An extended, modular library of distributed protocols

Bachelor Thesis

Marcel Haag

from

Bern, Switzerland

Faculty of Science, University of Bern

2. February 2024

Prof. Christian Cachin
Jovana Milojevic
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

Distributed programming aims to solve computing problems where different connected software and hardware components need to work together to solve some common tasks. This poses extra difficulties to the traditional centralized algorithms, like how to deal with disconnected or Byzantine processes in the system. Byzantine processes work arbitrarily, either because they are bugged or malicious, therefore interfering with the proper execution of the algorithm.
In this work, we extended an already existing modular library of distributed algorithms to deal with the consensus problem in different ways. Consensus refers to the process by which a group of processes agree on a common value, which was proposed by one of them. First, we start with *hierarchical consensus*, improve it with *hierarchical uniform consensus*, and end with a way to find consensus in a partially hostile environment with *randomized consensus*. This will all be done in the high-level programming language DistAlgo, which is specially optimized to deal with distributed algorithms.

Acknowledgments

I express my gratitude to my supervisor, Jovana Milojevic, for her guidance and assistance during this project. I am also grateful for the time she took for me, despite her busy schedule. I would also like to thank Professor Christian Cachin, who provided me with all the information necessary for the implementation of this work. Last but not least, I want to thank my family and friends for their continuous support.

Contents

Chapter 1

Introduction

Distributed algorithms allow multiple processes to work together in a network. This network can range from different command shells on a computer to globally distributed servers like the Bitcoin network. Networks need to tolerate different types of failures. Usual candidates are crashing processes, disconnection issues, bugged processes and malicious attacks. Coding distributed systems that are robust enough to tolerate failing processes and keeping the synchronisation between the working processes up is the main challenge of distributed algorithms.

To create those fault-tolerant systems, we use distributed programming abstractions. Abstractions allow us to strip down the complexity of code into the basic properties a model needs. Abstract models are independent services, that can interact with each other and need to fulfill certain properties. We start from basic abstraction models such as *processes* and *communication links* and gradually increase the complexity, by building them on top of each other. This is not always the most computationally efficient approach, but the error risk is much greater in monolithic algorithms [1].

In this thesis, we extend the library of distributed algorithms, developed by Lazic [4], which is implemented using distributed algorithm models with the *DistAlgo* library [2]. DistAlgo is a high-level programming language very close to pseudocode. This characteristic allows the programmer to focus on the core ideas of algorithms instead of low-level implementation trivialities. Further advantages of DistAlgo are that it is easy to send messages between processes and receive them. DistAlgo compiles its code into Python [1]. Thus, most of the code is written in Python, which helps students to learn and understand the language with ease and try it out themselves. Therefore, the main aim of this thesis is to provide a complement to the theoretical parts of the distributed algorithm class at the University of Bern with some concrete code to look at.

To fulfill this aim, the already existing library [4] is extended by the implementation of new consensus protocols. Consensus refers to the process by which a group of processes agree on a common value, which was proposed by one of them. This should be done, even when some processes crash and the communication is asynchronous. The implemented algorithms are: *hierarchical consensus*, *hierarchical uniform consensus* and *randomized binary consensus*.

In Chapter 2, we first introduce the most basic abstractions and assumptions, which are used throughout the work and introduce the basics of DistAlgo. Some models, that are necessary to understand the code are introduced at the end of Chapter 2. In Chapter 3, the implemented algorithms are described. Chapter 4 discusses the working process and gives ideas for further work on the library.

[1]https://www.python.org/

Chapter 2

Background

This chapter introduces essential distributed programming abstractions. For example the composition model, processes, communication links and different classes of algorithm abstractions. Later, the DistAlgo language is introduced. In the end, we introduce abstractions that are already implemented in the library [4].

2.1 Distributed Systems

Distributed systems are systems, whose components are located on different connected computers that seek to achieve some kind of cooperation. They consist of different processes running simultaneously, where all of them communicate through message exchange. Consequently, we need to consider that subsets of processes might crash or disconnect, which we call *partial failures*. This characteristic is what is used to differentiate between concurrent and distributed systems. Let us quote a famous definition for distributed systems from Leslie Lamport to highlight the challenges in distributed computing: *"A distributed system is one in which the failure of a computer you did not even know existed can render your own computer unusable."* [3].
When partial-failures occur, the hurdle is to ensure, that the functioning processes remain consistent. Sometimes distributed systems need to be able to tolerate processes that are controlled by malicious adversaries. However, we will not interact with such systems in this thesis.
All of the algorithms in this work are DistAlgo implementations of pseudocode from the book: *"Introduction to Reliable and Secure Distributed Programming"*[1]. Cachin *et al.* explain how to construct robust systems, given different problem assumptions, shown later in this chapter.

2.2 Distributed Programming Abstractions

Addressing the recurring challenge of solving identical issues across various devices or operating systems becomes redundant, underscoring the significance of abstraction in streamlining this process. Therefore it is paramount to have clear definitions of them, to understand the core components of a distributed system. They will reoccur throughout this thesis in different versions and forms, which makes them helpful to understand.

