UiO : Department of Informatics
University of Oslo

Case Studies of Modeling Distributed Algorithms in ABS

Daniel Rødskog
master thesis spring 2014

Case Studies of Modeling Distributed Algorithms in ABS

Daniel Rødskog

May 14, 2014

Abstract

This thesis presents four case studies in which well known distributed algorithms are implemented in the *abstract behavioral language* ABS. These algorithms are the *Dining Philosophers problem*, the *Paxos* consensus algorithm, the *Kademlia* distributed hash table, and the *Rarest piece* algorithm of the BitTorrent protocol.

The implemented models form the basis of an evaluation of the ABS language and its suitability for modeling such algorithms. The different selected algorithms require different strategies for synchronization, and it is shown that the ABS language copes well with this.

Contents

1 Introduction **1**

2 The ABS Language **3**
2.1 The functional layer . . . 3
2.1.1 Data type definitions . . . 3
2.1.2 Function definitions . . . 6
2.2 The imperative layer . . . 8
2.2.1 Interfaces . . . 8
2.2.2 Classes . . . 8
2.3 The distributed layer . . . 8
2.3.1 Cooperative scheduling in concurrent object groups . . . 9
2.3.2 Communication by asynchronous method calls . . . 9

3 The Dining Philosophers Problem **11**
3.1 Problem outline . . . 11
3.2 Naive ABS implementation . . . 13
3.3 Deadlock . . . 15
3.4 Starvation . . . 16
3.5 Solutions . . . 18
3.5.1 Anti-symmetry . . . 18
3.5.2 Servant process . . . 18
3.6 Chandy and Misra's "hygienic solution" . . . 19
3.6.1 Solution overview . . . 19
3.6.2 ABS implementation overview . . . 19
3.6.3 Forks as an algebraic data type . . . 20
3.6.4 `Philosopher` class overview . . . 21
3.6.5 Scheduling points . . . 22
3.6.6 The `thinking` and `eating` states . . . 22
3.6.7 The `hungry` state . . . 23
3.6.8 The fork request . . . 26
3.6.9 Testing the model . . . 26

4 Paxos **31**
4.1 Algorithm overview . . . 31
4.2 Agents . . . 32
4.3 Quorum of majority . . . 32
4.4 Reaching consensus . . . 32

4.4.1 The prepare phase . . . 33
4.4.2 The accept phase . . . 33
4.4.3 Progress . . . 34
4.5 Modeling Paxos in ABS . . . 34
4.5.1 Data types . . . 36
4.5.2 Agents . . . 37
4.5.3 The prepare phase . . . 39
4.5.4 The accept phase . . . 42
4.5.5 Learning the chosen value . . . 45
4.5.6 Testing the model . . . 46
4.6 Modeling faulty communication . . . 50
4.6.1 Implementation . . . 53
4.6.2 Testing . . . 59

5 Distributed Hash Tables: The Kademlia algorithm 63
5.1 Distributed hash tables . . . 63
5.1.1 Consistent hashing . . . 64
5.2 The Kademlia protocol . . . 64
5.2.1 The XOR metric . . . 64
5.2.2 The routing table . . . 65
5.2.3 Remote procedure calls . . . 67
5.2.4 The node lookup procedure . . . 68
5.2.5 Joining a network . . . 69
5.3 ABS implementation . . . 69
5.3.1 Basic data types . . . 70
5.3.2 The `Node` interface . . . 71
5.3.3 Extended node interface . . . 71
5.3.4 Beginning of the `Node` class . . . 72
5.3.5 Routing table data structure . . . 72
5.3.6 Comparing hashes . . . 75
5.3.7 Inserting contacts into the routing table . . . 76
5.3.8 Searching through the routing table . . . 78
5.3.9 The node lookup procedure . . . 80
5.4 Performance of the implementation . . . 83
5.4.1 The `xor` function . . . 84
5.5 Potential improvements of the model . . . 84

