b
UNIVERSITÄT
BERN

Byzantine-fault tolerant algorithms in DistAlgo

Bachelor Thesis

Roland Widmer
from
Bern, Switzerland

Faculty of Science, University of Bern

July 21th, 2020

Prof. Christian Cachin
Orestis Alpos, Luca Zanolini
Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland

Abstract

DistAlgo is a high-level programming language for implementing distributed algorithms. Due to the high degree of abstraction, DistAlgo code is close to pseudocode. This makes it ideal for educational purposes, while it is still a running and efficient implementation. In this thesis, a DistAlgo compiler is used that produces regular Python programs. After an introduction to the programming language and the compiler, some Byzantine-fault tolerant algorithms are introduced and implemented in DistAlgo. The differences between pseudocode and DistAlgo implementation are analyzed and discussed.

Acknowledgements

First and foremost, I would like to thank my supervisors, Orestis Alpos and Luca Zanolini, for the great support. With their creative ideas they contributed a lot, especially when it came to questions about algorithms and DistAlgo. I would also like to thank Prof. Christian Cachin with his profound expertise in this topic. Furthermore, I wish to thank Alexandra Elia and Martin Widmer for proofreading the thesis.

Contents

1 Introduction

Distributed algorithms are algorithms intended to achieve a certain degree of cooperation on a set of processes [1]. They may be written in high-level pseudocode using certain assumptions, but practical implementations are usually much more extensive. DistAlgo is a high-level programming language to implement distributed algorithms in an abstract and well-structured way. Despite its high degree of abstraction, a working Python program is compiled and executed. The executability and clarity of DistAlgo implementations have many advantages, for example, researchers could find improvements in distributed algorithms using a DistAlgo implementation [2].

DistAlgo needs to make many implicit assumptions due to its high degree of abstraction. These assumptions may be constraints for programming algorithms as well. Considering some Byzantine-fault tolerant algorithms, we will determine how they can be implemented in DistAlgo. An additional goal is to obtain DistAlgo code that is as similar as possible to the pseudocode.

This thesis starts with an explanation of the abstractions used in the pseudocode implementation of the algorithms in Chapter 2. An introduction to DistAlgo follows, which also includes the abstractions used in DistAlgo. Chapter 3 introduces and describes the algorithms using pseudocode. In Chapter 4, we look at the implementations in DistAlgo. The implementations have some caveats, which are discussed in Chapter 5. All implementations may be downloaded and executed, the instructions can be found in the appendix.

2 Background

2.1 Abstractions in Distributed Programming

Abstractions are used in Distributed Programming to focus on the essential and to hide unnecessary details. This thesis uses abstractions from the book "Introduction to Reliable and Secure Distributed Programming" [1]. The algorithms introduced in the next chapter are also from this book and build on those abstractions.

System model. The physical system may be represented with two main concepts, *processes* and *links*. Entities, that are able to perform computations in a distributed system are called processes. A process may for example represent a computer or a thread on a processor. In distributed computing, processes should be able to communicate with each other. The logical and physical network connecting processes is abstracted as a link. Links allow processes to exchange messages.

Composition model. To describe distributed algorithms in pseudocode, a composition model is helpful. A process hosts a set of software components, called *modules*. A module is identified by a name and characterized by a set of properties. It provides an interface with *events* that it accepts and produces. Using modules, a layered architecture using a software stack may be constructed. At each process, a module represents a layer in the stack. Modules within the same stack communicate through the exchange of events. Since there may exist multiple *instances* of a module running on the same process at the same time, instances are identified with a corresponding identifier.

In this thesis an *algorithm* in pseudocode implements a specific module. Algorithms consist of a set of *event handlers*. Some are implementations of the received and produced events described in the module definition. There may be additional event handlers, which are triggered if some condition becomes TRUE. Those events are called *internal events* and the condition uses usually local variables. Another type of event is the *initialization event*. This event is generated automatically by the runtime system when an instance is created.

Process abstractions. A process executes a distributed algorithm using the modules which implement the algorithm within that process. If a process does not behave according to the algorithm, a *failure* occurs. Different types of faults lead to the failure of a process. Process abstractions are used to define if a process is faulty or correct. The simplest failure of a process is a crash, where a process stops to execute steps. With this *crash-stop* process abstraction, a process is considered *faulty* if it crashes at some time during the execution, otherwise it is called *correct*.

This process abstraction is simple, but it covers only a subset of process failures. Other types of failures may occur, for example, if a process does not send a message it is supposed to send, which is called *omission* fault. The most general process abstraction is the *arbitrary-fault* process abstraction. There are no assumptions on the behavior of faulty processes, which means they can deviate in an arbitrary manner from the algorithm. This includes intentional and malicious attacks, but also failures caused by a bug in the implementation. Those failures are also called *Byzantine* and are the most expensive to tolerate. The distinction between faulty and correct processes is static. In other words, a process which is correct in the beginning and then crashes later is considered faulty.

Link abstractions. Abstractions are also used for links in order to assume certain properties. In this thesis, all algorithms will use the *Authenticated perfect link* abstraction, which ensures the following three properties:

1. **Reliable delivery:** If a correct process sends a message m to a correct process q , then q eventually delivers m.
2. **No duplication:** No message is delivered by a correct process more than once.
3. **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 may also be represented as a module using our composition model, where two events are provided as an interface, one to send and one to receive messages.

Broadcast. To disseminate information among a set of processes, *broadcast communication* abstractions are used. They differ according to the reliability of the dissemination. With the arbitrary-fault process abstraction, a process can deviate arbitrarily from its assigned algorithm. A process may also try to prevent the goals of the algorithm from being reached. An algorithm has to tolerate such behavior.

Quorums. A helpful tool to design fault-tolerant algorithms is *quorums*. A quorum is a set of processes with special properties. Using the arbitrary-fault process abstraction, *Byzantine quorums* may be used. A Byzantine quorum tolerating f faulty processes in a system with N processes is a set of more than $\frac{N+f}{2}$ processes. A useful property of Byzantine quorums is that two quorums overlap in at least one correct process. In every Byzantine quorum, there may be at most f faulty processes. It follows that a quorum has more correct processes than $\frac{N-f}{2}$, because

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

Considering two disjoint quorums, then there are more than $N - f$ correct processes, since

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

This is a contradiction, hence there is at least one correct process occurring in both quorums.

The algorithms introduced in the following chapter require a Byzantine quorum of correct processes. This is achieved by restricting the number of faulty processes f in the system, namely f is required to be smaller than $\frac{N}{3}$. This inequality is derived from the following formula, which states that the cardinality of the set of correct processes $N - f$ is larger than the cardinality of a Byzantine quorum:

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

2.2 Introduction to DistAlgo

The advantage of a pseudocode implementation is the high degree of abstraction, which is used to facilitate the understanding of algorithms. However, such implementations may not be executed on a system. *DistAlgo* is a very high-level language for clear descriptions of distributed algorithms. It combines the advantages of pseudocode and programming languages. The introduction follows roughly the "DistAlgo language description" [3].

Compiler. In this thesis, a *DistAlgo compiler* with Python as a target language is used. The DistAlgo code is processed by the compiler and the output is an ordinary Python program, which can be run on a system with Python installed. In the following chapters, the DistAlgo syntax for the Python compiler is used, as currently only this Python compiler exists.