System Model. The underlying physical system is represented by two abstractions: *processes* and *links*. Units that are capable of computation inside of the distributed system are called processes. Links abstract the physical network used for processes to communicate. Those abstractions are not precise nor should they, as every process and link can be different depending on the distributed system. Moreover, the process abstraction will be different, depending on the type of faults they can have. Even the link abstraction changes on the delivery property they need to fulfill. See for example perfect links in Section 2.4.

Composition Model. Distributed algorithms are described through the composition model. The model comes straight from [1], which uses pseudocode for the description of the algorithms. The different algorithms are independent services, which can be accessed and be interacted with through *events*. Possible events are defined in the Application Programming Interface (API) of every algorithm.
Inside the composition model, a process consists of several layers to build a software stack. The layers can be imagined as similar to the famous OSI model. Usually, the top of the stack consists of the application layer and the bottom is the networking layer. The number of layers can differ, depending on the requirement of the algorithm. On every layer, there is a module, which can only communicate directly with the module in the upper and lower layers of the same process. Figure 2.1 shows that the modules only deal with two possible events: *requests* and *indications*.
Request events are used by a module to invoke a service at another module or to signal some kind of condition to another module. Those events usually come from an upper-layer component and invoke services in the ones below. An example of this would be when the application layer triggers a request event to a broadcasting component because it wants to deliver a message to a group of processes.
Indication events are the reverse of the request events. They deliver information usually to an upper layer. Let us use the broadcast example again. The message from before is delivered by the broadcast module to the application layer. This happens in all processes the broadcast was delivered to.

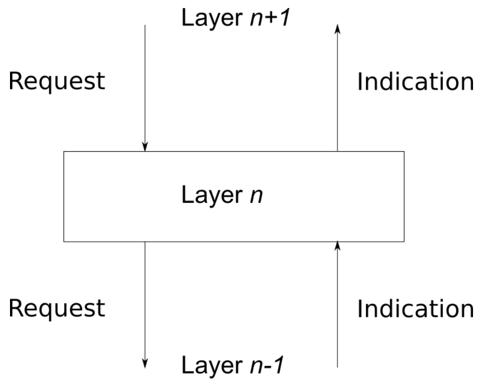

Figure 2.1. The composition model represents how and in which directions the different layers can communicate with each other.

Abstracting processes. A *process* is defined as a computational unit inside a distributed system. The system consists of N different processes usually named $P = \{p, q, ..., n\}$ etc. All of them should execute an instance of the algorithm. ‘Should’ is used here to indicate, that some processes might deviate from the correct execution of the algorithm. We call such processes *failures*. Failures exist for different reasons, like simple crashes or malicious behaviour.
We differentiate between *correct* and *faulty* processes. Correct processes are those, that never stop and follow the algorithm’s specifications. Faulty processes are all those that are not correct.
A different type of failures are *Byzantine* processes or *arbitrary faults*. They describe processes that deviate from the algorithm arbitrarily. It does not matter whether processes are bugged or intentionally controlled by a malicious adversary. As these types of failures will not be directly addressed in any of the abstractions of this thesis, it suffices to know that they exist.

Abstracting Communication. Networking components of the distributed system are described by the *link* abstraction. We assume a network, where every process is fully connected and capable of communication with every other process. All messages exchanged over a link are unique. They further contain enough information for a receiver to recognize the sender of the message. As the link abstraction tries to abstract real links, messages may get lost during the transition.

Classes of Algorithms. Classes of algorithms exist for the sake of grouping distributed programming abstractions. Different system assumptions, differentiate them. The different classes from the book [1] are:

1. *fail-stop* algorithms, 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* algorithms, where process crashes can never be reliably detected;
3. *fail-noisy* algorithms, 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* algorithms, where processes can crash and later recover and still participate in the algorithm;
5. *fail-arbitrary* algorithms, where processes can deviate arbitrarily from the protocol specification and act in malicious, adversarial ways;
6. *randomized* algorithms, where in addition to the classes presented so far, processes may make probabilistic choices by using a source of randomness.

In this thesis, we will only look at *fail-stop* and *fail-silent* classes of algorithms.

Quorums. In a common system with N processes, a *quorum* is defined as any set of more than $N/2$ processes. This guarantees, that every two quorums overlap in at least one process. Quorums play a pivotal role in the formulation of fault-tolerant algorithms. If Byzantine processes inside the system are assumed, we need extra precautions, for the scenario, where the overlapping process is Byzantine. For this, we use a *Byzantine quorum*, which tolerates f faults. Let us prove that two disjoint Byzantine quorums overlap in at least one correct process through contradiction.
A Byzantine quorum needs more than $\frac{N+f}{2}$ processes by definition. Consequently all such quorums contain more than $\frac{N-f}{2}$ correct processes, because

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

But this would mean, that two disjoint Byzantine quorums would have more than $N-f$ correct processes, as shown here

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

As there are only $N - f$ correct processes and the two quorums are assumed to be disjoint, we have our contradiction. Therefore the two Byzantine quorums overlap in at least one correct process.
Through some basic math, we can further find out how many faulty processes f are allowed in a fail-arbitrary model. A Byzantine quorum must consist of correct processes, which are received from at least one other correct process so that a system can make progress. This condition gives,

$$N - f > \frac{N+f}{2},$$

which is equivalent to $N > 3f$ the number of faulty processes allowed in a robust fail-arbitrary model.