6 BitTorrent 87
6.1 BitTorrent Overview . . . 87
6.2 BitTorrent peer protocol . . . 88
6.3 ABS implementation . . . 89
6.3.1 Data types . . . 89
6.3.2 Defined functions . . . 90
6.3.3 Selecting a piece to download from a remote peer . . . 91
6.3.4 Handling requests . . . 93
6.4 Test run . . . 95

7 Conclusion **97**
7.1 Summary 97
7.2 Tool evalution 98

Appendices **103**

A The Dining Philosophers **105**
A.1 Complete implementation of naive solution 105
A.2 Complete implementation of hygienic solution 107

B Paxos **111**
B.1 Complete basic implementation 111
B.2 Faulty communication 115

C Kademlia **121**
C.1 Full implementation 121

D BitTorrent **133**
D.1 Full implementation 133

E Additional code listings **143**
E.1 Alternative XOR function 143

List of Figures

3.1 The dining philosophers 12
3.2 Sequence diagram illustrating the starvation of a process 17
3.3 Chandy/Misra *hygienic* philosophers entering a cyclic configuration 24
3.4 Hygienic philosopher giving up one fork while waiting for another 29

4.1 Two test simulations of the Paxos algorithm 49
4.2 Two different values being accepted 50
4.3 Intercepting an agent's messages 52
4.4 Example run of the Paxos model with faulty communication . . . 59

5.1 The longest common prefix of two IDs 65
5.2 The structure of a Kademlia routing table 66

6.1 A file being shared between three BitTorrent peers. 95

List of ABS Code Listings

3.1 Basic implementation of a dining philosopher 13
3.2 The fork class 14
3.3 A simple method for printing status messages 15
3.4 Beginning of the Chandy/Misra Philosopher class 21
3.5 The thinking and eating states of a hygienic philosopher 23
3.6 The `hungry` state of a hygienic philosopher 25
3.7 The fork request 26

4.1 Interfaces for Paxos agents 38
4.2 Declaration of the Proposer class and its fields 40
4.3 The proposer's run method – starting point of the Paxos algorithm 41
4.4 Head of the Acceptor class and the prepare method 42
4.5 The proposer's response method 44
4.6 The acceptor's accept method 44
4.7 The Paxos Learner class 47
4.8 A simple test of the Paxos module 48
4.9 AcceptorProxy's method for intercepting the prepare request from a basic proposer. 54
4.10 The FaultyProposer class 57
4.11 The FaultyAcceptor class 58

5.1 The Kademlia protocol expressed as an ABS interface 71
5.2 Extended Kademlia node interface 72
5.3 Header of the Node class 72
5.4 Data structure for Kademlia routing tables 74
5.5 Longest Unique binary Suffix of two integers 75
5.6 The Bitwise Exclusive Or of two integers 75
5.7 Helper functions on k-buckets 76
5.8 Splitting a k-bucket 77
5.9 Inserting a contact in the routing table 78
5.10 The main logic of the lookup procedure 82

6.1 Selecting a piece for download from a remote peer 92
6.2 Handling and enqueuing requests 94

Preface

This thesis concludes my time as a student at the University of Oslo. I have learned a lot during my years here, about informatics as well as life in general. I have met many great people, and especially I want to send regards to my fellow master students on the 8th floor.

Finally, I must thank my supervisors. Thank you Einar Broch Johnsen, especially for pulling a few strings for me when I wanted to study a semester abroad. Lastly, many thanks to my main supervisor Rudolf Schlatte, who has given me valuable feedback and guidance, listened to my suggestions, and shown me patience.

Chapter 1

Introduction

ABS is an *abstract behavioral specification* language for modeling distributed systems. It was presented by Johnsen et al. [10], and has been under development for some years at the University of Oslo. This thesis explores the suitability of ABS for modeling a number of well-known algorithms and systems.

Case studies

We chose the following algorithms as subjects for the ABS case studies.

