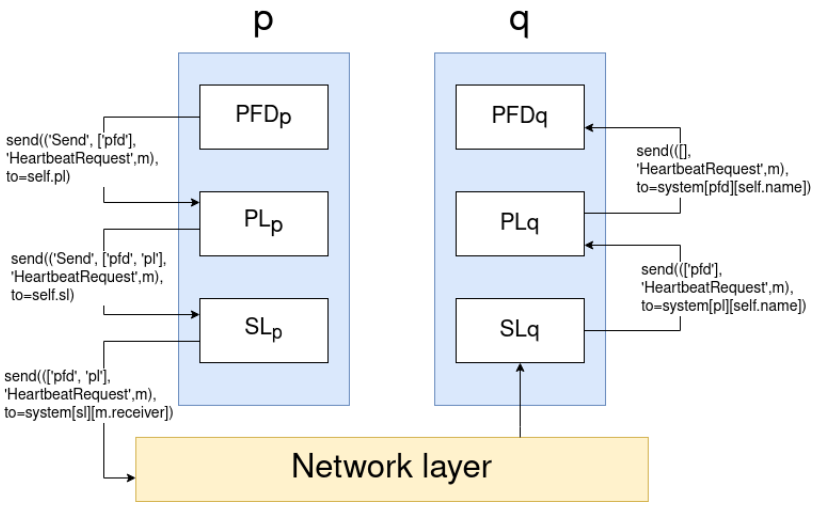


**Figure 3.1.** Processing of a message across multiple modules, showing how a message transits between the different modules of the same node and within the modules of the destination node.

**Message class.** We observed during the development of the different algorithms that the more the complexity and modularity increased, the more the number of variables needed to process a message varied between the different algorithms. Since a message sent from the top of the stack must pass through all the modules below, the receive handler of each abstraction must match the message. For this reason, a message class gathering all the necessary information has been created.

**Broadcast class.** Following the logic of the message class, a broadcast class has also been created.

**Global configuration.** The user has the possibility to define some global parameters used in several algorithms. The *config.ini* file contains the different customizable configurations. This includes things like the retransmission interval for stubborn links or the default reliable broadcast for example.

```
[DEFAULT]
StubbornLinksRetransmit = 2
PerfectFailureDetectorTimeOut = 3
DefaultPerfectLinks = pl_optimized
DefaultReliableBroadcast = rb_eager
DefaultUniformReliableBroadcast = urb_aa
DefaultByzantineConsistentBroadcast = bcb_seb
```

## 3.3 Communication Links

### 3.3.1 Stubborn Links

Due to the unreliability of the network and the potential loss of messages, the first abstraction ensures that messages are continuously retransmitted until they eventually reach their destination. The *stubborn links* abstraction can be defined by two properties. The *stubborn delivery* property implies that each message sent over the link is delivered to the recipient an unbounded number of times. The *no creation* property prevents the link from making up messages out of nowhere. The stubborn links interface is represented in Module 1 [1].

---

**Module 1 :** Interface and properties of stubborn point-to-point links

---

**Module:**

**Name:** StubbornPointToPointLinks, **instance** *sl*.

**Events:**

**Request:** $\langle$ *sl*, *Send* | $q$, $m$ $\rangle$: Requests to send message $m$ to process $q$

**Indication:** $\langle$ *sl*, *Deliver* | $p$, $m$ $\rangle$: Delivers message $m$ sent by process $p$.

**Properties:**

**SL1:** *Stubborn delivery:* If a correct process $p$ sends a message $m$ once to a correct process $q$, then $q$ delivers $m$ an infinite number of times.

**SL2:** *No creation:* If some process $q$ delivers a message $m$ with sender $p$, then $m$ was previously sent to $q$ by process $p$.

---

This first algorithm simply keeps on retransmitting all the messages sent through the stubborn links whenever a timeout event is triggered. It is very unlikely that all messages exchanged between two processes will be lost, unless there is a serious network failure.

**Run method.** The *run* method is started at the creation of the DistAlgo process. It loads the retransmission interval from the global configuration file and ensures the constant retransmission of the messages.

```
def run():
    self.retransmit_interval = int(get_config_value('StubbornLinksRetransmit'))
    retransmit()

def retransmit():
    for (path, tag, msg, receiver) in self.msgs:
        send((path, tag, msg), to=receiver)
    threading.Timer(self.retransmit_interval, retransmit).start() #restart
```

**Send event.** When the send event is triggered in the stubborn link, the message is made unique using a local counter and is then stored in the retransmission array. Finally, the message is forwarded to the receiver's node stubborn link module.

Later, when broadcast abstractions are presented, it will be shown that some messages may be retransmitted multiple times to ensure the accuracy of certain algorithms. For this reason, this algorithms check the value of the *broadcast_retransmit* field in order to prevent the storage of duplicate messages.

```
def receive(msg=('Send', path, tag, m), from_=p):
    receiver = self.procs[m.receiver]
```

```
    if m.msgid == None:
        self.counter += 1
        m.msgid = self.counter

    if not m.broadcast_retransmit:
        msgs.append((path, tag, m, receiver))

    send((path, tag, m), to=receiver)
```

**Deliver event.** When a message is delivered, the algorithm first verifies whether the path is empty as explained in Fig. 3.1. As stubborn links are the lowest abstraction in the module stack, the path is reversed to get the next module destination and forward the message.

```
def receive(msg=(path, tag, m), from_=p):
    if path == []:
        output("SL : ", m.msg, " --> FROM ", m.sender)
    else :
        path.reverse()
        send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
```

### 3.3.2 Perfect Links

Perfect links add a mechanism for detecting and suppressing message duplicates. The specification of this abstraction is presented in Module 2 [1]. The *reliable delivery* property ensures that messages sent to a correct process are eventually delivered. The *no duplication* property guarantees that a message is delivered by the receiver exactly once. Finally, the *no creation* property is the same as in the stubborn link abstraction.

---

**Module 2 :** Interface and properties of perfect point-to-point links

---

**Module:**

**Name:** PerfectPointToPointLinks, **instance** $pl$.

**Events:**

**Request:** $\langle$ *pl*, *Send* $\mid$ $q$, $m$ $\rangle$: Requests to send message $m$ to process $q$

**Indication:** $\langle$ *pl*, *Deliver* $\mid$ $p$, $m$ $\rangle$: Delivers message $m$ sent by process $p$.

**Properties:**

**PL1:** *Reliable delivery:* If a correct process $p$ sends a message $m$ to a correct process $q$, then $q$ eventually delivers $m$.

**PL2:** *No duplication:* No message is delivered by a process more than once.

**PL3:** *No creation:* If some process $q$ delivers a message $m$ with sender $p$, then $m$ was previously sent to $q$ by process $p$.

---

This algorithm relies on the stubborn link abstraction. Its main objective is to remove duplicate messages by keeping trace of all messages that have already been delivered. Each time a new message arrives, it is delivered only if it is not a duplicate.

**Send event.** The *send* event simply adds $pl$ in the path array and passes the message to its own stubborn link abstraction.

```
def receive(msg=('Send', path, tag, m), from_=p):
    path.append('pl')
    send(('Send', path, tag, m), to=self.sl)
```

**Deliver event.** The *no duplication* property is handled in the *deliver* event. When a message is received, the algorithms checks whether the message has already been delivered. If it is not the case, the message is added to the $delivered$ array and forwarded to the next module.