2.3 DistAlgo

DistAlgo [2] is a high-level programming language very close to pseudocode. This allows the user to concentrate on the logic of the algorithm instead of implementation details. Additionally, the language provides abstractions to send messages between processes and statements to provide some synchronisation. We use a DistAlgo compiler with Python as the target language. The following section introduces the basics of DistAlgo according to the language description [2] and how it is implemented in Python.

2.3.1 Processes and Messages

Process definition. A DistAlgo process is defined in the following form. A type of process with the name *p* is defined as a class *p* which extends the class *process*. The method and handler definitions are in the *process_body*.

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

⟶ **In Python syntax:**

```
class p (process):
    process_body
```

Inside the process_body there are two special methods. The *setup* method sets up data in the process before the process starts. Secondly, the *run* method, is used to carry out the main flow of a process. In DistAlgo the special *self* variable refers to the process itself, which runs the execution.

Process creation. Process creation entails such statements as creating, setting up and starting processes. The following statement would create *n* new processes of type *p* at the node or group of nodes *node_exp*. The number of processes and the host location are optional.

```
n new p at node_exp
```

⟶ **In Python syntax:**

```
new(p, num = n, at = node_exp)
```

Sending messages. Sending messages can be done with the following statement. *mexp* is the type of the message and *pexp* is the set of processes, which will receive the message.

```
send mexp to pexp
```

⟶ **In Python syntax:**

```
send(mexp, to = pexp)
```

Deliver messages. Messages can only be delivered to processes if they have defined a *receive* clause. Such a clause is defined in the following statement, where *mexp* is the form of the message and *pexp* is the set of processes from which the message should come. The *from* statement is optional. As in the process definition, the statements inside the *body_handler* will only be executed, when the message fulfills the conditions of the receive statement.

```
receive mexp from pexp:
    process_body
```

⟶ **In Python syntax:**

```
def receive(msg = mexp, from_ = pexp):
    process_body
```

Pattern Matching. It is possible to receive and send matching messages through pattern matching. The first one defines components inside a message as tuples. Possible matching components are constants such as *'ack'*, variables like *n* and the _ statement, which matches anything. The following example also has a *from* statement, which binds *a* to the sender.

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

⟶ **In Python syntax:**

```
def receive(msg = ('ack', n, _), from_ = a)
```

Synchronisation. For synchronisation we use the *await* statement. It is blocking and waits until the boolean *bexp* inside returns true.

```
await bexp
```

⟶ **In Python syntax:**

```
await(bexp)
```

2.3.2 High-level Queries

There are three High-level queries, which are used to create complex synchronisation conditions.

Comprehension. The comprehension query returns a set of processes, which fulfill one or several boolean expressions. The following example returns the set of processes, that received a *Hello* message.

```
{p: receive ('Hello') from p}
```

⟶ **In Python syntax:**

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

Aggregation. There are several possible aggregation queries such as *count, sum, min* and *max*. The following combination extends the comprehension query with an aggregation query, to return the process with the smallest identifier, that sent *Hello*.

```
min {p: receive ('Hello') from p}
```

⟶ **In Python syntax:**

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

Quantification. Quantification queries have two forms. Firstly, the existential form returns true, if *some* members of a set satisfy a boolean expression. The universal quantification query returns true if *each* element of a set satisfies the boolean expression. The following expressions are examples of this.

```
some {p in procs: receive ('Hello') from p}
each {p in procs: receive ('Hello') from p}
```

⟶ **In Python syntax:**

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

2.3.3 Configuration

One can configure several parts of the system, such as the way messages are delivered, like first-in first-out ordering (FIFO). Another configurable part is the logical clock. Currently, only Lamport's clock is available. Only shown in Python syntax here.

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

2.3.4 Logging

Last but not least, there is a logging method in DistAlgo. Per default, the method prints logs on runtime into the console. But one can redirect the logs into a file as well. One can even select printing levels, such that the printing will only be displayed if the level is equal or higher.

```
output(exp1, ..., expk, level=1)
```

2.4 Building blocks

In this section, we will introduce certain modules, which have been previously implemented in an earlier library [4]. The code for these modules is available in the mentioned library. Subsequent algorithms introduced later in this paper rely on these modules. As a result, we introduce them selectively.

2.4.1 Perfect Links

Perfect links abstract a common sense understanding of what it means to send a message from one process to another. The precise specification of the abstraction is given in Module 1. All of the modules come from the book [1]. This common sense understanding is abstracted with two events and three properties. The *request* event comes from a modular layer above and requests to *send* a message to a certain process. The inverse, *indication*, *delivers* a message to an upper layer. The first property, *reliable delivery*, ensures the delivery of messages to correct processes if the message was sent by a correct process. *No duplication* means, that a message is exactly delivered once to the upper layer in the composition model by the receiver. The last property is *no creation*. Its meaning is simple, messages should not appear, if they were not sent by a correct process.
The perfect link abstraction is actually not the lowest level of abstraction in the book [1]. *Stubborn links* are a layer below perfect links. As the name implies, they stubbornly repeat the same message over and over, so that messages do not get lost. The code for the perfect link abstraction is essentially a filter, such that messages are only delivered once, even though they are sent infinitely. However, it is possible to code perfect links without the use of stubborn links in the DistAlgo language [4], which is why we can safely ignore stubborn links for the rest of the thesis.