DistAlgo abstractions. The system model of a DistAlgo program is similar to the model described beforehand. Since a DistAlgo program is executable on a system, it has to be more specific. Processes in DistAlgo are represented as an instance of a *process* class. They use the Internet protocol as *link*, the usage of TCP or UDP can be configured.

Messages. Processes are identified by a process identifier *pid*. To send and receive messages only the identifier is used, the implementation with the Internet protocol is hidden in the DistAlgo code. Messages are implemented as tuples, in this example, a 2-tuple is sent to the process with identifier *pid*.

```
send(('Hi.', 'Greetings.'), to = pid)
```

Receive handlers may be added, similar to the event handlers of the abstract model. In this example, the function is called if a 2-tuple as in the example above is received. The variable *p* specified in the *from_* argument may store a process identifier. The receive handler is only triggered if the sender of the received message matches. If the variable *p* is not defined, the sender of the received message is accessible in the following code block using this variable. The argument *from_* is used since *from* is a reserved word in Python.

```
def receive(('Hi.', 'Greetings.'), from_ = pid):
  # function body
```

All messages sent and received by a process are kept in variables *sent* and *received*.

Synchronisation. The *await* statement waits for the value of a Boolean-valued expression to become TRUE. This example waits until such a message is received, then the normal flow of control continues.

```
await(received(('Hi.', 'Greetings')))
```

To wait for multiple Boolean-valued expressions to become TRUE, a multi-part await statement may be used. The order of the statements is irrelevant, the code block of the statement which becomes TRUE first is executed.

```
if await(received(('Hi.',)):
  ...
elif received(('Bye.',)):
  ...
elif received(('How are you?',)):
  ...
```

The expression may also be more complex, for example using queries described in the next paragraph. If the process is waiting in an await statement, all messages are handled by default. In all other places, a yield point has to be added, otherwise incoming messages are not handled. This example includes a busy waiting loop. Without the yield point *handleMessages*, incoming messages are not handled. The *handleMessages* statement is a special DistAlgo structure that is transformed into a function call by the compiler. This function handles the received messages.

```
while True:
  -- receiveMessages
  # loop body
```

Queries. A high-level query is over arbitrary sets or sequences, however often the message history *received* is used. Three concepts may be used, namely *comprehension*, *aggregation*, and *quantification*. Combinations and nesting of these are possible as well. In this introduction the formal definitions are omitted, instead, some explicit examples are introduced, which are similar to the queries used in the implementations of the algorithms.

A *comprehension* is implemented using the *setof* statement. The following example returns the set of all processes from which the process received a *Hi* message.

```
setof(proc, received(('Hi',), from_= proc))
```

Aggregations use an aggregation operator, which may be *len*, *sum*, *min*, or *max*. The first line returns the cardinality of the set of all processes from which the process received a *Hi* message. This set consists of DistAlgo process identifiers. There is an order implemented on this set, therefore *min* and *max* operators may also be used. The second line returns the highest process identifier from the same set as before.

```
len(setof(proc, received(('Hi',), from_= proc)))
max(setof(proc, received(('Hi',), from_= proc)))
```

A *quantification* is implemented using the *each* or the *some* statement and a Boolean value is returned. Assuming the variable *clients* is the set containing all processes which have sent a *Hi* message to the process, then TRUE is returned. Otherwise, if a process exists in this set which has not sent a *Hi* message to the process, then FALSE is returned. The second line returns TRUE if a process exists which has sent a *Hi* message to the process. Sometimes, it is helpful to know for which process the condition in the *has* argument is true. This process is accessible later by using the variable *proc*.

```
each(proc in clients, has=received(('Hi',), from_ = proc))
some(proc in clients, has=received(('Hi',), from_ = proc))
```

Structure of a DistAlgo program. DistAlgo programs are written in *.da* files. Each file consists of one or more process definitions and a *main* function. A process *p* is defined by extending the class *process*.

```
class Bob(process):
  ...

class Alice(process):
  ...

def main():
  ...
```

The process body contains a set of methods and handlers. A special *setup* method may be used to initialize data before the process is started. All arguments of the method automatically become instance variables of the process. If additional instance variables are required, it is possible to define them in the setup method. Another special method is the *run* method. It is defined for carrying out the main flow of execution. A process is alive as long as this *run* method is running, otherwise the process is terminated

and thus may no longer send or handle messages. In addition to these special methods, the process body may consist of additional helper methods or receive handlers. It is important to remember that receive handlers are only triggered if the flow of control reaches a yield point or an await statement, as described in paragraph Synchronisation.

The following example shows a process body. The setup method takes a process identifier as a parameter, which is later accessible using the variable *alice*. Another instance variable, the Boolean flag *greeted*, is also defined. The *run* method contains an await statement, therefore the subsequent receive handler may be triggered until the *run* method has terminated.

```
class Bob(process):
  def setup(alice:da.common.ProcessId):
    self.greeted = False

  def run():
    send(('Hi Alice',), to=alice)

    if await(self.greeted == True):
      send(('Bye',), to=alice)

  def receive(msg=('Hi Bob',), from_=p):
    output("Hi Bob received from", p)
    self.greeted = True
```

Below the process bodies is the entry point of a DistAlgo program, the *main* method. It is used to create, set up, and start processes. The *new* statement creates processes and returns the corresponding process identifiers. With the argument *num*, the number of processes to create may be specified. The setup statement is used to pass arguments to the processes. The statement executes the setup method of the corresponding processes. Then, as a third step, the processes may be started with the *start* statement. This executes the *run* method of the corresponding processes.

The following example creates three *Bob* processes and one *Alice* process. We can see the benefits of this three-part process creation. The *Bob* processes need the process identifier of the *Alice* process. Assuming that the *Alice* process also needs the process identifiers of the *Bob* processes, this would not be directly possible with a one-part process creation, because the *setup* statement is the only way to pass data to processes besides normal messages.

```
def main():
  alice = new(Alice, num=1)
  bobs = new(Bob, num=3)

  setup(alice, (n_bobs,))
  setup(bobs, (alice,))

  start(alice)
  start(bobs)
```

Compiling and Running. To execute DistAlgo source code in a *.da* file a computer is required that has the DistAlgo compiler installed. The command `python3 -m da <SOURCE>.da` is used to compile the source code. Subsequently, the program is executed automatically [4].

Nodes. DistAlgo processes are executed on nodes. By default, all processes are executed on the same unnamed node. But it is also possible to execute the processes on multiple nodes. To distinguish between them, a name is assigned to each of them. They might also run on different machines that are connected through the Internet. To run a DistAlgo program on multiple nodes, some modifications to the *main* method are necessary. It has to be specified on which node which processes are executed. For this purpose the *at* argument of the *new* statement is used. In the following example the three *Bob* processes should be executed on a node with the name *BobNode*. The *Alice* process should be executed on *AliceNode*. For this purpose, an *at* argument is added to the respective *new* statement.

```
def main():
  alice = new(Alice, num=1, at='AliceNode')
  bobs = new(Bob, num=3, at='BobNode')

  setup(alice, (n_bobs,))
  setup(bobs, (alice,))

  start(alice)
  start(bobs)
```

To execute the program on the two nodes, two terminals are required. A node is named using the *-n* option. More information about running a DistAlgo program on multiple nodes can be found in the DistAlgo repository [4].

```
python3 -m da -n AliceNode helloworld.da
python3 -m da -n BobNode -D helloworld.da
```

A first difference to the pseudocode abstraction becomes already visible. DistAlgo processes are independent of each other and always have their own TCP interface. Even if they are running on the same node, the only way to communicate is regular message exchange, in contrast to the composition model no local events exist. Therefore, a layered architecture with modules as layers as described in the composition model is not possible. This difference influences the DistAlgo implementations as they may not be translated directly. The following chapters introduce some strategies to address this issue.