In the case that the message is a broadcast retransmission, the message is simply forwarded to the next module and must not be considered as a duplicate.

```
def receive(msg=(path, tag, m), from_=p):
    if m.broadcast_retransmit:
        send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
        return

    if (m.msgid, m.msg, m.sender) not in delivered:
        delivered.append((m.msgid, m.msg, m.sender))
        if path == []:
            output("PL : ", m.msg, " --> FROM ", m.sender)
        else:
            send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
```

### 3.3.3 Perfect Links Optimized

An optimized version of perfect links has also been implemented that does not depend on stubborn links but uses the native DistAlgo reliable send primitive. It keeps the same properties as the normal version but has less overhead and memory usage since messages are no longer sent continuously by the stubborn link.

**Send Event** The send event forwards the message to the receiver node using the send primitive.

```
def receive(msg=('Send', path, tag, m), from_=p):
    receiver = self.procs[m.receiver]
    send((path, tag, m), to=receiver)
```

**Receive Event** In this version, the receiving event does not need to check if the message has already been delivered thanks to the reliability of the DistAlgo primitive. The message is forwarded if this module is not the final receiver.

```
def receive(msg=(path, tag, m), from_=p):
    if path == []:
        output("PL_OPTIMIZED : ", m.msg, " --> FROM ", m.sender)
    else :
        path.reverse()
        send((path[1:], tag, m), to=self.stack[path[0]])
```

The user has the possibility to choose the version of perfect links to be used by modifying the *config.ini* file.

### 3.3.4 Authenticated Perfect Links

Authenticated perfect links consider messages transmitted within the fail-arbitrary class of algorithms where Byzantine processes may behave arbitrarily. Cryptographic authentication can prevent the forgery

of messages on the links between two correct processes. The interface and properties described in Module 3 [1], is quite similar to the perfect links except that the *authenticity* is stronger than the corresponding *no creation* property.

**Module 3 :** Interface and properties of authenticated perfect point-to-point links

**Module:**

**Name:** AuthPerfectPointToPointLinks, **instance** $al$.

**Events:**

**Request:** $\langle$ $al$, *Send* | $q$, $m$ $\rangle$: Requests to send message $m$ to process $q$

**Indication:** $\langle$ $al$, *Deliver* | $p$, $m$ $\rangle$: Delivers message $m$ sent by process $p$.

**Properties:**

**AL1:** *Reliable delivery:* If a correct process sends a message $m$ to a correct process $q$, then $q$ eventually delivers $m$.

**AL2:** *No duplication:* No message is delivered by a process more than once.

**AL3:** *Authenticity:* If some correct process $q$ delivers a message $m$ with sender $p$ and process $p$ is correct, then $m$ was previously sent to $q$ by $p$.

This abstraction uses the stubborn links and filters duplicate messages. Each time a message is sent, an authenticator is computed using the *MAC*. When a message is received, the algorithm verifies that the message is valid according to the authenticator.

A shared symmetric key is required for each pair of processes. Therefore the initiator will generate those keys and distribute them to each process during the initialization phase.

**Key generation.** The Python *secrets* module is used to create the shared symmetric keys.

```
def generate_symm_keys(al_procs):
    keys = dict()
    #create pair of procs
    combinations = itertools.combinations_with_replacement(al_procs, 2)
    for c in combinations:
        keys[c] = secrets.token_bytes(16)
    return keys
```

**Key distribution.** The initiator sends the generated keys with the corresponding process identifier to each process. A receive handler is also required in the authenticated link abstraction to store the received keys. We assume that the implementation of DistAlgo uses secure links to implement this across hosts.

```
#key distribution by the initiator
def send_symm_keys():
    for procs in self.symm_keys:
        #send key to both parties
        send(('KEY', procs[1], self.symm_keys[procs]), to=self.system['al'][procs[0]])
        send(('KEY', procs[0], self.symm_keys[procs]), to=self.system['al'][procs[1]])

#receive handler in the al
def receive(msg=('KEY', name, key), from_=p):
    self.keys[name] = key
```

**Send event.** The send event first computes the authenticator and appends it to the message. Then, 'al' is added to the path and the message is forwarded to the stubborn link.

```
def receive(msg=('Send', path, tag, m), from_=p):
    m.auth = cryptoc.authenticate(self.keys[m.receiver], m.sender, m.receiver, m.msg)
    path.append('al')
    send(('Send', path, tag, m), to=self.sl)
```

**Deliver event.** The deliver event is the same as for perfect links except that the receiver also checks whether the received authenticator is valid.

As in the perfect link abstraction, the algorithm checks whether the message is a broadcast retransmission and simply passes it to the next module if the condition holds.

```
def receive(msg=(path, tag, m), from_=p):
    valid = cryptoc.verifyauth(self.keys[m.sender],m.receiver,m.sender,m.msg,m.auth)
    if valid and m.broadcast_retransmit:
        send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
        return

    if  valid and (m.msgid, m.msg, m.sender) not in delivered :
        delivered.append((m.msgid, m.msg, m.sender))
        if path == []:
            output("APL : ", m.msg, " --> FROM ", m.sender)
        else:
            send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
```

## 3.4 Failure Detection

The *failure detector* abstraction provides information about which processes have crashed and which are correct.

### 3.4.1 Perfect Failure Detection

In a synchronous system with a crash-stop process abstraction, crashes can be accurately detected using *timeouts*. The detection of a crash is realized by asking the processes to periodically send a *heartbeat* to prove that they are still alive. A perfect failure detector is characterized by two properties presented in Module 4 [1]: the *strong completeness* ensures that all crashed processes have been detected and the *strong accuracy* guarantees that the detector never detects a non-crashed process.

This algorithm uses perfect links to ensure the reliable delivery of the messages sent among correct processes. The goal is to exchange heartbeat messages among all processes within a certain amount of time. When the *timeout* event expires, all processes from which no reply has been received are declared to have crashed. Finally, a new round of heartbeat exchange is started.

**Client registration.** Depending on the algorithm, the perfect failure detector might be used by multiple abstractions. It is then necessary to first register all those clients to be able to notify them whenever a process crashes.

```
def receive(msg=('CLIENT', module), from_=p):
    self.clients.append(module)
```

**Module 4 :** Interface and properties of the perfect failure detector

**Module:**

**Name:** PerfectFailureDetector, **instance** $\mathcal{P}$.

**Events:**

**Indication:** $\langle\ \mathcal{P}, \mathit{Crash} \mid p\ \rangle$: Detects that process $p$ has crashed.

**Properties:**

**PFD1:** *Strong completeness:* Eventually, every process that crashes is permanently detected by every correct process.