Module 1 : Interface and properties of perfect point-to-point links

---

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

Events:
    Request: ⟨ *pl*, *Send* | q, m ⟩: Requests to send message m to process q
    Indication: ⟨ *pl*, *Deliver* | p, m ⟩: 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.

2.4.2 Perfect Failure Detector

A *failure detector* abstraction supplies all processes with the information about which processes crashed. The *perfect failure detector* detects crashed processes, if we take the fail-stop failure assumption in a synchronous system. The abstraction is presented in Module 2. The meaning of *strong completeness* is that at some point every crashed process will be detected. The perfect failure detector should never label a correct process as crashed (the *strong accuracy* property).
The perfect failure detector uses perfect links to ask, which processes are alive with so-called *heartbeat* messages. If fail-stop is assumed, the process will only respond if it is alive. Otherwise, it will not respond. If a process does not respond to the heartbeat message until a timer runs out, the process counts

as detected. Finally, the failure detector will notify all processes of the crash.

Module 2 : Interface and properties of the perfect failure detector

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

Events:
  Indication: $\langle$ $\mathcal{P}$, *Crash* | 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

---

2.4.3 Best Effort Broadcast

As the name suggests *best effort broadcast* is a type of broadcast abstraction. Broadcast abstractions try to distribute messages to a set of processes. The abstraction is represented in Module 3. It consists of three properties. The attentive reader recognizes the *no duplication* and the *no creation* property from perfect links. The *validity* property states, that a correct process, who broadcasts a message, will deliver the message. If the process crashes, there is no guarantee, that every process will receive the message. Best effort broadcast is the most basic kind of broadcast. It only consists of sending the message to all processes via perfect links and delivering the message, if it receives one. This fulfills all the necessary properties of best effort broadcast, as *no duplication* and *no creation* are given through perfect links and the *validity* property is fulfilled by the *reliable delivery* on every perfect link sent.

Module 3 : 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.
  BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

2.4.4 Reliable Broadcast

The *reliable broadcast* abstraction aims to broadcast messages, even when the original sender of the messages crashes. The properties can be seen in Module 4. The only difference to best effort broadcast is the *agreement* property. It ensures, that a message from a crashed process will be delivered to all processes. This can only happen if at least one process different from the sender has seen the message. There are different ways to implement the reliable broadcast abstraction. *Lazy reliable broadcast* uses a perfect failure detector and best effort broadcast. If a crashed process is detected, all messages from this process are re-transmitted via best effort broadcast from every correct process. *Eager reliable broadcast* does not need the perfect failure detector. Instead, every process re-transmits every message immediately.

Module 4 : 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.

Chapter 3

Extended Library of Algorithms

This chapter presents the implementation of new algorithms to the library [4] using DistAlgo. The focus lies on staying close to the pseudocode of the book [1] and not on performance and other optimizations. This way it can be assured, that the modularity and the pedagogical structure of the algorithms are as close to the book as possible.

3.1 Architecture Design

Like in the earlier master's thesis [4], this thesis implements abstractions as described in Chapter 2. Everything is constructed in a modular way, such that complex problems can be simplified into individual modules using other modules. The building blocks from the library, which are used, can be seen at the end of Chapter 2.
At the beginning, DistAlgo processes execute their *run* method. Afterwards, they wait for messages to be handled. All modules that implement an algorithm are at first represented as a singular DistAlgo process. With the architecture design from the master thesis [4], modules are stacked upon each other, according to their dependencies. For example, a perfect failure detector module uses perfect links, which uses stubborn links. So every node running a perfect failure detector runs those three layers. But they can only communicate with the layer above and below. Only the lowest layer can communicate with the other nodes. Therefore the architecture design is very close to the modular design from the abstractions in the book [1].
The inter-module communication is handled with receive and send clauses from DistAlgo described in Section 2.3. Usually, a send message looks like this:

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

This is a send message inside a stubborn link. The four elements of the tuple are as follows. First, *'Send'* describes the type of event. The *path* tells the message through which modules it went. By reversing the path, a message can find out to which module the message needs to go in a new node. The *tag* is used when a module has more than one type of message to disseminate. Finally, *m* stands for the message, which is sent. More details can be read in the master thesis [4].

3.2 Regular Consensus

The consensus problem consists of *proposing* a value from every process and *agreeing* on one of them. Regular consensus is an abstraction described in Module 5. The module has a *propose* event, which is a request from an upper layer. As an Indication, the consensus module returns a *decision*. In addition, four properties need to be fulfilled for a regular consensus abstraction. *Termination* and *integrity* ensure, that every process decides exactly once. *Validity* ascertains, that the consensus primitive does not invent a decision value. Finally, *agreement* states that no two correct processes should decide differently. Hierarchical consensus is a version of regular consensus.

Module 5 : Interface and properties of regular consensus

Module:

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

Events:

Request: $\langle$ *c*, *Propose* | v $\rangle$: Proposes value v for consensus.
Indication: $\langle$ *c*, *Decide* | 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.