3 Preliminaries

We continue with the introduction to some Byzantine-fault tolerant algorithms. The specifications are taken from [1] and use the abstract model introduced in Chapter 2.

3.1 Byzantine Consistent Broadcast

The *Byzantine consistent broadcast* (Module 1) abstraction may be represented as a *module* using the composition model. Every instance of the module has a pre-defined sender process *s*, which broadcasts a message *m*. If another message has to be sent or if another process has to send a message, an additional instance is required. The interface of the module consists of two events. One of them, the *broadcast* event, may only be triggered once by the sender process *s*. The *deliver* event is triggered by a correct process, if a message was received, that was broadcast by a process. As faulty processes can deviate arbitrarily from the algorithm assigned to them, some properties are specified. An *algorithm* that implements this module must satisfy these properties.

The *validity* property guarantees that if a process *p* broadcasts a message *m*, then every correct process eventually delivers *m*. The second property, *no duplication*, allows correct processes to deliver at most one message. This constraint is appropriate since only one message may be sent per instance. *Integrity* ensures, that if a correct process delivers a message *m* with sender *p* and process *p* is correct, then *m* was previously broadcast by *p*. The fourth property is called *consistency* and it states that if some correct process delivers a message *m* and another correct process delivers a message *m'*, then is $m = m'$. In the following section, two algorithms are introduced which fulfill these properties.

3.1.1 Authenticated Echo Broadcast

Requirements. The amount of Byzantine processes f in the system should be smaller than $\frac{N}{3}$. *Authenticated perfect links* are used as a link abstraction.

Algorithm overview. The *Authenticated echo broadcast* (Algorithm 1) consists roughly of two rounds. After initialization, the sender process sends the message to every processes in the system. In a second round, the message from the sender process is resent by all processes to every processes again as an ECHO message. Once a process has received more than $\frac{N+f}{2}$ ECHO messages with the same message, the message is delivered.

Module 1 Interface and properties of Byzantine consistent broadcast.

Module:

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

Events:

Request: $\langle bcb, Broadcast \mid m \rangle$: Broadcasts a message *m* to all processes. Executed only by process *s*.
Indication: $\langle bcb, Deliver \mid p, m \rangle$: 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*.
BCB4: *Consistency:* If some correct process delivers a message *m* and another correct process delivers a message *m'*, then *m* = *m'*.

Algorithm 1 Authenticated Echo Broadcast.

Implements:

ByzantineConsistentBroadcast, **instance** *bcb*, with sender *s*.

Uses:

AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle bcb, Init \rangle$ **do**
    sentecho := FALSE
    delivered := FALSE
    echos := $[\perp]^n$

upon event $\langle bcb, Broadcast \mid m \rangle$ **do***
    forall $q \in \Pi$ **do**
        trigger $\langle al, Send \mid q, [\text{SEND}, m] \rangle$

upon event $\langle al, Deliver \mid p, [\text{SEND}, m] \rangle$ **such that** $p = s$ **and** *sentecho* = FALSE **do**
    sentecho := TRUE
    forall $q \in \Pi$ **do**
        trigger $\langle al, Send \mid q, [\text{ECHO}, m] \rangle$

upon event $\langle al, Deliver \mid p, [\text{ECHO}, m] \rangle$ **do**
    if $echos[p] = \perp$ **then**
        $echos[p] := m$

upon exists $m \neq \perp$ **such that** $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$ **and** *delivered* = FALSE **do**
    delivered = TRUE
    trigger $\langle bcb, Deliver \mid s, m \rangle$

* only sender process *s*

Correctness. It follows from the $f < \frac{N}{3}$ assumption that there are more than $\frac{N+f}{2}$ correct processes in the system. If the sending process is correct and sends a SEND message, then more than $\frac{N+f}{2}$ processes will receive it and send an ECHO message. Hence, more than $\frac{N+f}{2}$ processes will receive an ECHO message with the same message *m*. The messages are not lost or have not been sent by another process, as there are also requirements for the links. The correct processes will receive enough ECHO messages to deliver the message *m*. This proves the *validity* property. *No duplication* follows directly from the algorithm. If the flag *delivered* is TRUE, no further message may be delivered. If a correct process delivers a message *m* with sender *s*, then the process has received ECHO messages containing *m* from more than $\frac{N+f}{2}$ processes. Among these processes, at least one process *p* is correct, since these processes form a Byzantine quorum. Thus *p* has received a SEND message from *s*. If *s* is correct, then *s* has broadcast *m*, which shows *integrity*. *Consistency* follows from a property of Byzantine quorums. Two Byzantine quorums overlap in at least one correct process. If a correct process delivers a message *m*, then it received ECHO messages with *m* from more than $\frac{N+f}{2}$ processes. This set of processes is a Byzantine quorum. If another correct process *p'* delivers a message *m'*, then there is another Byzantine quorum of ECHO messages containing *m'*. The two quorums overlap in a correct process, which sent all ECHO messages with the same message, hence $m = m'$.

3.1.2 Signed Echo Broadcast

The *Authenticated echo broadcast* sends a quadratic number of messages over the links. To reduce this amount, a cryptographic *digital signature scheme* may be used. This section introduces another algorithm which fulfills the properties of Byzantine consistent broadcast but only sends a linear number of messages over the links.

Requirements. As in the algorithm before, the *Authenticated perfect link* abstraction is used and the amount of Byzantine processes in the system is limited to $f < \frac{N}{3}$.

In addition, a *digital signature scheme* is required that provides two functions *sign* and *verifysig*. A process may get a signature of a message using the *sign* function with its process identifier. On the other hand, a process may verify a signature of an incoming message by calling the *verifysig* function, which returns a Boolean value.

Algorithm overview. In a first round of *Signed echo broadcast* (Algorithm 2), the message is sent from the sender process to all processes. Instead of sending the ECHO messages in the second round to all processes as in the algorithm before, the ECHO messages are only sent back to the sender, however, a signature is added. If the sender has received a message *m* in more than $\frac{N+f}{2}$ ECHO messages, it sends a FINAL message to all processes. This message also contains the signatures of all ECHO messages. Any process that receives the FINAL message may then use the signatures to verify that the sender process has received enough ECHO messages to deliver the message.

Correctness. The FINAL message with the signatures of the ECHO messages contains indirectly the same information as the ECHO messages sent to all processes in the *Authenticated echo broadcast*. If a correct process delivers a message *m*, it has received a FINAL message with $\frac{N+f}{2}$ valid signatures. The *digital signature scheme* guarantees that the sender process has received an ECHO message containing *m* from at least $\frac{N+f}{2}$ processes. With this observation, the proof of the *Authenticated echo broadcast* may be used to show the four properties.