**PFD2:** *Strong accuracy:* If a process $p$ is detected by any process, then $p$ has crashed

---

**Timeout event.** The duration of the timeout can be configured in the *config.ini* file. For each process in the system, the algorithms checks whether the process is in the *alive* set or in the *detected* set. If that is not the case, the process is considered as newly crashed. After the detection phase, another round is started by sending a *HeartbeatRequest* to all processes.

```
def detect():
    #detection phase
    for proc in self.procs:
        if proc not in alive and proc not in detected:
            detected.append(proc)
            notify_crash(proc)
    #start new round
    self.alive = []
    for receiver in self.procs:
        m = message.Message('', self.name, self.name, receiver)
        send(('Send', ['pfd'], 'HeartbeatRequest', m), to=self.pl)
    #start timer
    threading.Timer(self.timeout, detect).start()
```

**Deliver HeartbeatRequest.** When a *HeartbeatRequest* is received from some process, a *HeartbeatReply* is sent back using the perfect links.

```
def receive(msg=(path, 'HeartbeatRequest', m), from_=p):
    msg = message.Message('', self.name, self.name, m.sender)
    send(('Send', ['pfd'], 'HeartbeatReply', msg), to=self.pl)
```

**Deliver HeartbeatReply.** The delivery of a *HeartbeatReply* message ensures that the sending process is alive and can therefore be added to the alive set.

```
def receive(msg=(path, 'HeartbeatReply', m), from_=p):
    self.alive.append(m.sender)
```

### 3.4.2 Leader Election

The *leader election* abstraction provides a *leader* process that acts as a coordinator in a distributed system. This can be useful when some leader handles all the requests from a client and then updates the backups in a database system. If the leader is detected as crashed, a new round of election is held.

This abstraction is defined with a single indication event $\langle$ *Leader* | $p$ $\rangle$ where $p$ is the elected leader until he crashes. The properties of this abstraction are described in Module 5 [1]. The *eventual detection* ensures that a leader is eventually detected unless all processes have crashed. Finally, the *accuracy* states that a leader may only change if the previous leader has crashed.

**Module 5 :** Interface and properties of perfect failure detector

**Module:**

**Name:** LeaderElection, **instance** *le*.

**Events:**

**Indication:** $\langle$ *le*, *Leader* | $p$ $\rangle$: Indicates that process $p$ is elected as leader.

**Properties:**

**LE1:** *Eventual Detection:* Either there is no correct process, or some correct process is eventually elected as leader

**LE2:** *Accuracy:* If a process is leader, then all previously elected leaders have crashed.

The leader election algorithm elects a leader according to the ranking of processes and uses a perfect failure detector to detect crashes. The initial leader is the process with the highest ranking in the system. Therefore, a process can become leader only if all processes with a higher rank have crashed. An internal event checks periodically whether the process with the highest rank among the correct ones is the leader.

**Leader change.** An await statement is used to check whether the leader is still the process with the highest rank. This can be easily done using the aggregation operator *max*.

```
def run():
    while(True):
        await(self.leader != max(self.procs.values()))
        self.leader = max(self.procs.values())
```

**Crash event.** When a crash is indicated by the perfect failure detector, the process is removed from the correct set.

```
def receive(msg=('Crash', name_crashed), from_=p):
    del self.procs[name_crashed]
```

## 3.5 Broadcast

*Broadcast* abstractions are used to disseminate information among a set of processes. The different broadcast abstractions studied differ in the reliability of the delivery.

### 3.5.1 Best-Effort Broadcast

The weakest form of broadcast is the *best-effort broadcast* where only the sender ensures the reliability. Therefore, no guarantee of delivery is offered if the sender fails. This abstraction is characterized by the following three properties described in Module 6 [1].

**Module 6 :** Interface and properties of best-effort broadcast

**Module:**

**Name:** BestEffortBroadcast, **instance** *beb*.

**Events:**

**Request:** $\langle$ *beb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.

**Indication:** $\langle$ *beb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**BEB1:** *Validity:* If a correct process broadcasts a message $m$, then every correct process eventually delivers $m$.

**BEB2:** *No duplication:* No message is delivered more than once.
to $q$ by process $p$.

**BEB3:** *No creation:* If a process delivers a message $m$ with sender $s$, then $m$ was previously broadcast by process $s$.

**Broadcast event.** The *broadcast* event sends the message using the perfect links to every process in the system. The broadcast object received in the parameters is converted into a message for each receiver. In case of a broadcast retransmission situation, the identifier is not updated because it already has its own identifier created during the initial sending.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    path.append('beb')
    if not b.broadcast_retransmit:
        self.counter += 1
        b.msgid = self.counter
    for receiver in list(self.procs.keys()):
        m = message.Message(b.msg, b.original_sender, self.name, receiver, args=b.args,
                            msgid=b.msgid, broadcast_retransmit=b.broadcast_retransmit)
        send(('Send', path, tag, m), to=self.pl)
```

**Deliver event.** The *deliver* event forwards the message to the next module if it is not the intended receiver.

```
def receive(msg=(path, tag, m), from_=p):
    if path == []:
        output("BEB : ", m.msg, " --> FROM ", m.sender)
    else:
        send((path[1:], tag, m), to=self.system[path[0]][m.receiver])
```

### 3.5.2 Reliable Broadcast

*Reliable broadcast* algorithms ensure that the correct processes agree on all the messages they deliver, even when the sender of those messages crashes during the transmission. If the sender crashes before he can send a message, no process will deliver it. The interface and properties of this abstraction is described in the Module 7 [1]. The *agreement property* differentiates this abstraction from the best-effort broadcast because either no message is delivered or each correct process eventually delivers the message.

Two versions of the reliable broadcast are implemented: the *lazy reliable broadcast* in a fail-stop model relying on the perfect failure detector abstraction and the *eager reliable broadcast* in the fail-silent class of

**Module 7 :** Interface and properties of reliable broadcast

**Module:**

**Name:** ReliableBroadcast, **instance** *rb*.

**Events:**

**Request:** $\langle$ *rb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.

**Indication:** $\langle$ *rb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**RB1:** *Validity:* If a correct process $p$ broadcasts a message $m$, then $p$ eventually delivers $m$.

**RB2:** *No duplication:* No message is delivered more than once.

**RB3:** *No creation:* If a process delivers a message $m$ with sender $s$, then $m$ was previously broadcast by process $s$.

**RB4:** *Agreement:* If a message $m$ is delivered by some correct process, then $m$ is eventually delivered by every correct process.

algorithm without a perfect failure detector. The global configuration file allows the user to choose which version of the reliable broadcast to use by default.

**Lazy Reliable Broadcast.** This abstraction uses the perfect failure detector and the best-effort broadcast to disseminate messages. In order to respect the agreement property, each process retransmits messages received from a crashed process to ensure that each correct process delivers the same set of messages. Moreover, when delivering a message the algorithm also checks that the sender process has not crashed in the meantime to relay the messages if necessary.

The *broadcast_retansmit* class variable mentioned in the previous sections must be set to True when a message is retransmitted. Otherwise, the *no duplication* property of perfect links will consider this message as a duplicate.

**Broadcast event.** The broadcast event triggers the best-effort *Broadcast* event and adds itself to the *path* variable.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    path.append('rb_lazy')
    send(('Broadcast', path, tag, b), to=self.beb)
```