3.2.1 Hierarchical Consensus

Hierarchical consensus uses the best effort broadcast (Module 3) and perfect failure detector (Module 2) abstraction for its implementation of regular consensus, described in Module 5. Clearly, we assume fail-stop, as we use the perfect failure detector. The algorithm starts by proposing a value for every process. In the most straightforward case, the highest-ranking process broadcasts its value and decides on it. Then a new round starts and the second highest ranking process decides on the value of the broadcast and broadcasts the same value. Rinse and repeat until all processes have been decided. If a process crashes, its round is ignored. If the highest ranking process crashes without broadcasting, the next highest ranking correct process decides on a starting value and broadcasts it.

Propose. At first, every correct process proposes a value *v* and saves it in a variable.

```
def receive(msg=('Propose', path, v), from_=p):
    self.proposal = v
```

Starting the decisions. Afterward, the processes wait for an event to happen. Either, that the process with the highest rank has the same round number as its rank and does not propose FALSE or that the current round is in a list called *detectedranks* or the process of the current round has decided on a value. In the first case, the process decides on its value and broadcasts it to every other process. The other two cases start a new round.

```
def run():
    await(((self.rnd == self.rank(self.name))
            and (self.proposal != None)
            and (self.broadcastbool == False))
            or (self.rnd in self.detectedranks)
            or (self.delivered[self.rnd] == True))
    if((self.rnd == self.rank(self.name))
            and (self.proposal != None)
            and (self.broadcastbool == False)):
        self.broadcastbool = True
        b = broadcast.Broadcast('', self.name)
        b.args = self.proposal
        send(('Broadcast', ['hier_cons'], 'DECIDED', b), to=self.beb)
        if path ==[]:
            output("DECIDE : ", self.proposal)
        else:
            send((path[1:], '', b), to=self.system[path[0]][self.name])
    if((self.rnd in self.detectedranks) or (self.delivered[self.rnd] == True)):
        self.rnd += 1
```

The detected ranks list is extended, every time a process crashes.

```
def receive(msg=('Crash', name_crashed), from_=p):
    detectedranks.append(self.rank(name_crashed))
```

Deliver DECIDED messages. In this section of the code, the DECIDED messages are received. The *if* condition ascertains, that the decision came from a higher ranking process. The second part of the if statement and the *proposer* variable concern themselves with an interesting case. Imagine a system, where the processes *p, q, r* ranked in this order exist. Now the process *p* broadcasts its value, but only *r* receives it because p crashed in the middle of the delivery. As *q* detects *p*, it starts the second round, decides on its own value and broadcasts the decision. Thanks to the *proposer* variable, *r* takes the value from *q* and not from *p*. This is what we need, to fulfill the *agreement* property from Module 5. In the end the *delivered* list is filled, such that a new round can start.

```
def receive(msg=(path, 'DECIDED', m), from_=p):
    r = self.rank(m.sender)
    if (r < self.rank(self.name)) and (r > self.proposer):
        self.proposal = m.args
        self.proposer = r
    self.delivered[r] = True
```

Ranking Subroutine. The rank function is not initially implemented into DistAlgo. Here is a trivial implementation of it.

```
def rank(process):
    for i in range(len(self.procs)):
        if (process == self.procs[i]):
            return i + 1
```

3.3 Uniform Consensus

The uniform consensus abstraction only deviates in one property from regular consensus. Instead of the *agreement* property, uniform consensus has *uniform agreement*. So no two processes should ever decide differently instead of no two correct processes. Module 6 describes it precisely.

Module 6 : Interface and properties of uniform consensus

Module:

Name: UniformConsensus, **instance** *uc*.

Events:

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

Properties:

UC1: *Termination:* Every correct process eventually decides some value.
UC2: *Validity:* If a process decides v, then v was proposed by some process.
UC3: *Integrity:* No process decides twice.
UC4: *Uniform agreement:* No two processes decide differently.

3.3.1 Hierarchical Uniform Consensus

Hierarchical uniform consensus uses the perfect failure detector (Module 2), perfect link (Module 1), best effort broadcast (Module 3) and reliable broadcast (Module 4) abstractions. The basic idea is, that the highest ranking process proposes its value on every other process with a best effort broadcast. Every process receiving the broadcast acknowledges this with a perfect link message. During the whole time, the highest ranking process keeps a list of every process that acknowledged its proposal and the detected ranks. As soon as the union of those two lists equals the set of all processes, the decision is broadcasted via reliable broadcast. This way, no process that later crashes can decide on a value, unless it is the one from the reliable broadcast. As in hierarchical consensus, there is a round system. But the round only ever changes if the highest ranking process crashes.