3.2 Byzantine Reliable Broadcast

To guarantee that a correct process delivers a message if and only if every other correct process delivers a message, a fifth property may be added to the properties of *Byzantine consistent broadcast*. It is called *totality*. With this fifth property, we define *Byzantine reliable broadcast* (Module 2). As before, only

Algorithm 2 Signed Echo Broadcast.

Implements:
ByzantineConsistentBroadcast, **instance** *bcb*, with sender *s*.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle bcb, Init \rangle$ **do**
$sentecho := \text{FALSE}$
$sentfinal := \text{FALSE}$
$delivered := \text{FALSE}$
$echos := [\bot]^n$
$\Sigma := [\bot]^n$

upon event $\langle bcb, Broadcast \mid m \rangle$ **do***
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{SEND}, m] \rangle$

upon event $\langle al, Deliver \mid p, [\text{SEND}, m] \rangle$ **such that** $p = s$ **and** $sentecho = \text{FALSE}$ **do**
$sentecho := \text{TRUE}$
$\sigma := sign(self, bcb||self||\text{ECHO}||m)$
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{ECHO}, m, \sigma] \rangle$

upon event $\langle al, Deliver \mid p, [\text{ECHO}, m, \sigma] \rangle$ **do***
if $echos[p] = \bot$ **and** $verifysig(p, bcb||\text{ECHO}||m, \sigma)$ **then**
$echos[p] := m$
$\Sigma[p] := \sigma$

upon exists $m \neq \bot$ **such that** $\#(p \in \Pi \mid echos[p] = m) > \frac{N+f}{2}$ **and** $sentfinal = \text{FALSE}$ **do***
$sentfinal := \text{TRUE}$
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{FINAL}, m, \Sigma] \rangle$

upon event $\langle al, Deliver \mid p, [\text{FINAL}, m, \Sigma] \rangle$ **do**
if $\#(\{p \in \Pi \mid \Sigma[p] \neq \bot \text{ and } verifiysig(p, bcb||p||\text{ECHO}||m, \Sigma[p])\}) > \frac{N+f}{2}$ **and** $delivered = \text{FALSE}$ **do**
$delivered := \text{TRUE}$
trigger $\langle bcb, Deliver \mid s, m \rangle$

* only sender process *s*

one message may be broadcast and the sender process is pre-determined. Combining the *consistency* and *totality* property, a simpler property arises. It states that if a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Module 2 Interface and properties of Byzantine reliable broadcast.

Module:

Name: ByzantineReliableBroadcast, **instance** *brb*, with sender *s*.

Events:

Request: $\langle brb, Broadcast \mid m \rangle$: Broadcasts a message m to all processes. Executed only by process s.
Indication: $\langle brb, Deliver \mid p, m \rangle$: Delivers a message m broadcast by process p.

Properties:

BRB1: *Validity:* If a correct process p broadcasts a message m, then every correct process eventually delivers m.
BRB2: *No duplication:* Every correct process delivers at most one message.
BRB3: *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.
BRB4: *Consistency:* If some correct process delivers a message m and another correct process delivers a message m', then $m = m'$.
BRB5: *Totality:* If some message is delivered by any correct process, every correct process eventually delivers a message.

3.2.1 Authenticated Double-Echo Broadcast

The *Authenticated double-echo broadcast* (Algorithm 3) is an extension of the *Authenticated echo broadcast*, to additionally satisfy *totality*. This is achieved by adding a third round of message exchange.

Requirements. The requirements are the same as for the *Authenticated echo broadcast*. As links, the *Authenticated perfect link* abstraction is used. The number of faulty processes f should be smaller than $\frac{N}{3}$.

Algorithm overview. First, the sender process sends the message to all processes. Then, the processes send an ECHO message to all processes. Once a process has received a Byzantine quorum of ECHO messages, it sends a READY message to all processes to indicate that it is ready to deliver. In addition, the algorithm includes an amplification step for the READY messages. If a process has received READY messages from more than f processes, it sends a READY message to all processes itself. This mechanism is required to satisfy the *totality* property.

Correctness. To show the *validity*, *no duplication*, and *integrity* properties, the arguments of the proof of *Authenticated echo broadcast* may be used. If some correct processes send a READY message, then they all send the same message m. This follows from the *consistency* property of the *Authenticated echo broadcast*. *Consistency* of the *Authenticated double-echo broadcast* follows from this observation. If some correct process delivers a message m, then the process received READY messages from more than $2f$ processes. Thus more than f correct processes sent a READY message to the process. As they are correct, they send a READY message by the amplification step or after receiving enough ECHO messages. In both cases, they eventually deliver the message m. This shows the *totality* property.

Algorithm 3 Authenticated Double-Echo Broadcast.

Implements:
ByzantineReliableBroadcast, **instance** *brb*, with sender s.

Uses:
AuthPerfectPointToPointLinks, **instance** al.

upon event $\langle brb, Init \rangle$ **do**
$sentecho := \text{FALSE}$
$sentready := \text{FALSE}$
$delivered := \text{FALSE}$
$echos := [\bot]^n$
$readys := [\bot]^n$

upon event $\langle brb, Broadcast \mid m \rangle$ **do***
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{SEND}, m] \rangle$

upon event $\langle al, Deliver \mid p, [\text{SEND}, m] \rangle$ **such that** $p = s$ **and** $sentecho = \text{FALSE}$ **do**
$sentecho := \text{TRUE}$
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{ECHO}, m] \rangle$

upon event $\langle al, Deliver \mid p, [\text{READY}, m] \rangle$ **do**
if $readys[p] = \bot$ **then**
$readys[p] := m$

upon event $\langle al, Deliver \mid p, [\text{ECHO}, m] \rangle$ **do**
if $echos[p] = \bot$ **then**
$echos[p] := m$

upon exists $m \neq \bot$ **such that** $\#(\{p \in \Pi \mid echos[p] = m\}) > \frac{N+f}{2}$ **and** $sentready = \text{FALSE}$ **do**
$sentready := \text{TRUE}$
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{READY}, m] \rangle$

upon exists $m \neq \bot$ **such that** $\#(\{p \in \Pi \mid readys[p] = m\}) > f$ **and** $sentready = \text{FALSE}$ **do**
$sentready := \text{TRUE}$
forall $q \in \Pi$ **do**
trigger $\langle al, Send \mid q, [\text{READY}, m] \rangle$

upon exists $m \neq \bot$ **such that** $\#(\{p \in \Pi \mid readys[p] = m\}) > 2f$ **and** $delivered = \text{FALSE}$ **do**
$delivered := \text{TRUE}$
trigger $\langle brb, Deliver \mid s, m \rangle$

* only sender process s

3.3 Byzantine Consistent Channel

The two broadcast abstractions we considered so far may send at most one message. The *Byzantine consistent channel* abstraction (Module 3) introduced in this section allows us to send multiple messages without specifying the sender beforehand. Again, the abstraction is represented as a module in our composition model. The interface consists of two events, a *broadcast* and *deliver* event. Additionally, a label ℓ is used to distinguish the different roles a message may play. The implementation of the labels is determined by the algorithm under the condition that labels are unique on a process for all messages from a given sender. For example, a *per-sender sequence number* may be used. The four properties are similar to the *Byzantine consistent broadcast* abstraction, except that the labels are also included.