**Deliver event.** The *deliver* event is divided into two parts. The first part checks whether the message has already been delivered in the case of a retransmission. The second checks that the sender has not crashed while the message was being sent, resulting in a potential loss of messages.

```
def receive(msg=(path, tag, m), from_=p):
    #check if msg already received
    msg = (m.msgid, m.msg, m.original_sender, m.args, tag, path)
    if msg not in rcv_from[m.original_sender]:
        rcv_from[m.original_sender].append(msg)
        if path == []:
            output("RB_LAZY : ", m.msg, " --> FROM ", m.original_sender)
        else:
            send((path[1:], m), to=system[path[0]][m.receiver])
```

```
        #check if sender is correct to retransmit
        if m.original_sender not in correct:
            new_path = path.copy()
            new_path.append('rb_lazy')
            b = broadcast.Broadcast(m.msg,m.original_sender,msgid=m.msgid,args=m.args,
                                    broadcast_retransmit=True)
            send(('Broadcast', new_path, tag, b), to=self.beb)
```

**Crash event.** When a crash is detected, all the received messages from the crashed process are retransmitted.

```
def receive(msg=('Crash', name_crashed), from_=p):
    correct.remove(name_crashed)
    if name_crashed in self.rcv_from.keys():
        for (msgid,msg,original_sender,args,tag,path) in self.rcv_from[name_crashed]:
            new_path = path.copy()
            new_path.append('rb_lazy')
            b = broadcast.Broadcast(msg, original_sender, msgid=msgid, args=args,
                                    broadcast_retransmit=True)
            send(('Broadcast', new_path, tag, b), to=self.beb)
```

**Eager Reliable Broadcast.** This version of reliable broadcast requires only the best-effort broadcast abstraction. The *agreement* property is guaranteed by relaying each message as soon as it is received.

**Broadcast event.** The broadcast event is similar to the lazy version, except that the *rb_eager* module is added to the path.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    path.append('rb_eager')
    send(('Broadcast', path, tag, b), to=self.beb)
```

**Deliver event.** The *deliver* event first verifies that the message has not already been delivered and then immediately retransmits it to everyone.

```
def receive(msg=(path, tag, m), from_=p):
    if (m.msgid, m.msg, m.original_sender) not in delivered:
        delivered.append((m.msgid, m.msg, m.original_sender))
        if path == []:
            output("RB_EAGER : ", m.msg, " --> FROM ", m.sender)
        else:
            send((path[1:], tag, m), to=system[path[0]][m.receiver])

        #retransmit broadcast msg
        path.append('rb_eager')
        b = broadcast.Broadcast(m.msg, m.original_sender, msgid=m.msgid, args=m.args,
                                broadcast_retransmit=True)
        send(('Broadcast', path, tag, b), to=self.beb)
```

### 3.5.3 Uniform Reliable Broadcast

The *uniform reliable broadcast* is a stronger form of reliable broadcast because it ensures that the messages delivered by *faulty* processes are always a subset of the messages delivered by correct processes. The interface and properties are described in Module 8 [1].

**Module 8 :** Interface and properties of uniform reliable broadcast

**Module:**

**Name:** UniformReliableBroadcast, **instance** *urb*.

**Events:**

**Request:** $\langle$ *urb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.

**Indication:** $\langle$ *urb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**URB1-URB3:** Same as properties RB1-RB3 in reliable broadcast (Module 7).

**URB4:** *Uniform agreement:* If a message $m$ is delivered by some process (whether correct or faulty), then $m$ is eventually delivered by every correct process.

---

The Lazy Reliable Broadcast and Eager Reliable Broadcast do not ensure the *uniform agreement* because a process might deliver a message and then crash before being able to relay it. In the uniform version, a process must deliver a message only when it knows that the message has been seen by all correct processes.

Two versions of this abstraction have been developed. The All-Ack Uniform Reliable Broadcast is part of the fail-stop class of algorithms and uses a perfect failure detector and a best-effort broadcast. On the other hand, the Majority-Ack Uniform Reliable Broadcast belongs to the fail-silent algorithms and doesn't rely on a perfect failure detector but waits until a majority of correct processes has seen and retransmitted the message.

**All-Ack Uniform Reliable Broadcast.** In order to ensure the *uniform agreement*, all processes relay the message once they have seen it. Each process keeps track of the processes from which it received the message, either the original or the relayed version. Therefore, a process knows that it can deliver a message when it has received a message from all the correct processes.

**Broadcast event.** When broadcasting a message, it is added to the pending set and then forwarded to the underlying best-effort abstraction.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    pending.append((b.original_sender, (tag, b.msg, path.copy())))
    path.append('urb_aa')
    send(('Broadcast', path, tag, b), to=self.beb)
```

**Crash event.** When a process crashes, it is removed from the correct set.

```
def receive(msg=('Crash', name_crashed), from_=p):
    correct.remove(name_crashed)
```

**Deliver event.** When a message is delivered, the sender is first added to the set of processes that saw the message. Then, if the message is not in the pending set, it is added and then retransmitted to all processes.

```
def receive(msg=(path, tag, m), from_=p):
    self.ack[m.msg].append(m.sender)
    if(m.original_sender, (tag, m.msg, path)) not in self.pending:
```

```
        pending.append((m.original_sender, (tag, m.msg, path)))
        new_path = path.copy()
        new_path.append('urb_aa')
        b = broadcast.Broadcast(m.msg, m.original_sender, msgid=m.msgid, args=m.args,
                                broadcast_retransmit=True)
        send(('Broadcast', new_path, tag, b), to=self.beb)
```

The run method of this module constantly checks if there are some messages that can be delivered. This can happen when every correct process has seen the message.

```
#check that every correct has seen the message
def candeliver(m):
    for c in correct:
        if c not in ack[m]:
            return False
    return True
def run():
    while(True):
        #go through all messages and check if can deliver
        for (s, (tag, m, path)) in pending:
            await(candeliver(m) and (s, m) not in delivered)
            delivered.append((s, m))
            msg = message.Message(m, s, s, self.name)
            if path == []:
                output("URB_ALL_ACK : ", m, " --> FROM ", s)
            else:
                send((path[1:], tag, msg), to=system[path[0]][self.name])
```

**Majority-Ack Uniform Reliable Broadcast.** As this version assumes that a majority quorum has relayed the message before delivering it, the *candeliver* method needs to be changed compared to the All-Ack algorithm.

```
def candeliver(m):
    N = len(list(self.procs.keys()))
    return len(ack[m]) > N/2
```

### 3.5.4 First-In First-Out Reliable Broadcast

The previous broadcast versions offer no guarantee of ordering among the messages delivered by different processes. The *first-in first-out (FIFO) order* ensures that messages from the same sender are delivered in the order in which they were sent. However, this does not affect messages from different senders.

The properties presented in Module 9 [1] are an extension of the Reliable Broadcast with the *FIFO delivery* property in addition.

The FIFOReliableBroadcast implementation maintains a sequence number per sender attached to the message during broadcast. Processes buffer all received messages and deliver them according to the sender's sequence number.

**Broadcast event.** The broadcast event increments the sequence number and sends it with the message to its reliable broadcast abstraction.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    path.append('frb')
    lsn += 1
    b.args = lsn
    send(('Broadcast', path, tag, b), to=self.rb)
```

**Module 9 :** Interface and properties of FIFO-order reliable broadcast

**Module:**

**Name:** FIFOReliableBroadcast, **instance** *frb*.

**Events:**

**Request:** $\langle$ *frb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.
**Indication:** $\langle$ *frb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**FRB1-FRB4:** Same as properties RB1-RB4 in reliable broadcast (Module 7).
**FRB5:** *FIFO delivery:* If some process broadcasts message $m1$ before it broadcasts message $m2$, then no correct process delivers $m2$ unless it has already delivered $m1$.

---

**Deliver event.** Each received message is added to a pending set with the sequence number and is then delivered in the appropriate order.