Propose. The proposal is basic, it saves the proposal in a variable on the process.

```
def receive(msg=('Propose', path,v), from_=p):
    if proposal == None:
        self.proposal = v
```

Initial broadcast. In hierarchical uniform consensus, three different methods wait for a special event to happen which is not a message. They all await a statement, which needs to become true on the run method of DistAlgo. Due to this, the *run()* method gets quite big, which is why we opted to only show the *run()* method in sections instead of the whole. The initial best effort broadcast is one of them. It waits for the case that the round equals the rank of the process and nothing has been decided. Further, it should not be the case, that no valid proposal (i.e. the *None* variable) is proposed.

```
if((self.rnd == self.rank(self.name)) and (self.proposal != None)
        and (self.decision == None)):
    b = broadcast.Broadcast('', self.name)
    b.args = self.proposal
    send(('Broadcast', ['hier_uni_cons'], 'PROPOSAL', b), to=self.beb)
```

Sending ACK messages. If the broadcast from above is received, the receiving process sends an ACK message over a perfect link, to acknowledge this fact. Moreover, the value of the proposal is saved inside a python dictionary, a precautionary measure for crashes.

```
def receive(msg=(path, 'PROPOSAL', m), from_=p):
    self.proposed[self.rank(m.sender)] = m.args
    if(self.rank(m.sender) >= self.rnd):
        path.append('hier_uni_cons')
        m.msg = "Ack" + str(m.msgid)
        m.receiver = m.sender
        m.sender = self.name
        send(('Send', path, 'ACK', m), to=self.pl)
```

Receiving ACK messages. The ACK message makes the highest ranking correct process to save the sender inside the *ackranks* set. The set of all acknowledged ranks.

```
def receive(msg=(path, 'ACK', m), from_=p):
    self.ackranks.append(self.rank(m.sender))
```

Crash. In the case of a crash, the crashed process rank is added to the set of the ranks of all crashed processes.

```
def receive(msg=('Crash', name_crashed), from_=p):
    detectedranks.append(self.rank(name_crashed))
```

Additionally, the round changes, if the current round is inside the *detectedranks* variable. This is done inside the run method because it waits for an event instead of a message from another abstraction. The following method further concerns itself with setting the proposal variable to the highest ranking proposal, which the process received.

```
if((self.rnd in self.detectedranks)):
    if(self.proposed[self.rnd] != None):
        self.proposal = self.proposed[self.rnd]
    self.rnd += 1
```

Broadcast DECIDED message. As soon as the union of all *ackranks* and *detectedranks* equals the set of all processes, the DECIDED message is broadcasted.

```
if(set(self.detectedranks).union(set(self.ackranks)) == set(self.proposed.keys())):
    m = broadcast.Broadcast('', self.name)
    m.args = self.proposal
    send(('Broadcast', ['hier_uni_cons'], 'DECIDED', m), to=self.rb)
```

Deliver DECIDED message. In the end, a value from the highest ranking process is decided on every process.

```
def receive(msg=(path, 'DECIDED', m), from_=p):
    self.decision = m.args
    if path == []:
        output("DECIDE : ", self.decision)
    else:
        send((path[1:], '', m), to=self.system[path[0]][self.name])
```

3.4 Randomized Consensus

Randomized consensus needs the same properties to be fulfilled as regular consensus. Nonetheless, they are completely different, because randomized consensus assumes a fail-silent model, instead of a fail-stop model. As we cannot detect crashes, we use randomness to make sure that the processes eventually

Module 7 : Interface and properties of regular randomized consensus

Module:

Name: RandomizedConsensus, **instance** *rc*.

Events:

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

Properties:

RC1: *Probabilistic termination:* With probability 1, every correct process eventually decides some value.
RC2: *Termination:* Every correct process eventually decides some value.
RC3: *Validity:* If a process decides v, then v was proposed by some process.
RC4: *Integrity:* No process decides twice.
RC5: *Agreement:* No two correct processes decide differently.

terminate. The description of the abstraction is in Module 7.

To ensure randomness, randomized consensus algorithms delegate their probabilistic choices to a *common coin* abstraction. The precise specifications of the common coin can be seen in Module 8. A common coin is invoked by triggering a *release* event at every process. Every process randomly flips a coin. The value of the coin *c* is only sent if an *output* is indicated. The *termination* property ensures an output of the coin. The second property hides the coin value until a process releases the coin. The last two properties specify the probability distribution of the coin.

Module 8 : Interface and properties of a common coin

Module:

Name: CommonCoin, **instance** *coin*, with domain $\mathcal{B}$.

Events:

Request: $\langle$ *coin*, *Release* $\rangle$: Releases the coin.
Indication: $\langle$ *coin*, *Output* | b $\rangle$: Outputs the coin value $b \in \mathcal{B}$.

Properties:

COIN1: *Termination:* Every correct process eventually outputs a coin value.
COIN2: *Unpredictability:* Unless at least one correct process has released the coin,
no process has any information about the coin output by a correct process.
COIN3: *Matching:* With probability at least δ, every correct process outputs the same coin value.
COIN4: *No bias:* In the event that all correct processes output the same coin value,
the distribution of the coin is uniform over $\mathcal{B}$
(i.e., a matching coin outputs any value in $\mathcal{B}$ with probability $\frac{1}{\#\mathcal{B}}$).

3.4.1 Randomized Binary Consensus

In randomized *binary* consensus, the processes can only decide between the values 0 and 1. The abstractions used are: best effort broadcast, reliable broadcast and multiple instances of the common coin. As in the two hierarchical consensus variants, the algorithm works in rounds. Every round has two phases.