Module 3 Interface and properties of Byzantine consistent channel.

Module:

Name: ByzantineConsistentBroadcastChannel, **instance** *bcch*, with sender *s*.

Events:

Request: $\langle bcb, Broadcast \mid m \rangle$: Broadcasts a message m to all processes.
Indication: $\langle bcb, Deliver \mid p, l, m \rangle$: Delivers a message m with label ℓ broadcast by process p.

Properties:

BCCH1: *Validity:* If a correct process p broadcasts a message m, then every correct process eventually delivers m.
BCCH2: *No duplication:* For every process p and label ℓ, every correct process eventually delivers at most one message with label ℓ and sender p.
BCCH3: *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.
BCCH4: *Consistency:* If some correct process delivers a message m with label ℓ and sender s, and another correct process delivers a message m' with label ℓ and sender s, then $m = m'$.

3.3.1 Byzantine Consistent Channel

The idea of the *Byzantine consistent channel* (Algorithm 4) is to use multiple instances of the *Byzantine consistent broadcast* abstraction. Since the sender is pre-determined in this abstraction, an instance is created for every process. Therefore, for every process an instance exists, whereby the process is the sender. Since at most one message may be sent, a new instance with the corresponding sender process is created after every *deliver* event of the broadcast instance.

Requirements. The algorithm has no additional requirements. However, if for example the *Authenticated echo broadcast* is used as an implementation of the *Byzantine consistent broadcast* abstraction, the requirements for this algorithm have to be considered.

Algorithm overview. A per-sender sequence number is used as a label implementation. It is set to 0 for every process during initialization. Then an instance of *Byzantine consistent broadcast* is created for every process. The instances have a unique name, which includes the sender process and the sequence number of this process. In the case of a broadcast event on the channel, the broadcast event of the instance, where the process itself is the sender, is triggered. Since an instance may only broadcast one message, a *ready* flag is used. If an instance of *Byzantine consistent broadcast* delivers, the channel also delivers. Then, a new instance is created for this process and the corresponding sequence number is incremented. If the process is also the sender process of the new instance, then the *ready* flag is set to TRUE, as the new instance may broadcast another message.

Correctness. Together with the properties of the labels and the *Byzantine consistent broadcast* abstraction, the four properties may be shown directly.

Algorithm 4 Byzantine Consistent Channel.

Implements:
ByzantineConsistentChannel, **instance** *bcch*.

Uses:
ByzantineConsistentBroadcast, multiple instances.

upon event $\langle bcch, Init \rangle$ **do**
$n := [0]^n$
ready := FALSE
forall $p \in \Pi$ **do**
Initialize a new instance of ByzantineConsistentBroadcast with sender p named *bcb.p.n[p]*

upon event $\langle bcch, Broadcast \mid m \rangle$ **such that** *ready* = TRUE **do**
trigger $\langle bcb.self.n[self], Broadcast \mid m \rangle$
ready := FALSE

upon event $\langle bcb.p.n[p], Deliver \mid p, m \rangle$ **do**
trigger $\langle bcch, Deliver \mid p, n[p], m \rangle$
$n[p] := n[p] + 1$
Initialize a new instance of ByzantineConsistentBroadcast with sender p named *bcb.p.n[p]*
if $p = self$ **then**
ready := TRUE

4 Design and Implementation

For the implementation of the algorithms in DistAlgo, we will represent the algorithms implementing the modules as DistAlgo processes. Even if this seems unexpected, this choice is reasonable. Similar to the algorithms in our abstract model, DistAlgo processes contain definitions of event handlers and methods. However, a DistAlgo process represents a process in our system model as well, since a DistAlgo process is connected to others by links.

4.1 Byzantine Consistent Broadcast

4.1.1 Authenticated Echo Broadcast

The *initialization event* consists of two flags *sentecho* and *delivered* that are set to FALSE. In addition, a variable *echos* is initialized with FALSE for all processes in the system. This event is realized conveniently in DistAlgo by using the *setup* method. A Python dictionary is used as data structure for *echos*. To send messages to all processes later on, the identifiers of all processes are required. For this purpose, the argument *procs* is used. The number of processes n and the number of Byzantine processes f are used to calculate the bound whether sufficiently many ECHO messages have been received to deliver or not. Since the bound is a static value, it is pre-calculated and stored in the variable *quorumbound*, so it only needs to be calculated once. The argument *sender* contains the identifier of the pre-defined sender process. The last argument *msg* is an empty string by default, it only contains the message to be broadcast for the sender process. Eventually, if the process is the sender, the broadcast helper method is directly called.

```
def setup(procs:set, n:int, f:int, sender:da.common.ProcessId,
    msg:str = ''):
  self.sentecho = False
  self.delivered = False
  self.echos = dict((p, False) for p in procs)
  self.is_sender = True if sender == self else False
  self.quorumbound = (self.n + self.f) / 2

  if self.is_sender:
    broadcast()
```

Also the *broadcast* method is similar to the pseudocode counterpart. However, it should be mentioned that this is implemented as a helper method and not as an event. The reason for this choice and the resulting consequences are discussed in Chapter 5.

```
def broadcast():
  send(('SEND', self.msg), to = procs)
```

The following receive handler is used to handle received SEND messages. The SEND message is sent by the sender process. This is verified using the *from_* argument. The other condition, that no ECHO messages have been sent so far, is directly tested in the handler definition in the pseudocode implementation. This is not possible in DistAlgo, hence an if statement is used.

```
def receive(msg=('SEND', m), from_= self.sender):
  if not(self.sentecho):
    self.sentecho = True
    send(('ECHO', m), to= self.procs)
```

To handle the ECHO messages another handler is defined as in pseudocode. In this handler, the *from_* argument is no longer used to determine whether the sender of the message is a certain process. In this case, the identifier of the sender is stored in the variable *p*. Within the following block, the identifier can be accessed using this variable *p*. Since the Python data structure dictionary is used for the variable *echos*, the messages are stored in a similar way as in the pseudocode. At the end, a helper method *check_echos_received* is called. The underlying idea is explained in the next paragraph.

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

    check_echos_received()
```

The helper method contains an extensive condition. The condition corresponds to the condition of the last event handler specified in the pseudocode. First, it is verified that the flag *delivered* is FALSE. Then a quantification is used to check whether a message *m* from *echos* exists which fulfills the condition in the *has* argument. With an aggregation, it is checked if the given set is larger than the variable *bound*. It is also verified that *m* is a message and not the initial value FALSE. The aggregated set is formed by a comprehension. It consists of all processes in the system from which the message *m* was received in an ECHO message. If the complete condition holds, the helper method *deliver* is called. In this paragraph, it becomes clear that DistAlgo queries provide an elegant way to implement extensive conditions.

In the pseudocode, this condition was defined as an event handler. Since in DistAlgo only receive handlers for messages may be defined, a direct translation is not possible. However, given that the condition only depends on the variables *echos* and *delivered*, it is sufficient to check the condition only in case the variables have been changed. The helper method call in the receive handler above is used for this purpose.