```
def receive(msg=(path, tag, m), from_=p):
    pending.append((tag, m, path))
    exist = True
    while(exist):
        exist = False
        for (msg_tag, msg, p_path) in pending:
            if msg.args == next_msg[msg.original_sender]:
                exist = True
                next_msg[msg.original_sender] += 1
                pending.remove((msg_tag, msg, p_path))
                if p_path == []:
                    output("FRB : ", msg.msg, " --> FROM ", msg.original_sender)
                else:
                    send((path[1:],msg_tag, msg),to=system[p_path[0]][msg.receiver])
```

### 3.5.5 Causal-Order Reliable Broadcast

The *causal order* property can be defined as follows: if a message $m1$ has caused another message $m2$, then every process first delivers $m1$ and then $m2$. This means that when a message is delivered, all messages that causally precede it have already been delivered. The properties presented in Module 10 [1] are an extension of the Reliable Broadcast with the *causal delivery* property in addition.

The causal-order implementation uses the reliable broadcast abstraction. The causal ordering is represented using a vector clock $V$ of integers. Each process $p$ maintains its vector clock such that entry $V[rank(q)]$ denotes the number of messages that $p$ has delivered from $q$.

**Broadcast event.** Each time a process broadcasts a message, its vector clock is updated and sent along with the message.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    W = self.V.copy()
    W[self.name] = self.lsn
    self.lsn += 1
    b.args = W
    path.append('crb')
    send(('Broadcast', path, tag, b), to=self.rb)
```

**Module 10 :** Interface and properties of causal-order reliable broadcast

**Module:**

**Name:** CausalOrderReliableBroadcast, **instance** *crb*.

**Events:**

**Request:** $\langle$ *crb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.

**Indication:** $\langle$ *crb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**CRB1-CRB4:** Same as properties RB1-RB4 in reliable broadcast (Module 7).

**CRB5:** *Causal delivery:* For any message $m1$ that potentially caused a message $m2$, i.e. $m1 \rightarrow m2$, no process delivers $m2$ unless it has already delivered $m1$.

**Deliver event.** Received messages are stored in a queue. Messages are then delivered as soon as the vector clock of the received message is smaller than the receiver process vector clock. A vector clock $V$ is considered smaller than some vector clock $W$ if for every $i = 1, ..., N$ we have $V[i] \leqslant W[i]$.

```
def receive(msg=(path, tag, m), from_=p):
    pending.append((tag, m, path))
    exist = True
    while(exist):
        exist = False
        for (msg_tag, msg, p_path) in pending:
            if is_smaller_vector(msg.args):
                exist = True
                pending.remove((msg_tag, msg, p_path))
                V[msg.original_sender] += 1
                if p_path == []:
                    output("CRB : ", msg.msg, " --> FROM ", msg.original_sender)
                else:
                    send((p_path[1:],msg_tag,msg),to=system[p_path[0]][msg.receiver])
```

### 3.5.6 Byzantine Consistent Broadcast

A broadcast abstraction in the fail-arbitrary model with Byzantine processes is now described. The approach here consists of having one instance of broadcast as an abstraction of its own. *Byzantine broadcast channels* provide an equivalent to the reliable broadcast abstraction in the crash-stop model by using multiple instances of such byzantine consistent broadcast. Byzantine broadcast channels have not been implemented in this thesis but can be found in [1].

Every instance of consistent broadcast has a designated sender $s$, who is the only process able to broadcast a message $m$. If the sender is correct, then every correct process will eventually deliver the message. Otherwise, either each correct process delivers the same message, or no message is delivered at all. The interface and properties of the Byzantine consistent broadcast abstraction are described in Module 11 [1]

Two versions of this abstraction have been implemented: Authenticated Echo Broadcast that relies on authenticated perfect links and uses Byzantine quorums to ensure *consistency*. The second version, called Signed Echo Broadcast, also uses authenticated perfect links and digital signatures. In addition, Byzantine consistent broadcast abstraction requires that the number of faulty processes satisfies $f < N/3$.

**Module 11 :** Interface and properties of Byzantine consistent broadcast

**Module:**

**Name:** ByzantineConsistentBroadcast, **instance** $bcb$, with sender $s$.

**Events:**

**Request:** $\langle$ $bcb$, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes. Executed only by process $s$.

**Indication:** $\langle$ $bcb$, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**BCB1:** *Validity:* If a correct process $p$ broadcasts a message $m$, then every correct process eventually delivers m.

**BCB2:** *No duplication:* Every correct process delivers at most one message

**BCB3:** *Integrity:* If some correct process delivers a message $m$ with sender $p$ and process $p$ is correct, then $m$ was previously broadcast by $p$.

**BCB3:** *Consistency:* If some correct process delivers a message $m$ and another correct process a message $m'$, then $m = m'$.

---

**Authenticated Echo Broadcast.** This algorithm uses two rounds of messages. First, the sender $s$ disseminates the message to all processes using the authenticated perfect links with the *SEND* tag. The second part consists of an *ECHO* round where all processes acting as *witnesses*, retransmit the message to everyone. Therefore, when a process receives more than $(N + f)/2$ such ECHO messages with the same message $m$, it can deliver the message.

**Broadcast SEND message.** The broadcast event verifies that the process is actually the sender, and then propagates the message with its own authenticated perfect links abstraction.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    if self.name != self.sender:
        output("Only sender ", self.sender, "can send messages")
        return
    path.append('bcb_aeb')
    self.counter += 1
    b.msgid = self.counter
    b.args = tag
    for receiver in list(self.procs.keys()):
        m = message.Message(b.msg, b.original_sender, self.name, receiver, args=b.args,
                            msgid=b.msgid, broadcast_retransmit=b.broadcast_retransmit)
        send(('Send', path, 'SEND', m), to=self.al)
```

**Deliver SEND message.** When a process delivers the message with the SEND tag, it relays the message with the ECHO tag if he has not done it yet.

```
def receive(msg=(path, 'SEND', m), from_=p):
    if m.original_sender == self.sender and not self.sent_echo:
        path.append('bcb_aeb')
        for receiver in list(self.procs.keys()):
            m = message.Message(m.msg, m.original_sender, self.name, receiver,
                        args=m.args, msgid=m.msgid, broadcast_retransmit=True)
            send(('Send', path, 'ECHO', m), to=self.al)
```

**Deliver ECHO message.** Each ECHO message received is then stored in an array.

```
def receive(msg=(path, 'ECHO', m), from_=p):
    if self.echos[m.sender] == (False, False, []):
        self.echos[m.sender] = (m.msg, m, path)
```

**Deliver event.** The message is finally delivered by all correct processes if the number of echos received is greater than $(N+f)/2$. This is implemented using DistAlgo's built-in comprehension and quantification queries.