- The Dining Philosophers problem, which is a relatively simple problem and fitting for an inital demonstration of the language. It is also a historically significant problem that demonstrates important concepts in synchronization. Different solutions will be discussed.
- Paxos, a well known distributed algorithm for obtaining concensus among processes.
- The Kademlia distributed hash table, which is a data structure designed for storing and retrieving key–value pairs across a big and complicated overlay network. It is an interesting data structure in itself, and can be used to measure the scalability of ABS models.
- The BitTorrent protocol, a well known peer-to-peer file sharing system that aims to at all times maximize the replication of the files among the participating peers.

For each of the chosen algorithms, after the required study of the literature, we chose the appropriate abstraction and created an ABS model. We tried to remove superfluous details from the models and bring forth the essences of the algorithms. Afterward we evaluated the available tool chain against the models that we created.

Structure

The rest of this thesis is structured as follows.

Chapter 2 gives an overview of the ABS language and explains the syntax and functionality of the features that will be used for the implementation of the algorithms.

Chapters 3 through 6 cover the implementations of the different algorithms. An overview of each case is provided, and the implementations are explained. Throughout these chapters are given listings of the ABS code that is being discussed. Certain aspects of the algorithms or ABS models are illustrated with sequence diagrams that are created with the sequence diagram visualizer of the ABS Eclipse plug-in [5].

Chapter 7 summarizes our finding and presents conclusions.

Chapter 2

The ABS Language

This chapter gives an explanation of the key features of the ABS language. The syntax of the language is explained and demonstrated with examples.

Overview

ABS was presented by Johnsen et al. [10] in 2010. The language is also described in *The ABS Language Specification*, a reference manual which is included in the ABS repository.

Syntactically the ABS language is very similar to Java. A user with experience with Java will immediately recognize the keywords and usage of symbols in ABS, although the two languages' functionality differ quite a bit in some respects.

ABS offers pure side effect free data types and functions, constructs for object-orientations, and a high level of abstraction for concurrency. The functionality of ABS can be divided into three different layers, which will be presented over the following sections.

2.1 The functional layer

At the lowest level of the language is the functional layer. This section describes how the most basic expressions of the ABS language are built up and evaluated. The main features at this level are algebraic data types and function definitions.

The features in the functional layer of ABS are completely side effect free. This means that expressions can be evaluated and executed, but not change the state of a running ABS program.

2.1.1 Data type definitions

The *algebraic data types* provide the most basic way of describing data in ABS. There are built-in types such as `Int`, `Bool` and `Strings`, for integers, boolean values and text strings respectively, and the users can define their own types as well.

The values of the algebraic data types are immutable, meaning that they cannot change, and they have no state. Unlike object, the values of the algebraic data types have no identity, so to equal values are not only equal – they are *the same*.

types are the *algebraic data types*, which include predefines types such as

One of the most important parts of the language are the algebraic data types.

Syntax

To define an algebraic data type, the **`data`** keyword is used. In the data type definition the user enumerates the *data type constructors*. These constructors describes every possible value that the data type can take. The following example shows how the simple boolean data type is defined with the two constructors `True` and `False`.

```
data Bool = True | False;
```

The data type constructors can also be given a list of parameters. The following shows the simple data type `Value` which is used in the implementation of Paxos in Chapter 4. The data type's only constructor, `Value`, has an integer parameter. Even though there is only one constructor of the data type, there can be constructed as many `Values` as there are integers – which is infinite.

```
data Value = Value(Int);
```

Two `Value` types are only equal if they hold the same integer. For instance, `Value(9) != Value(16)`. The parameters given to a data type constructor cannot be changed – the types are still immutable, and the types have no states and are not subject to side effects.

A type's constructor can also be given a parameter of the same type. This allows the user to define data structures such as singly linked lists and trees easily.