```
def check_echos_received():
  if not(self.delivered) and
      some(m in self.echos.values(), has=(
      len(setof(p, p in procs, self.echos[p] == m)) >
      self.quorumbound and m != False) ):
    deliver(m)
```

The helper method *deliver* sets *delivered* to TRUE and outputs the message in the terminal.

```
def deliver(m: str):
  self.delivered = True
  output(m)
```

As in every DistAlgo process, a *run* method is required. In this case, a simple await statement is used. It waits indefinitely, as FALSE never becomes TRUE, nevertheless incoming messages are handled and the corresponding receive handlers are triggered.

```
def run():
  await(False)
```

To create and start the processes, we use the *main* function. All processes in the system are of the same class *AuthEchoBroadcast*, whose methods and receive handlers are described above. The sender process is also of the same class, but it has other arguments for the setup statement, especially the message being broadcast, thus the setup statement is called separately for this process. The algorithm uses the *Authenticated perfect link* abstraction as links. To use TCP for the message exchange between processes, a config statement is used. However, TCP does not fully comply with an *Authenticated perfect link* abstraction; in particular, *authenticity* is not guaranteed by TCP [5]. This issue is discussed in Chapter 5.

```
def main():
  config(channel={'reliable','fifo'}) # use TCP

  nprocesses = 10
  f = 2
  message = 'This is a test message.'

  procs = new(AuthEchoBroadcast, num= nprocesses)
  procs_without_sender = procs.copy()
  sender = procs_without_sender.pop()

  setup(procs_without_sender, (procs, nprocesses, f, sender))
  setup(sender, (procs, nprocesses, f, sender, message))

  start(procs)
```

Alternative. Await statements can also be used as internal events in DistAlgo. The condition that is checked in the event handler in the pseudocode may also be placed in an await statement. Since an effort was made to implement the await statements efficiently, the performance is not considerably lower [6].

The *check_echos_received* method and the corresponding calls are no longer needed. This alternative version does not use busy waiting since the block in the while loop is only executed once. When the process starts, the *run* method is called. Then, the await statement waits until the condition for delivering the message becomes TRUE. The deliver helper method is called, which sets the variable *delivered* to FALSE, which will exit the while loop.

```
def run():
  while not self.delivered:
    await(some(m in self.echos.values(), has=(
      len(setof(p, p in procs, self.echos[p] == m)) >
      self.quorumbound and m != False) ))
    deliver(m)
```

The complete implementation can be found in the appendix.

4.1.2 Signed Echo Broadcast

The initial step of implementing the pseudocode of the *Signed echo broadcast* algorithm is to provide the functions *sign* and *verifysgn*. For this purpose, an ordinary Python class *Crypto* is imported. This class provides *sign* and *verifysgn* as static methods. The actual implementation is irrelevant, it is only assumed that the methods fulfill the requirements of a *digital signature scheme*. The initialization and the *broadcast* method are implemented analogously to the *Authenticated echo broadcast*. Additionally, another Python dictionary *sigma* is initialized, in which the signatures of the ECHO messages are stored later on.

```
def setup(procs:set, n:int, f:int, sender:da.common.ProcessId, msg:str = ''):
  self.sentecho = False
```

```
    self.sentfinal = False
    self.delivered = False
    self.echos = dict((p, False) for p in procs)
    self.sigma = dict((p, False) for p in procs)
    self.is_sender = True if sender == self else False
    self.quorumbound = (self.n + self.f) / 2

    if self.is_sender:
      broadcast()

  def broadcast():
    output('BROADCAST \'', self.msg, '\'')
    send(('SEND', self.msg), to= procs)
```

To create a signature, the *sign* method of the *Crypto* class is used. The ECHO messages are no longer sent to all processes in the system, but only back to the sender process.

```
  def receive(msg=('SEND', message), from_= self.sender):
    if not(self.sentecho):
      self.sentecho = True

      signature = Crypto.sign(self, repr(self) + 'ECHO' + message)
      send(('ECHO', message, signature), to= self.sender)
```

The following receive handler is only relevant to the sender process because only this process receives ECHO messages. The received signatures are stored in the variable *sigma*. With every modification of the *echos* variable, it is checked if sufficiently many ECHO messages were received. If so, the FINAL message including the signatures is sent to all processes.

```
  def receive(msg=('ECHO', message, signature), from_= p):
    if self.is_sender:
      if self.echos[p] == False and
          Crypto.verifysig(source, repr(source) + 'ECHO' + message, signature):
        self.echos[p] = message
        self.sigma[p] = signature

        check_echos_received()

  def check_echos_received():
    if not(self.delivered) and some(m in self.echos.values(), has=(
        len(setof(p, p in procs, self.echos[p] == m)) > self.quorumbound and
        m != False and not(self.sentfinal)) ):
      send_final(m)

  def send_final(m: str):
    self.sentfinal = True
    send(('FINAL', m, self.sigma ), to= self.procs)
```

If a process receives a FINAL message, it verifies the received signatures. If there are sufficiently many valid signatures for this message, then the message is delivered. Similarly, no event handler is used to check the condition. Instead, it is checked immediately after a FINAL message has been received.

```
  def receive(msg=('FINAL', message, sigma), from_= source):
    if not(self.delivered) and len(setof(p, p in self.procs, sigma[p] and
        Crypto.verifysig(p, repr(p) + 'ECHO' + message, sigma[p]))) > self.quorumbound:
      self.delivered = True
      output(message)
```

The *run* method and the *main* function are the same as in the *Authenticated echo broadcast* implementation.

4.2 Byzantine Reliable Broadcast

The description of the initialization, the *broadcast* method, and the receive handler for handling the SEND and ECHO messages is omitted since these are almost identical to the corresponding structures in the *Authenticated echo broadcast* implementation.

The READY message receive handler is analogous to the pseudocode. Then, two helper methods are defined, which are required later. The method *send_ready* sends the message to all processes in the system and sets the flag *sentready* to TRUE to ensure that READY messages are only sent once per process. The method *deliver* sets *delivered* to TRUE and outputs the given message in the terminal.

```
def receive(msg=('READY', message), from_= source):
  if self.readys[source] == False:
    self.readys[source] = message

def send_ready(m: str):
  self.sentready = True
  send(('READY', m ), to= self.procs)

def deliver(m: str):
  self.delivered = True
      output(m)
```

Instead of checking the three conditions when receiving specific messages, a multi-part await statement is used in this case. The only reason for this choice is the readability of the code. Another option is to write helper methods that check the conditions and call them whenever a relevant message is received as in the implementations before. The while loop does not involve busy waiting since most of the time the process waits for one of the three expressions to become TRUE. The first part represents the event in the pseudocode, which sends the READY messages if sufficiently many ECHO messages are received. The second part is the amplification step, as described in the previous chapter. The last part delivers the message if there are sufficiently many processes from which a READY message with a certain message has been received, which was represented as the last event in the pseudocode implementation. All three parts use the helper methods described above.