```
def run():
    await(not self.delivered and some(m in self.echos.values(),
            has=(m != (False, False, []) and
            len(setof(p, p in list(self.procs.keys()), self.echos[p][0] == m[0]))
            > self.quorum_size)))
    self.delivered = True
    (msg, msg_obj, path) = m
    if path == []:
        output("BCB_AEB : ", msg_obj.msg, " --> FROM ", msg_obj.original_sender)
    else:
        send((path[1:], msg_obj.args, msg_obj), to=system[path[0]][msg_obj.receiver])
```

**Signed Echo Broadcast.** This algorithm is composed of three rounds of communication. The basic idea is the same as the Authenticated Echo Broadcast except that witnesses don't send the ECHO message to all processes. They will sign the received message with their private key and send it back to the sender process who will then collect all signed statements and relay them to all processes as a *FINAL* message. The message is finally delivered if the number of correct signatures is greater than $(N + f)/2$.

Since digital signatures are used, the initiator must first generate the private/public key pair for all nodes present in the system. The Rivest-Shamir-Adleman (RSA) public key cryptography system is used to generate these keys. Then, all public keys with the corresponding process identifier will be distributed to all nodes in the system. Private keys are only known by the process itself.

```
#key generation
def generate_RSA_keys():
    priv = RSA.generate(1024)
    pub = priv.publickey()
    return (priv, pub)

#key distribution
def send_RSA_keys():
    for p in self.system['bcb_seb']:
        (priv, pub) = cryptoc.generate_RSA_keys()
        priv_export = priv.export_key().decode('utf-8')
        send(('PRIVATE_KEY', priv_export), to=self.system['bcb_seb'][p])
        self.public_keys[p] = pub.export_key().decode('utf-8')
    send(('PUBLIC_KEYS', self.public_keys), to=list(self.system['bcb_seb'].values()))
```

**Broadcast SEND message.** The first step is the same as in the Authenticated Echo Broadcast.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    if self.name != self.sender:
        output("Only sender ", self.sender, "can send messages")
        return
    path.append('bcb_seb')
    self.counter += 1
```

```
    b.msgid = self.counter
    b.args = (tag,)
    for receiver in list(self.procs.keys()):
        m = message.Message(b.msg, b.original_sender, self.name, receiver, args=b.args,
                            msgid=b.msgid, broadcast_retransmit=b.broadcast_retransmit)
        send(('Send', path, 'SEND', m), to=self.al)
```

**Deliver SEND message.** When a process delivers the message with the SEND tag, it signs the message with the ECHO tag and sends it back to the sender.

```
def receive(msg=(path, 'SEND', m), from_=p):
    if m.original_sender == self.sender and not self.sent_echo:
        sig = sign(self.private_key, 'bcb'+self.name+"ECHO"+m.msg)
        path.append('bcb_seb')
        m = message.Message(m.msg,m.original_sender,self.name,self.sender,args=m.args,
            msgid=m.msgid, broadcast_retransmit=True, dig_sig=sig)
        send(('Send', path, 'ECHO', m), to=self.al)
```

**Deliver ECHO message.** The sender will store every valid signed statement.

```
def receive(msg=(path, 'ECHO', m), from_=p):
    if verifysig(self.public_keys[m.sender], "bcb"+m.sender+"ECHO"+m.msg, m.dig_sig)
            and self.echos[m.sender] == (False, False, []):
        self.echos[m.sender] = (m.msg, m, path)
        self.sigma[m.sender] = m.dig_sig
```

**Broadcast FINAL message.** If the sender process is able to collect more than $(N+f)/2$ valid signatures of the message, he can send the FINAL message to all processes.

```
def run():
    await(not self.sent_final and some(m in self.echos.values(),
            has=(m != (False, False, []) and
            len(setof(p, p in list(self.procs.keys()), self.echos[p][0] == m[0]))
            > self.quorum_size)))
    self.sent_final = True
    (msg, msg_obj, path) = m
    path.append('bcb_seb')
    msg_obj.args += (self.sigma,)
    for receiver in list(self.procs.keys()):
        msg_obj.receiver = receiver
        send(('Send', path, 'FINAL', msg_obj), to=self.al)
```

**Deliver FINAL message.** Finally, the processes can deliver the message if the number of valid signatures is greater than $(N + f)/2$.

```
def receive(msg=(path, 'FINAL', m), from_=p):
    if not self.delivered and len(setof(p, p in list(self.procs.keys()), m.args[1][p]
        and verifysig(self.public_keys[p], 'bcb'+p+'ECHO'+m.msg, m.args[1][p])))
        > self.quorum_size:
        self.delivered = True
    if path == []:
        output("BCB_SEB : ", m.msg, " --> FROM ", m.original_sender)
    else:
        send((path[1:], m.args[0], m), to=system[path[0]][m.receiver])
```

## 3.6 Consensus

Consensus is one of the most fundamental problems in distributed computing. It can be used when processes need to agree on a common value among the values they initially proposed. There are many forms of consensus but we will only focus on *regular* consensus in the scope of this thesis.

### 3.6.1 Regular Consensus

This abstraction can be defined by two events: *propose* and *decide*. Each process initially proposes a value $v$. In the end, all correct processes have to decide on a value among those proposed. The regular form of the consensus is described in Module 12 [1].

**Module 12 :** Interface and properties of regular consensus

**Module:**

**Name:** Consensus, **instance** $c$.

**Events:**

**Request:** $\langle\ c, \mathit{Propose} \mid v\ \rangle$: Proposes value $v$ for consensus.
**Indication:** $\langle\ c, \mathit{Decide} \mid v\ \rangle$: Outputs a decided value $v$ of consensus.

**Properties:**

**C1:** *Termination:* Every correct process eventually decides some value.
**C2:** *Validity:* If a process decides $v$, then $v$ was proposed by some process.
**C3:** *Integrity:* No process decides twice.
**C4:** *Agreement:* No two correct processes decide differently.

**Flooding Consensus.** This implementation uses a perfect-failure detector abstraction and a best-effort broadcast. The algorithm works in rounds: each process maintains the set of proposed values that it has seen and broadcasts it to all processes as a PROPOSAL message. Each process then merges its own proposal set with the received one. A process can finally decide a value when it has reached a round where all correct processes have seen all the possible values. The chosen value among the set of proposal is a deterministic function chosen in advance. In this implementation, the *min* function taking the lowest value of the set is used.