The parameters to the data type constructors can be given names, as shown below. Naming a parameter causes an *accessor function* to be automatically created for the constructor. In the `Contact` data type from the Kademlia implementation in Chapter 5, the constructor has two named parameters, `id` and `node`. For example given a `Contact` reference `c`, the ID of the contact can be accesses with the function call `id(c)`. More on functions in Section 2.1.2 on page 6.

```
data Contact = Contact(ID id, Node node);
```

An algebraic data types can be given *type parameters*, defined inside a pair of angle brackets, after the type's name. The type is then called a *Parametric Data Type*. Type parameters give data types a broader purpose, and are very useful for defining general data structures such as lists, sets and maps. The predefined type `List` is an example of a parametric data type.

```
data List<A> = Nil | Cons(A head, List<A> tail);
```

The type parameter `A` of the list type is used as the type of the parameters to the `Cons` constructor. The `List` data type can therefore be used for making any sort of lists: for example a list of integers has the type `List<Int>`.

TODO list builtin types here.

Usage and examples

The algebraic data types cover a wide range of uses, some of which have been mentioned in the examples above. The following is a brief recapitulation, and some usage examples are further elaborated.

Enumeration In their most basic form, the algebraic data types provide a way to enumerate values because constructors define the only values that a type can have. Without any internal structure, the algebraic data types give the user a safe way to express ideas precisely and make reasoning simple and clear. For instance, in Section 3.6.6, the current state of a Dining Philosopher is expressed with the data type `PState`, which which clearly signals the meaning of its different constructors.

```
data PState = Thinking | Hungry | Eating;
```

Record types By *record type* is meant a simple construct for encapsulating data, like a tuple. The data type `Contact` is an example of this: the constructor `Contact` takes a fixed number of parameters and can serve as a lightweight object – an object without any state or behavior. A record type can hold information the same way an object does, but being an algebraic data type in ABS, it has no identity and is immutable.

Data structures The algebraic data types provides the simplest way of defining data structures in ABS. A number of helpful general-purpose data structures are already defined in the language: The linked list type `List<A>` have already been mentioned. `Set<A>` is an ordered list in which the same value may only appear once. `Map<A,B>` provides a mapping from a set of keys of type `A`, to values of type `B`.

In the Kademlia implementation in Chapter 5, a more advanced data type is defined for the Kademlia nodes' routing tables. The `Table` data type exhibits usage of many of the aspects of the algebraic data types that have been mentioned. The purpose of the `EmptyTable` constructor is quite self explanatory, and should ease the understanding of the logic in the code that uses this data type. The constructor named `KBucket` holds the contacts that are to be stored in the routing table, and provides an accessor function to retrieve these contacts. The routing table has a tree-like structure, which is realized by the `Split` constructor that holds two sub-tables.

```
data Table = EmptyTable
           | KBucket(List<Contact> contacts, Int)
           | Split(Table, Table, Int);
```

An important aspect of using algebraic data type – as opposed to objects – for data structures, is the lack of side effects. A list can be passed around between different objects and processes, and it is still ensured that the local copy of the list remains unchanged, even when working in the imperative layer with side effects. The data type definitions are also much shorter and more concise than

class definitions. Moreover, the structures can be created on the fly by listing the values explicitly.

It is however, somewhat cumbersome to write down a long list of values solely using the constructors of the data type. Therefore the user can define a function with the same name as the data type, that defines how the data structure can be constructed. Defining such a function allow the user to use a special syntax called *n-ary constructors*.

An *n*-ary constructor is for instance available for the `List` data type. The two following expressions are equivalent:

```
Cons(4, Cons(8, Cons(15, Cons(16, Cons(23, Cons(42, Nil))))))
==
list[4, 8, 15, 16, 23, 42]
```

Type synonyms

In addition to defining new data types, the existing ones can be given *type synonyms*. This does not in itself provide more functionality to a program, but can ease the readability of the code. Especially for the more general data types such as integers or lists, a type synonym can be used to express the intended purpose of the data type in the given setting.

For example, the `ID` type that was shown as a parameter to the `Contact` constructor on page 4 is a type synonym for the type `Hash`, which itself is a synonym for `Int`.

```
type Hash = Int;
type ID = Hash;
```

A reference to a type which is a synonym for another can hold any value of the original type. The values are indistinguishable even if they are referenced by different type synonyms.