In the first round, all processes broadcast their proposal. If a process receives more than half of the proposals from other processes, it saves them as its proposal, if they are all the same. Otherwise it uses *"noMajFound"* as its value. At the end of phase 1, phase 2 is started and the new proposal is shared via best effort broadcast. Phase 2 waits until $N - f$ values from phase 1 have been proposed. It starts a coin round with the common coin abstraction and releases the coin. In the output of the coin event, the processes look, whether more than the faulty processes have proposed the same value. Now there are three things, which could happen. A majority is found and the process sends a DECIDED message with reliable broadcast. No majority is found, but there exists a value different from *"noMajFound"*. This value is taken as its own. If none of this happens, the value from the common coin is taken. In the last two cases, a new round is started.

Propose. Upon the proposal, the first round and first phase is initialized. The proposal is saved inside a *proposal* variable and then shared.

```
def receive(msg=('Propose', path, v), from_=p):
    self.proposal = v
    self.rnd = 1
    self.phase = 1
    b = broadcast.Broadcast('', self.name)
    b.args = (self.rnd, self.proposal)
    send(('Broadcast', ['random_bin_cons'], 'PHASE-1', b), to=self.beb)
```

Delivery in phase 1. Every process has a Python dictionary of all processes called *val*. As soon as the phase-1 message is delivered, the proposed value is saved inside the dictionary. In the code from the section above, one can see that *b.args* is a tuple. So in *b.args[0]* the round is saved and in *b.args[1]* the proposal. The if condition ensures, that all messages come from the correct round inside the correct phase.

```
def receive(msg=(path, 'PHASE-1', b), from_=p):
    if(self.phase == 1 and b.args[0] == self.rnd):
        self.val[b.original_sender] = b.args[1]
```

Starting phase 2. Once, more than half of the values inside the *val* dictionary are filled with an entry, the second phase starts. If more than half of the values in *val* are the same, the process takes this value as its proposal for phase 2. Otherwise, it proposes the value *"noMajFound"*. Before the start of phase 2, *val* is cleared.

```
if(numValues(self.val) > int(self.numberOfProcs / 2)
        and self.decision == None
        and self.phase == 1):
    majority = majorityCount(self.val)
    if majority[0] > int(self.numberOfProcs / 2):
        self.proposal = majority[1]
    else:
        self.proposal = "noMajFound"
    self.val.clear()
    for p in self.procs:
        self.val[p] = None
    self.phase = 2
    b = broadcast.Broadcast('', self.name)
    b.args = (self.rnd, self.proposal)
    send(('Broadcast', ['random_bin_cons'], 'PHASE-2', b), to=self.beb)
```

The code here differs a little from the randomized binary consensus described in [1, Algorithm 5.12]. First of all, the methods *numValues()* and *majorityCount* are new subroutines. They do mostly the same things as the pseudocode. The *numValues()* function returns the number of values inside *val* different

from *None*. The *majorityCount* subroutine returns a tuple, with the first element counting the number of times the majority value is inside *val* and the second element the majority value itself.
Another original creation is the *“noMajFound”* tag. In Algorithm 5.12 from [1], the *numValues()* function implicitly counts how many different proposals are inside *val*. This cannot be done directly in DistAlgo. The DistAlgo version counts how many values in *val* are different from $\perp$ (implemented with *None*). In the pseudocode, the *val* variable is filled with N times the value $\perp$ at the start. Now in the pseudocode version from the part above 3.4.1, if no majority has been found, the saved value is $\perp$ as well. So the case, where no value has been proposed and the case, where no process found a majority from the part above 3.4.1 would be treated the same in the implementation. To circumvent this problem, we introduce the tag *“noMajFound”*.
Another solution for the *numValues()* method would have been to use a list of tuples, instead of a dictionary for the representation of the *val* variable. We did not choose this option, because it would differ in much more ways from the pseudocode.

```
def numValues(val):
    x = list(val.values())
    return len(x) - x.count(None)
```

```
def majorityCount(val):
    majCount = 0
    majValue = None
    for i in set(val.values()):
        x = list(val.values())
        counter = x.count(i)
        if counter > majCount and i != None and i != "noMajFound":
            majCount = counter
            majValue = i
    return (majCount, majValue)
```

Delivery in phase 2. A delivery in phase 2 does the same thing as a delivery in phase 1, just with the phase-2 assertion.

```
def receive(msg=(path, 'PHASE-2', b), from_=p):
    if(self.phase == 2 and b.args[0] == self.rnd):
        self.val[b.original_sender] = b.args[1]
```

The coinround. As soon as all nonfaulty processes have proposed something in phase 2 of a round, a coin round is started. The coin round starts a new instance of common coin and requests a release of the coin. Additionally, the phase is set to 0, so no phase-1 or phase-2 broadcast will be received.
Here is another deviation from the original book [1]. Due to time and knowledge constraints, it was not possible to implement the common coin abstraction. A possible implementation is discussed at the end of the subsection. Instead, a shortcut was taken with best effort broadcast and the random library from Python.

```
if(numValues(self.val) >= self.numberOfProcs - self.f
        and self.decision == None and self.phase == 2):
    self.phase = 0
    b = broadcast.Broadcast('', self.name)
    send(('Broadcast', ['random_bin_cons'], 'COINROUND', b), to=self.beb)
```