**Propose.** Initially, each correct process proposes a value $v$ that will be disseminated using the best-effort broadcast abstraction.

```
def receive(msg=('Propose', path, v), from_=p):
    self.proposals[1] = [v]
    path.append('flood_cons')
    b = broadcast.Broadcast('', self.name)
    b.args = (1, self.proposals[1])
    send(('Broadcast', path, 'PROPOSAL', b), to=self.beb)
```

**Deliver PROPOSAL.** When delivering a PROPOSAL message, the process will merge its own proposal set with the received one.

```
def receive(msg=(path, 'PROPOSAL', m), from_=p):
    self.rcv_from[m.args[0]] += [m.original_sender]
    self.proposals[m.args[0]] += m.args[1]
```

Once the propositions have been received from every correct process, the algorithms checks if the same set of proposals has been received during two consecutive rounds. If that holds, a value is decided and a DECIDED message is broadcast, otherwise a new round is started by broadcasting the PROPOSAL message.

```
def run():
    await(set(correct).issubset(set(rcv_from[self.rnd])) and self.decision == None)
    if set(self.rcv_from[self.rnd]) == set(self.rcv_from[(self.rnd-1)]):
        self.decision = min(self.proposals[self.rnd])
        b = broadcast.Broadcast('', self.name)
        b.args = self.decision
        send(('Broadcast', ['flood_cons'], 'DECIDED', b), to=self.beb)
        output("DECIDE : ", decision)
    else:
        self.rnd += 1
        b = broadcast.Broadcast('', self.name)
        b.args = (self.rnd, self.proposals[self.rnd-1])
        send(('Broadcast', ['flood_cons'], 'PROPOSAL', b), to=self.beb)
```

**Deliver DECIDED message.** When a DECIDED message is received, the process can decide the value $v$ attached to the message and broadcast as well the DECIDED message.

```
def receive(msg=(path, 'DECIDED', m), from_=p):
    if m.original_sender in self.correct and self.decision == None:
        self.decision = m.args
        new_path = path.copy()
        new_path.append('flood_cons')
        b = broadcast.Broadcast('', self.name)
        b.args = self.decision
        send(('Broadcast', new_path, 'DECIDED', b), to=self.beb)
        if path == []:
            output("DECIDE : ", decision)
        else:
            send((path[1:], '', m), to=self.system[path[0]][m.receiver])
```

## 3.7 Total-Order Broadcast

The *total-order broadcast* abstraction ensures that messages are delivered in the same order by every process in the system. Indeed, previous implementations of the broadcast had no delivery guarantee for messages sent by different senders or those that were not causally related. The specification illustrated in Module 13 contains the *total order* property compared to the reliable broadcast abstraction [1].

The total-order broadcast implementation uses a reliable broadcast abstraction to disseminate messages and multiple instances of the consensus each time it needs to agree over a sequence of messages. Each time a new message is received, it is stored in an *unordered* set of messages. Then, the consensus is used to agree on a sequence of messages to deliver.

**Broadcast event.** The broadcast event transmits the message to broadcast to the underlying reliable broadcast abstraction.

**Module 13 :** Interface and properties of total-order broadcast

**Module:**

**Name:** TotalOrderBroadcast, **instance** *tob*.

**Events:**

**Request:** $\langle$ *crb*, *Broadcast* | $m$ $\rangle$: Broadcasts a message $m$ to all processes.

**Indication:** $\langle$ *crb*, *Deliver* | $p$, $m$ $\rangle$: Delivers a message $m$ broadcast by process $p$.

**Properties:**

**TOB1:** *Validity:* If a correct process $p$ broadcasts a message $m$, then $p$ eventually delivers $m$.

**TOB2:** *No duplication:* No message is delivered more than once.

**TOB3:** *No creation:* If a process delivers a message $m$ with sender $s$, then $m$ was previously broadcast by process $s$.

**TOB4:** *Agreement:* If a message $m$ is delivered by some correct process, then $m$ is eventually delivered by every correct process.

**TOB5:** *Total order:* Let $m1$ and $m2$ be any two messages and suppose $p$ and $q$ are any two correct processes that deliver $m1$ and $m2$. If $p$ delivers $m1$ before $m2$, then $q$ delivers $m1$ before $m2$.

```
def receive(msg=('Broadcast', path, tag, b), from_=p):
    path.append('tob')
    send(('Broadcast', path, tag, b), to=self.rb)
```

**Unordered messages.** Messages received from the reliable broadcast are stored in a set waiting to be delivered with the help of the consensus.

```
def receive(msg=(path, tag, m), from_=p):
    if (m.msgid, m.original_sender) not in self.delivered:
        unordered[(m.msgid, m.original_sender)] = m.msg
```

**Consensus instance.** Once some messages have been received, a consensus instance is initialized. The total-order abstraction therefore sends a message to the *initiator* asking to initialize some module.

```
def run():
    await(len(list(self.unordered.keys())) and not self.wait)
    self.wait = True
    send(('NEW_INSTANCE', 'flood_cons', self.name, 'tob'), to=self.initiator)
```

On the other hand, the initiator first checks whether the new module requires other abstractions that are not yet present in the software stack and will then update the stack with the relevant modules.

```
def receive(msg=('NEW_INSTANCE', module, name, module_requesting), from_=p):
    #check instance requirements exist
    for requirement in data[module]['modules']:
        if requirement not in system[name].keys():
            m = module_factory(requirement, name, None, None, None)
            self.system[name][requirement] = m
            send(('UPDATE_STACK', requirement, m), to=list(self.system[name].values()))
```

Once all the requirements have been created, the initiator can create the consensus instance and initialize it. Finally, the created DistAlgo process is sent to the total-order abstraction ending the instantiation phase.

```
#create, setup start instance
m = module_factory(module, name, None, None, None)
send(('UPDATE_STACK', module, m), to=list(self.system[name].values()))
#initialize instance
requirements = dict()
requirements['stack'] = system[node]
requirements['procs'] = list(self.system.keys())
send(('REQUIREMENTS', requirements), to=m)
#send instance to module requesting
send(('INSTANCE_CREATED', m), to=self.system[name][module_requesting])
```

Now that the consensus instance has been created, the total order abstraction can start a new round of consensus with its set of unordered messages.

```
def receive(msg=('INSTANCE_CREATED', consensus_instance), from_=self.initiator):
    send(('CLIENT', 'tob', self.rnd), to=consensus_instance)
    send(('Propose', list(), list(self.unordered.values())), to=consensus_instance)
```

**Decide event.** When the consensus has decided some value, the process can then deliver the received set of messages from the consensus instance.

```
def receive(msg=('DECIDED', decision, r), from_=p):
    await(self.rnd == r)
    for m in decision:
        key = list(self.unordered.keys())[list(self.unordered.values()).index(m)]
        delivered[key] = unordered[key]
        del unordered[key]
    self.rnd += 1
    self.wait = False
```

## 3.8 Run Algorithm

Before running an algorithm, make sure you have followed the installation steps in the *README* file to install the correct Python version and the *da* module. Python3.7 is used, which is the latest version supported by DistAlgo [2]. Depending on the type of the algorithm, you have to set some parameters in the command line:

- **Point-to-point communication:** *<algo> <nb_nodes> <sender> <receiver> <"message">*
- **Failure detection:** *<algo> <nb_nodes>*
- **Broadcast:** *<algo> <nb_nodes> <sender> <"message">*
- **Byzantine Broadcast:** *<algo> <nb_nodes> <faulty> <sender> <"message">*
- **Consensus:** *<algo> <nb_nodes> <prop_1> <prop_2> ... <prop_n>*

**Run locally on multiple nodes.** Assume that the perfect failure detector is run on four nodes. One process runs the main method of the program and does the whole initialization phase. Each node is also given a unique name to allow inter-process communication. This is done by passing the -n command line argument. Finally, since only one node should execute the main method, the -D argument must be added to all the other nodes to create them as *idle* nodes.

```
python3.7 -m da -n p run.da pfd 4
python3.7 -m da -n q -D run.da
python3.7 -m da -n r -D run.da
python3.7 -m da -n s -D run.da
```