2.1.2 Function definitions

Functions in ABS can be used for implementing anything from mathematical operators to operations on data structures. Functions are side effect free, so calling a function can never alter the state of the program, unlike a method.

A function definition starts with the keyword **`def`**, followed by the function's return type, and name and potential parameters. The function body is an expression which is listed after an equals sign. The following example is the predefined `abs` function definition which utilizes the if-expression to return the absolute value of a rational number.

```
def Rat abs(Rat x) = if x > 0 then x else -x;
```

The function body only can contain a single expression, which may feel restrictive when one wants to define more complicated functions, for instance such as operating on a data structure. There are however a few constructs in the ABS language that come in handy for implementing larger functions.

The *if expression* has already been demonstrated in the previous example, and allows the function body to return different values depending on the given condition. If expressions can be nested to allow multiple different return values.

In case the conditions in nested if expressions are checking the same values, a *case expression* may be more suitable. With the case expression a type can be checked against multiple *patterns* at once. Each pattern can consist of data type constructors, literals and *bound variables* – names of parameters or other variables that are known at the place in the code where the case expression is used. But the case statement can also used with *unbound* variables.

The case expression is an important feature in the ABS language because it provides *pattern matching*: Unbound variables can be used to match expressions and extract only parts of their value.

The following example shows the implementation of the predefined function `length` which returns the length – or the number of nested `Cons` values – of a given list. The first branch of the case expression is the `List` constructor `Nil`; After `Nil` is an arrow (`=>`) which "points to" 0, meaning that if the parameter `list` is Nil then 0 is returned.

The next branch matches the constructor `Cons`, using two unbound variables. Because the `List` data type only has two constructors, this pattern must match if `list` is not `Nil`. The return value reuses one of the unbound variables in a recursive call the to the `length` function. This is the typical, idiomatic way to handle nested algebraic data types.

```
def Int length<A>(List<A> list) =
    case list {
        Nil => 0 ;
        Cons(p, l) => 1 + length(l) ;
   };
```

The implementation of the length function also demonstrates that a function definition can be recursive, and have type parameters just like algebraic data types.

Instead of unbound variables, values can also be matched with the special *underscore pattern* (`_`) if the matched value does not need to be reused on the right-hand side of the arrow. The underscore simply matches any value.

Another useful expression is **`let`**, which is used for defining a variable that can be reused in an expression: The **`let`** keyword is followed by a variable declaration, with a type and a variable name; the variable assignment; and, after the **`in`** keyword, another expression in which the declared value can be used.

For example the `index` function, which is defined in the BitTorrent implementation of Chapter 6, uses the let statement to reuse the return value of a recursive function call, to avoid calling the same function twice. The return value of the call to `index` is bound to the `Int` variable `i` which is is used twice in the succeeding if expression.

```
def Int index<A>(List<A> l, A e) =
    case l {
        Nil => -1;
        Cons(e, _) => 0;
        Cons(x, rest) => let (Int i) = index(rest, e) in
```

```
                            if i == -1 then -1 else i+1;
    };
```

The function definition of `index` returns the zero-based index of the element `e` of the list `l`, or -1 if `e` is not found. The second case branch uses the underscore pattern to signify that the rest of the list is unimportant if the current `Cons` contains `e`.

2.2 The imperative layer

On the imperative layer of ABS lie the language constructs that have side effects and allow imperative and object-oriented programming.

Imperative *statements* are allowed inside the methods of classes and the optional *main block* of the module. Statements may have side-effects, meaning that they alter the state of the program. Typical statements are declarations and assignment of variables, method calls, conditional statements (**`if ... else`**), and loops (**`while ...`**), all of which should be well known. The syntax of the imperative layer of ABS is more or less identical to the one of Java.

The object-oriented model of ABS is quite simple and minimalistic. The following briefly explain how classes and interfaces work in ABS.