Output of the coinround. Every process receives a binary value from the coinround (through a best effort broadcast). The processes first check whether a majority of more than the number of faulty processes proposed the same value in phase 2. If this is the case, the value is decided and will be broadcasted with a reliable broadcast. Provided that less or equal to the number of faulty processes have proposed the same value, this value is saved as a proposal for the next round. If only *“noMajFound”* has been proposed for phase 2, the proposal for the next round will be the random value from the common coin. If nothing has been decided, a new round starts in phase 1.

```
def receive(msg=(path, 'COINROUND', c), from_=p):
    majority = majorityCount(self.val)
    if majority[0] > self.f:
        self.decision = majority[1]
        b = broadcast.Broadcast('', self.name)
        b.args = self.decision
        send(('Broadcast', ['random_bin_cons'], 'DECIDED', b), to=self.rb)
    else:
        if majority[1] != None :
            self.proposal = majority[1]
        else:
            self.proposal = random.randint(0, 1)
        self.val.clear()
        for p in self.procs:
            self.val[p] = None
        self.rnd += 1
        self.phase = 1
        b = broadcast.Broadcast('', self.name)
        b.args = (self.rnd, self.proposal)
        send(('Broadcast', ['random_bin_cons'], 'PHASE-1', b), to=self.beb)
```

Delivering DECIDED message. When a DECIDED message is received, the value from the message is decided.

```
def receive(msg=(path, 'DECIDED', m), from_=p):
    self.decision = m.args
    if path == []:
        output("DECIDE : ", self.decision)
    else:
        send((path[1:], '', m), to=self.system[path[0]][self.name])
```

A solution for Coinround. The best way to implement the common coin abstraction is a common coin module running on a separate process from the rest of the consensus processes. It waits until $N - f$ processes have released their coin. Then, a random binary output to the consensus processes is sent.
The library [4] provides the ability to run the same modules on different processes at the same time. A big change to the run and initiator classes of the library would be necessary, to run the common coin process on another process than the consensus processes, while still being able to communicate with each other. Due to a lack of understanding of the run and initiator classes, the change was not possible within the deadline.

3.5 Run Algorithm

To run the algorithms one needs to follow the installation steps of the README file. Python 3.7 is used because it is the latest version supported by DistAlgo [2]. To set the parameters of the algorithms one needs to type into the command line:

```
<algorithm> <n> <proposal_1> <proposal_2> ... <proposal_n>
```

Run locally on multiple consoles. Assume hierarchical consensus is run on four consoles. The first process runs the main method and starts the initialization process. With the `-n` flag, each console is given a unique name for the inter-process communication. `-m` tags a new module. The `-D` argument is added, so all but the first processes are run as *idle*. Down below is an example of a locally run version of hierarchical consensus.

```
    python3 -m da -n p src/run.da hier_cons 4 60 5 13 210
    python3 -m da -n q -D src/run.da
    python3 -m da -n r -D src/run.da
    python3 -m da -n s -D src/run.da
```

Chapter 4

Conclusion

4.1 Discussion

The goal of this thesis was the extension of the library [4] with implementations from [1]. The desired implementations were hierarchical consensus, hierarchical uniform consensus and randomized binary consensus. The initial goal has been mostly achieved, as the library was indeed extended by those three new functioning modules. Nonetheless, we found limits to the modular approach of the current implementation of the library with the common coin abstraction for randomized consensus. Otherwise, the library was a well-thought-out, but challenging work environment.
The first discussable decision was the implementation of the rank function. It would have been possible to implement the rank of a process upon initialisation. But due to the fact, that the ranking function is rarely used, we opted for a simple *for* loop.
A problem we encountered was by sending perfect links in hierarchical uniform consensus. After receiving the best effort proposal from the leader, the receiving process should respond with an ACK tagged message. The problem with this message was, that the message had no content different from the tag. Another module, which only used tagged messages was the perfect failure detector. Because of this similarity, the perfect links filtered out all ACK messages, as HEARTBEAT messages from the perfect failure detector are sent more often and earlier. To solve the problem, we added content to the ACK messages.
Another problem was caused by the implementation of the *numValues()* function in randomized consensus. The problem was solved through the introduction of a new variable as mentioned in the respective paragraph.
The final problem of this thesis was the implementation of the common coin. As already mentioned, the best possible implementation of the common coin algorithm went completely against the modular approach for which the library was created. It needs a separate node, which can communicate directly with the randomized binary consensus module, without using other abstractions in the best case. As those changes were too big to complete in the time scope of a bachelor thesis, we opted for an ugly quick fix.

4.2 Future Work

For future work, an implementation of randomized binary consensus, which is closer to the book would be interesting. It would also be interesting to implement new algorithms from the book. Time and memory optimizations for the algorithms come to mind for future work. As a challenge, one could implement the algorithms in a lower-level language like C, to compare the execution speed.

Bibliography

[1] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

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

[3] L. Lamport, "Mail from leslie lamport." Available at `https://lamport.azurewebsites.net/pubs/distributed-system.txt`.

[4] A. Lazic, "The library of distributed protocols," master's thesis, University of Bern, 2021. Available at `https://crypto.unibe.ch/archive/theses/2020.msc.aleksandar.lazic.pdf`.

Erklärung

Erklärung gemäss Art. 30 RSL Phil.-nat. 18

Ich erkläre hiermit, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.

Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum	Unterschrift