**Run on remote nodes.** To run an algorithm remotely, each node must receive a name and an IP address using the command line argument -H. In addition, when creating nodes in the code, the *at* parameter must be defined as follows: <node_name>@<ip_address>. The following example shows you how to create two remote nodes in DistAlgo: *PongNode* and *PingNode*.

```
pong = new(Pong, num=1, at='PongNode@159.89.6.49'})
ping = new(Ping, num=1, at='PingNode@167.99.137.210')
```

Here is how to run the consensus algorithm on three different machines created using Digital Ocean *droplets* running an Ubuntu instance[1]. Once created, one must connect to the droplet using ssh and run the *setup.sh* file as explained in the *README* file.

```
python3.7 -m da -n p -H 165.22.18.54 src/run_remote.da flood_cons 3 10 20 5
python3.7 -m da -n q -H 165.22.18.25 -D src/run_remote.da
python3.7 -m da -n r -H 165.22.18.43 -D src/run_remote.da
```

[1] https://www.digitalocean.com/

# 4
# Evaluation

A benchmark was carried out to compare the different execution times of the total-order broadcast by varying the number of nodes in the system and the number of messages sent. Two versions of the implementation are compared: a non-optimized version preserving the architecture presented in the book [1] with the use of stubborn links (Section 3.3.2), and an optimized version using only the send primitive proposed by DistAlgo (Section 3.3.3).

As shown in Fig. 4.1, we can see that the optimized version is much faster than the one using stubborn links. This results in a much better performance in terms of time and memory usage as there is no overhead because of the continuous retransmission of the messages. The optimized execution with 20 processes in the system and 150 messages sent is twice faster than its non-optimized version.

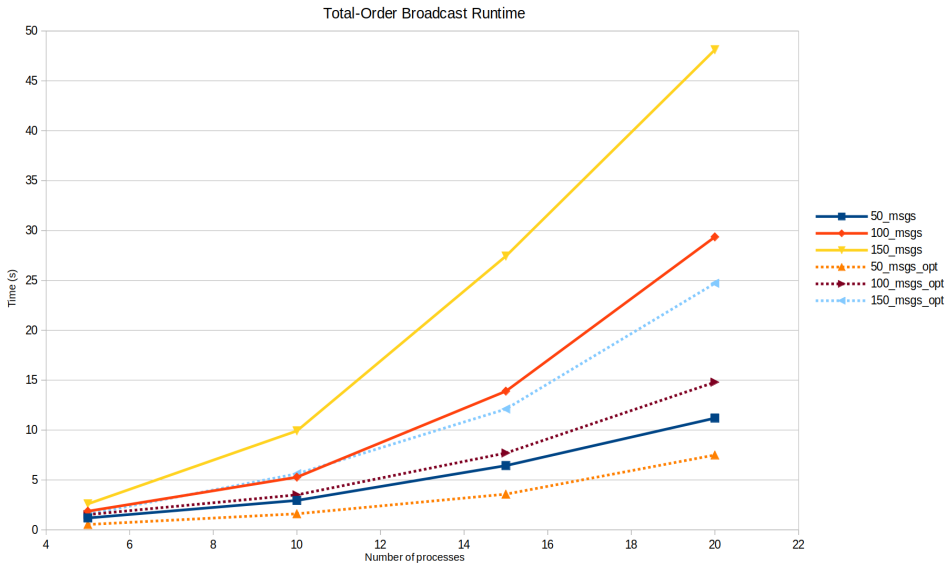


**Figure 4.1.** Total-Order Broadcast Benchmark.

# 5 Conclusion

## 5.1 Discussion

The main objective of this thesis was to see if the modular approach presented in [1] was both realisable and practical using DistAlgo. As this language has already proven itself in the world of distributed programming, it was now necessary to see if the development of algorithms using a stack of reusable components was feasible. We believe that the main objective has been achieved as we end up with a working instance of the total-order broadcast using multiple consensus instances. Nevertheless, several difficulties were encountered during the development phase.

Actually, at the beginning, the point-to-point links abstractions were implemented to send a single message at their creation, meaning that their lifetime was only during the sending of the message. However, when we started implementing the perfect failure detector, we realized that it was necessary to send a message periodically, which meant that we had the choice of either creating a new instance every time a message was needed or finding a way to have an instance that could send messages every time it was needed. The latter option has been chosen. In fact, the communication links are put on idle mode once created waiting for their *Send* event to be triggered. This strategy has also been adapted to the broadcast abstractions with a receive clause handling *Broadcast* requests.

Another problem caused by the modularity was that a message sent from the top of the stack had to go through every abstraction to capture each property and ensure the correctness of the algorithm. As a result, each receive clause on every module must match the required parameters to send a message. Initially, those were the content of the message, the sender and the receiver. Then, as the complexity increased, we needed to add certain parameters such as tags to differentiate incoming messages or some arguments. Therefore, each abstraction's receive clause had to be modified to match the number of parameters, otherwise the message would not be processed by the module. This led us to the creation of a *Message* class gathering all the necessary information. Thus, if new fields were to be added in the future, we would only need to modify this class.

In addition, the creation of the various modules also posed problems. At the beginning, we decided that each abstraction creates its own child modules. This method worked very well until we realized that some algorithms like the "Lazy Reliable Broadcast" used some abstractions multiple times, which brought a redundancy problem. We finally opted for the creation of an independent process called the *initiator*

which takes care of creating for each node an instance of each abstraction during the initialization phase.

Finally, the total-order abstraction has brought many new challenges as this module uses a consensus instance whenever necessary. Therefore, some way had to be found to instantiate one module knowing that this could lead to the eventual creation of other modules that did not necessarily exist yet. It was therefore necessary to notify existing modules of the creation of a new module and to inform them of the various changes. This whole operation results in many exchanges of messages between the initiator and the other DistAlgo processes.

Every implemented algorithm can also be run on remote nodes connected over the network. This has been accomplished with the help of DigitalOcean droplets running an Ubuntu instance. It is first required to setup the different machines and follow the installation steps provided in the README file. Then, the algorithm of your choice can be run on different nodes as explained in Section 3.8.

## 5.2 Future Work

First of all, it would be interesting to pursue the implementation of the library by adding new algorithms from the book in order to increase the overall complexity. Also, there is a space for the optimization of certain algorithms as it has been done for the perfect links.

It would also be interesting to compare the performance of a distributed system implemented using DistAlgo against a low-level language such as C++ or Java. These languages are obviously much faster than Python or DistAlgo however it adds a significant amount of complexity as the developer has more code to write than in a higher-level language like DistAlgo. It has been shown that the improper use of some C++ libraries could lead to a program an order of magnitude slower than its DistAlgo version [3]. This shows that DistAlgo's ease of use and understanding is a great advantage over its competitors.

# Bibliography


[1] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[2] DistAlgo. Distalgo language, 2020. Available at `https://github.com/DistAlgo`.

[3] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *ACM Trans. Program. Lang. Syst.*, 39(3), May 2017.

[4] Widmer Roland. Byzantine-fault tolerant algorithms in DistAlgo. Bachelor Thesis, Institute of Computer Science, University of Bern. 2020.

[5] Python Software Foundation. logging - logging facility for Python. Available at `https://docs.python.org/3/library/logging.html`.