2.2.1 Interfaces

Classes define the behavior of objects, but classes are not types in ABS; Objects can only be referenced as types defined by interfaces. Keeping an object in a variable therefore requires that the object's class implements at least one interface.

The interfaces can also define methods, which the classes then must implement to fulfil the requirement of the interface. The methods of a class is only visible from the outside if they are defined in the interface from which the object is referenced.

2.2.2 Classes

In addition to methods, a class can define *fields*, which are variables defined inside the class and define the state of the objects. There is a special type of fields called *class parameters* whose values can be set directly when the class is initialized. The fields of a class are not accessible from outside the object.

There is no inheritance in ABS, meaning that classes cannot extend other classes.

If a class implements a method named `run`, this method will automatically be started when the class is initialized. The class is then said to be *active*. The return type of the `run` method is `Unit`, which is a predefined algebraic data type used for methods without return values.

2.3 The distributed layer

The distributed layer describes the concurrency model of ABS which provides simple yet powerful mechanisms for parallel execution and synchronization.

2.3.1 Cooperative scheduling in concurrent object groups

Each object in ABS lives inside a *concurrent object group*, or COG. Within a COG, objects can communicate with normal synchronous method calls that are carried out sequentially. There are no threads or processes running in parallel inside the COG, only one process can run at a time.

The process currently running in a given COG will not be stopped or paused until it either terminates or *suspends* itself. Suspending is to temporarily give up the control and let other processes run for a while. This form of concurrency is known as cooperative scheduling.

There are two different statements that may suspend a process: **`suspend`** and **`await`**. The suspend statement suspends the process immediately, and once the control is given back to the process, it may resume. The await statement is conditional; the process will only be suspended if the given condition is not satisfied. If the await statement does suspend the process, the process may not resume until the condition fulfilled.

With the cooperative scheduling no object-level locking is needed to protect the internal state. Critical sections are protected by simply not suspending the process.

When classes are initialized with the keyword **`new`**, the new object is created in its own, new COG. To create an new object in the existing COG, **`new`** must be succeeded by the keyword **`local`**.

2.3.2 Communication by asynchronous method calls

To communicate with objects in other COGs than their own, objects can only send *asynchronous method calls*. These are different from normal method calls because the caller does not wait for the called method to terminate. An asynchronous method call is a sort of message which is sent between two COGs: The caller sends the message and continues its own execution directly after the call.

An asynchronous method call is syntactically identical to a normal method call, except for that an exclamation mark is used between the object reference and the method name.

When an object receives an asynchronous call to one of its methods, a separate process is created on the COG to execute the method. But, because of the cooperative scheduling, the method can not be executed at once if there is another processes running.

Asynchronous method calls can also return values. The return type of the actual *statement* that triggers the asynchronous method call, is a *future* with a type parameter that matches the return type of the function that is called. The future looks like a regular parametric data type, but it has special usage.

With a future variable, the return value of an synchronous method call can be retrieved later, when the method has returned from the other COG. The current process can be suspended until the method has returned, by using the future as a guard for the await statement. To retrieve the value immediately, the **`get`** statement can be used.

The following code listing is a snippet from the implementation of the *hygienic solution to the Diners Problem* in Chapter 3. The listing shows examples of asynchronous method calls, waiting for their return, and retrieving the return value.

The method called `request` is called on two philosopher objects. Because the `request` method has `Fork` as return type, the asynchronous method call statement returns a `Fut<Fork>` which means a future `Fork`. The await statement suspends the process until both the futures have been *resolved*, and the get statement is then used to retrieve the values.

```
Fut<Fork> reqL = philL!request(forkID(forkL));
Fut<Fork> reqR = philR!request(forkID(forkR));
await reqL? & reqR? ;
forkL = reqL.get;
forkR = reqR.get;
```

Chapter 3

The Dining Philosophers Problem

This chapter gives an introduction to the Dining Philosophers Problem and shows how it can be implemented in ABS. The problems of deadlock and starvation are explained, and it is shown how they appear in the Dining Philosophers. Finally, some solutions to the problem are discussed, and Chandy and Misra's *hygienic solution* is explained and implemented.