```
def run():
  while not(self.delivered):
    if await(some(m in self.echos.values(), has=(
        len(setof(p, p in procs, self.echos[p] == m)) > self.quorumbound and
        m != False and not(self.sentready)) ):
      send_ready(m)
    elif some(m in self.readys.values(), has=(
        len(setof(p, p in procs, self.readys[p] == m)) > self.f and
        m != False and not(self.sentready)) ):
      send_ready(m)
    elif some(m in self.readys.values(), has=(
        len(setof(p, p in procs, self.readys[p] == m)) > 2 * self.f and
        m != False) ):
      deliver(m)
```

The *main* function is omitted, as it is the same as in the algorithms described before.

4.3 Byzantine Consistent Channel

Until now, the pseudocode of the algorithms consisted of two layers, the algorithm layer, which implemented the algorithm, and the link layer, which was implemented by a *Authenticated perfect link* abstraction. Even if DistAlgo does not support a layered architecture due to the lack of local events, this was not an issue for the implementations, since the underlying layer was the link layer. This layer is available in every DistAlgo process by using the send and receive statements as described in Chapter 2.

However, the *Byzantine consistent channel* algorithm uses a third layer of multiple, dynamically generated instances of *Byzantine consistent broadcast*. For the implementation in DistAlgo, we have to find a way to exchange information between those two layers directly without using messages.

The main idea is to integrate the functionality of the *Byzantine consistent broadcast* directly in the process body of the *Byzantine consistent channel*. The dynamically created instances of *Byzantine consistent broadcast* are only virtual. More precisely, a label is added to all messages belonging to the third layer in the pseudocode. It follows from the properties of the label that an instance is uniquely identified by a label. Therefore the receiving DistAlgo process is able to determine the virtual instance of the received message. In this implementation, the *Authenticated echo broadcast* implementation is used, but another algorithm implementing *Byzantine consistent broadcast* would also be possible.

The first part of the *setup* method initializes the data structures for the *Byzantine consistent channel* itself. The second part initializes the data structures of the *Authenticated echo broadcast* algorithm. However, to distinguish the instances, different data structures are used than in the original implementation. The former flags are modeled as sets, while the variable *echos* is still a Python dictionary. The key now consists of a label and a process. The label is implemented as per sender sequence number, thus a label consists of the identifier of the sender process and the corresponding sequence number. A nested for-loop is necessary since it requires for every process an instance, such that for each process an instance exists with this process as a sender. This is the purpose of the outer loop. The *Authenticated echo broadcast* itself requires a FALSE in the variable *echos* for every process at the beginning.

```
def setup(procs:set, nprocesses:int, f:int):
  self.n = dict((p, 0) for p in procs)
  self.ready = True
  self.quorumbound = (self.nprocesses + self.f) / 2

  self.sentecho = set()
  self.delivered = set()
  self.echos = dict()
  for sender in procs:
    for p in procs:
      self.echos[(0, sender), p] = False
```

To start a broadcast, a normal message exchange is used. With the following receive handler, the *main* function may send a BROADCAST message to a process, which then broadcasts the message. In the pseudocode, the *ready* flag is set to FALSE and the *broadcast* event of the corresponding instance is triggered. Since there are no instances in this DistAlgo implementation, a SEND message with the associated label is sent. In this case, this is only the sequence number, since the identifier of the sender process is also the sender of the SEND message and may be obtained by using the *from_* argument.

```
def receive(msg=('BCCH', 'BROADCAST', m)):
  if self.ready:
    bcb_broadcast(m)
    self.ready = False

def bcb_broadcast(m: str):
  send(('BCB', 'SEND', m, self.n[self]), to = procs)
```

The following receive handlers are copied from the *Authenticated echo broadcast* and adapted to the new data structures. The variable *sentecho* is modeled as a set. If a label, consisting of the identifier of the sender process and the associated sequence number, is in the set, the corresponding flag is TRUE, otherwise it is FALSE. This implementation has several advantages. Since the flag is initially FALSE, no action is required whenever a new sequence number is used. In addition, membership checks are efficient in Python and a more complicated dictionary data structure is avoided. The variable *delivered* is modeled analogously.

```
def receive(msg=('BCB', 'SEND', m, sequence_number), from_= sender):
  if not((sequence_number, sender) in self.sentecho):
    self.sentecho.add((sequence_number, sender))
    send(('BCB', 'ECHO', m, sender, sequence_number), to= self.procs)

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

The next method is copied and adapted to the new data structure again. The Python method *items* returns key-value pairs as tuples in a list. In this case the key consists of a process, from which the ECHO message was sent, and the corresponding label. The value contains the message. With this information, it is verified that *delivered* with this label is still false. Furthermore, it is checked if the process has received ECHO messages with this label and a message other than FALSE from sufficient processes. If this condition is TRUE, the *bcb_deliver* helper method is called.

```
def check_echos_received():
     if some(item in self.echos.items(), has=(
      not(item[0][0] in self.delivered) and
      len(setof(tup, tup in self.echos.keys(), self.echos[tup] == item[1])) >
        self.quorumbound and
       item[1] != False) ):
    bcb_deliver(item[0][0], item[1])
```

The *bcb_deliver* method adds the label to the set *delivered*. This has the same effect as setting the flag *delivered* to TRUE in the *Authenticated echo broadcast*. In the pseudocode implementation, a new instance would then be created for this sender process. Without instances, only the data structures for the new label have to be modified. In other words, the variable *echos* is initialized to FALSE for every process with this new label. The two flags *sentecho* and *delivered* do not need to be adjusted, as they are modeled as sets as described before. If the process itself was the sender of the message, then *ready* is set to TRUE again, allowing another message to be broadcast. It is important to remember that a broadcast sends the message back to the sender as well.

```
def bcb_deliver(label:tuple, m:str):
       self.delivered.add(label)

  bcch_deliver(label, m)

def bcch_deliver(label:tuple, m:str):
      new_sequence_number = n[label[1]] + 1

  for p in procs:
    self.echos[(new_sequence_number, label[1]), p] = False
  if label[1] == self:
    self.ready = True

  output(m, label)
```

The *main* method consists of the creation of the processes, which is analogous as in the algorithms described before. Then, some messages are broadcast. The *main* function has also a TCP interface and may send messages to processes. In this case, four BROADCAST messages are sent, two of them directly after each other. There is not sufficient time for the process to broadcast the message, and *ready* will still be FALSE, therefore, this message will not be sent. The work statement waits for some seconds. During this time, the message is broadcast and the process will be ready again for the next message.

```
def main():
  ...
  procs = new(ByzantineConsistentChannel, num= nprocesses)
```

```
setup(procs, (procs, nprocesses, f))
start(procs)

sender = procs.pop()
sender2 = procs.pop()
send(('BCCH', 'BROADCAST', 'test'), to= sender)
send(('BCCH', 'BROADCAST', 'test2'), to= sender) # not ready yet

work()
send(('BCCH', 'BROADCAST', 'test3'), to= sender)
send(('BCCH', 'BROADCAST', 'test4'), to= sender2)
```

5 Conclusion

5.1 Discussion

The objective to implement some Byzantine-fault tolerant algorithms in DistAlgo was achieved. Furthermore, the DistAlgo code is quite similar to the pseudocode, several handlers could even be translated directly. Nevertheless, some issues arose which are discussed in this section.