3.1 Problem outline

The Dining Philosophers is a well-known problem in the world of concurrent programming, formulated by Edsger W. Dijkstra in 1965. Originally titled *The problem of The Dining Quintuple*, it was first used as an examination problem in a course Dijkstra was teaching [9].

Serving as a metaphor on computers running in a network with common resources such as peripherals that require exclusive access, the dining philosophers problem efficiently demonstrates how a deadlock, or "deadly embrace", may occur.

The scenario involves 5 philosophers who sit around a table. All they do is alternate between thinking and eating. Between each pair of adjacent philosophers there is one fork on the table. For some reason the philosophers need two forks in order to eat, and they can only use the forks that lie next to them on the table.

At most one philosopher can hold the same fork at a given point of time, so clearly, since a philosopher shares forks with the ones on either side of him, two philosophers sitting next to each other can not eat simultaneously [8].

Dijkstra provides the following pseudo-code outline of a naive implementation of the philosophers, using a semaphore for each of the forks as a way of picking them up. After acquiring the semaphores that correspond to his forks, a philosopher may eat without interruption until he releases the semaphores again.

```
cycle begin think;
            P(left hand fork);
            P(right hand fork);
            eat;
            V(left hand fork);
            V(right hand fork);
      end
```

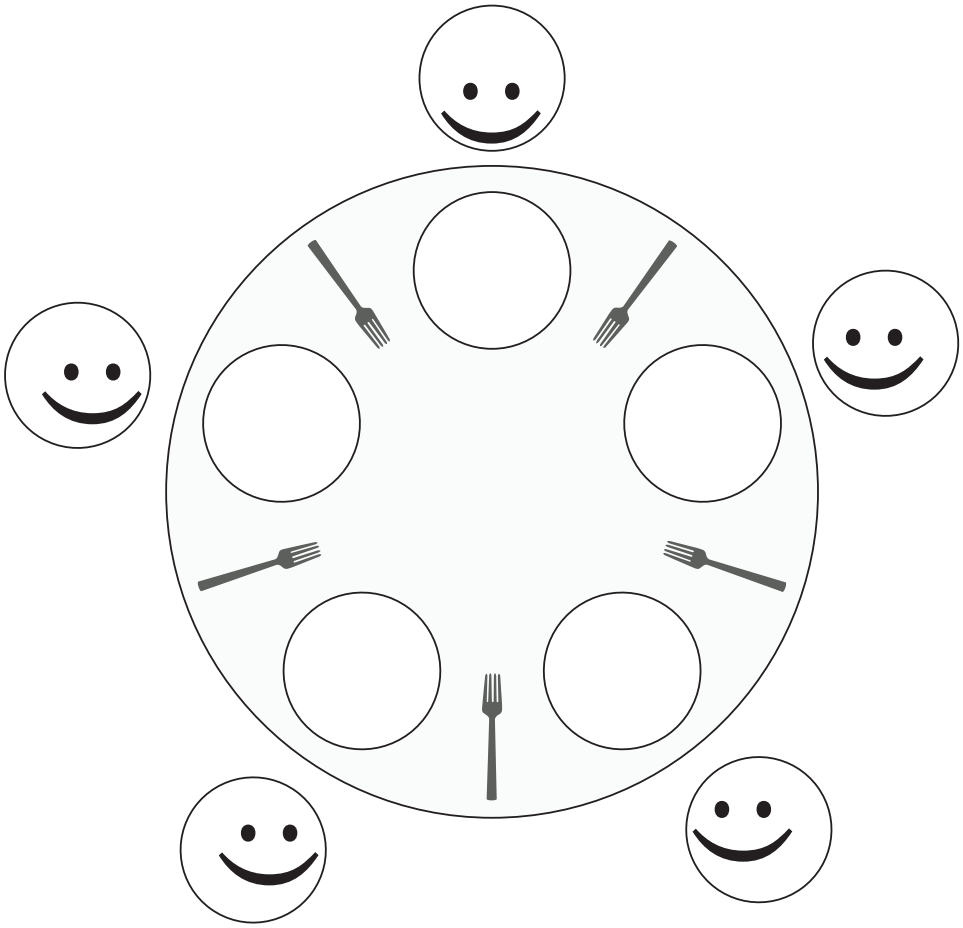


Figure 3.1: Five Dining Philosophers seated around a table with five forks. Each fork is shared between two adjacent philosophers.

3.2 Naive ABS implementation

To model the dining philosophers in ABS, semaphores cannot be used, as the language does not offer them. Instead, the object-oriented nature of the language may be utilized by letting the forks be objects, offering their own methods for obtaining and releasing exclusive access. The implementation will accordingly need two classes: `Philosopher` and `Fork`, which will also have their own interfaces.

The philosopher objects will simply run an infinite loop grabbing and releasing forks. Therefore they don't need any visible methods, and the `Philosopher` interface may be left empty. For the forks, two methods are specified, corresponding to the two semaphore operations, `P` and `V`. In typical object-oriented fashion, these methods are given more informative names, respectively `grab` and `release`. When calling either of these methods, a philosopher will need to identify itself by passing itself as a parameter.

```
interface Philosopher { }

interface Fork {
    Unit grab(Philosopher owner);
    Unit release(Philosopher owner);
}
```