Links. All the algorithms described before use the *Authenticated perfect link* abstraction. Up to this point, DistAlgo does not offer a way to exchange messages using a link that satisfies *authenticity*, for example TCP with TLS, as only UDP and TCP are offered. Alternatively, the authentication could be realized in the process definitions. Since this would make the algorithms more difficult to understand, this thesis did not pursue this idea any further. But even without complete correctness, the implementations allow us to examine the message flows and to understand the behavior of the algorithms more easily.

Layered architecture. DistAlgo does not support a layered architecture as in the composition model. This is mainly a problem in the implementation of the *Byzantine consistent channel* algorithm. The pseudocode of the algorithm could not be implemented directly in DistAlgo. In practice, many implementations of similar algorithms collapse the notion of the instances and distinguish the instances simply by the label [1]. Nevertheless, the implementation is less structured and more difficult to understand.

Event handlers. Minor differences between the abstract model and DistAlgo have also some consequences. In DistAlgo, only event handlers in the context of messages exist. Events triggered if a condition is fulfilled do not exist. There are different approaches, two of them have been already used in the implementations above. On the one hand, helper methods may be used that check the condition and are always called, if a variable on which the condition depends changes. Another possibility, as in the implementation of the *Authenticated double-echo broadcast*, is to use an await statement.

Broadcast event. Since DistAlgo processes may only communicate using messages after setup, starting a broadcast is not trivial. In the implementations of the three *Byzantine consistent broadcast* algorithms, the message is passed directly as an argument during setup. Thus, sending a broadcast message using a normal message is avoided. However, it is important to note that the broadcast process may also include a helper method, which creates or fetches a message to be broadcast. Such an implementation has been avoided, as this conflicts with the goal of keeping the implementation as close as possible to the pseudocode. In the *Byzantine consistent channel* implementation, the approach using the *setup* method

is no longer possible since the sender is not pre-defined and there may be multiple messages to be broadcast. To keep things simple, the rather unattractive way that a BROADCAST message is sent to a process was chosen. As long as the process is correct and is not already busy, the process then broadcasts the received message.

Sender process. Some algorithms contain events that are only triggered on the sender process, such as the *broadcast* event in *Byzantine consistent broadcast* algorithms. In DistAlgo different types of processes may be defined, for example a class may be defined for the sender process and another for all other processes. As a result, the normal processes would be without methods that only the sender process will use. This approach was not chosen for two reasons. Firstly, because the differences to the pseudocode would be increased, and secondly, the class of the sender process would contain a lot of duplicated code, since the sender process would only be an extension of a normal process.

5.2 Future work

It will be interesting to improve the implementations. On the one hand, we could provide *authenticity* and thus make the correctness of the algorithms complete. On the other hand, some parts of the code became difficult to understand as a result of the adaptation to DistAlgo. Here we could make the implementation less bound to the pseudocode and therefore better adapt it to DistAlgo. As an example, consider the *Byzantine consistent channel* implementation, where we could simplify the structure if we completely abandon the concept of instances also in the method names and data structures. Another possibility is to extend the features of DistAlgo itself. Probably it makes more sense to implement a message exchange that satisfies the *Authenticated perfect link* abstraction directly in DistAlgo. This way, programming with DistAlgo remains as abstract as it is today. A major modification would be to allow a layered architecture and a composition model that allows multiple, even dynamically created instances running on a process. This would bring more flexibility and it would be easier to implement the described algorithms.

A Extra material

A.1 Running DistAlgo programs

The algorithms were developed and tested on Ubuntu 16.04 LTS.

1. **Python:** Install Python 3.5, 3.6 or 3.7
2. **DistAlgo:** Install DistAlgo with pip: `pip3 install pyDistAlgo`
3. **Repository:** Clone the repository:
   `https://gitlab.inf.unibe.ch/crypto-students/2020.bsc.roland.widmer.git`
4. **Run:** Navigate to a folder and follow the instructions in the corresponding *readme.md* file

More information can be found in the repository of the DistAlgo project [4].

A.2 DistAlgo example program

This is the complete DistAlgo program, which is described in parts in Chapter 2.

```
import sys
import da

class Bob(process):
  def setup(alice:da.common.ProcessId):
    self.greeted = False

  def run():
    send(('Hi Alice',), to=alice)
    if await(self.greeted == True):
      send(('Bye',), to=alice)

  def receive(msg=('Hi Bob',), from_=p):
    output("Hi Bob received from", p)
    self.greeted = True;

class Alice(process):
  def setup(n_bobs:int): pass

  def run():
    await(len(setof(p, received(('Bye',), from_= p))) == n_bobs)
    output('Bye from all Bobs received')

  def receive(msg=('Hi Alice',), from_=p):
    send(('Hi Bob',), to=p)

def main():
  n_bobs = 3

  alice = new(Alice, num= 1)
  bobs = new(Bob, num= n_bobs)
  setup(alice, (n_bobs,))
  setup(bobs, (alice,))
  start(alice)
  start(bobs)
```

A.3 Authenticated Echo Broadcast

This is the complete DistAlgo program of the *Authenticated echo broadcast* implementation described in Chapter 4.

```
class AuthEchoBroadcast(process):
  def setup(procs:set, n:int, f:int, sender:da.common.ProcessId, msg:str = ''):
    self.sentecho = False
    self.delivered = False
    self.echos = dict((p, False) for p in procs)
    self.is_sender = True if sender == self else False
    self.quorumbound = (self.n + self.f) / 2

    if self.is_sender:
      broadcast()

  def broadcast():
    send(('SEND', self.msg), to = procs)

  def receive(msg=('SEND', m), from_= self.sender):
    if not(self.sentecho):
      self.sentecho = True
      send(('ECHO', m), to= self.procs)

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

  def check_echos_received():
    if not(self.delivered) and some(m in self.echos.values(), has=(
        len(setof(p, p in procs, self.echos[p] == m)) > self.quorumbound
        and m != False) ):
      deliver(m)

  def deliver(m: str):
    self.delivered = True
    output(m)

  def run():
    await(False)

  def main():
    config(channel={'reliable','fifo'})

    nprocesses = 10
    f = 2
    message = 'This is a test message.'

    procs = new(AuthEchoBroadcast, num= nprocesses)
    procs_without_sender = procs.copy()
    sender = procs_without_sender.pop()
    setup(procs_without_sender, (procs, nprocesses, f, sender))
    setup(sender, (procs, nprocesses, f, sender, message))
    start(procs)
```

Bibliography

[1] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.

[2] Y. Liu, S. Chand, and S. Stoller, "Moderately complex paxos made simple: High-level executable specification of distributed algorithms," *PPDP '19: Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages 2019*, Mar. 2019, arXiv:1704.00082v4.

[3] Y. Liu, B. Lin, and S. Stoller, *DistAlgo Language Description*, https://www3.cs.stonybrook.edu/ liu/distalgo/language.pdf, Mar. 2017.

[4] *DistAlgo Readme.md on GitHub*, https://github.com/DistAlgo/distalgo/blob/master/README.md, Mar. 2020.

[5] *RFC 793 - Transmission Control Protocol*, https://tools.ietf.org/html/rfc793, Sep. 1981.

[6] Y. Liu, S. Stoller, and B. Lin, "From clarity to efficiency for distributed algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 39, no. 12, pp. 7–10, Jul. 2017, arXiv:1412.8461v4.

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.

Bern, 21.07.2020
Ort/Datum

R. Widmer
Unterschrift