The basic structure of the philosopher class is implemented as follows. The two fork objects accessible to a philosopher are passed as class parameters. All the logic is put in the `run` method, making the philosopher class active. Each time after calling `grab()` on a fork, the **`get`** statement is used without a preceding **`await`** statement, thus blocking the COG and effectively making the calls synchronous.

Listing 3.1: Basic implementation of a dining philosopher

```
1  class Philosopher(Fork left, Fork right)
2  implements Philosopher {
3
4      Unit run() {
5          while (True) {
6              // think
7
8              Fut<Unit> f;
9              f = left!grab(this);
10             f.get;
11             f = right!grab(this);
12             f.get;
13             // eat
14
15             left!release(this);
16             right!release(this);
17 } } }
```

It is essential that at any given time, at most one philosopher can hold the same fork. It makes sense to leave it to the implementation of the forks to ensure that this invariant is enforced. Conveniently, the ABS language's cooperative scheduling guarantees that each COG only will have one task running at a time. Therefore, critical sections of the program are guaranteed by the language semantics.

What must be enforced however, is that only the last philosopher who has successfully called `grab()` will be given access to other methods on the fork object, even after the call to `grab` has terminated. A field `owner` is sufficient; The owner is set with the `grab` method, and reset with `release()`. When trying to grab a fork, the philosophers must wait until `owner` has been reset.

If the value of `owner` was not really checked, other than whether or not its value is **`null`**, this would have the same effect as a binary semaphore. The field `owner` could then have been replaced by a boolean value indicating whether or not the fork was currently being held by a philosopher. However, `owner` arguably states the purpose of the variable more clearly. It also makes it possible to check that the philosopher who is releasing a fork is in fact the current owner.

Listing 3.2: The fork class

```
class Fork implements Fork {
    Philosopher owner;

    Unit grab(Philosopher p) {
        await owner == null;
        owner = p;
    }

    Unit release(Philosopher p) {
        if (p == owner)
            owner = null;
    }
}
```

In order to start the simulation, five `forks` and five `philosophers` are initialized in the program's main block. All of these objects need to run in parallel and independently of each other, so each object is created within its own COG.

```
{   // Main block:
    Fork f1 = new Fork();
    Fork f2 = new Fork();
    Fork f3 = new Fork();
    Fork f4 = new Fork();
    Fork f5 = new Fork();
    new Philosopher(f1, f2);
    new Philosopher(f2, f3);
    new Philosopher(f3, f4);
    new Philosopher(f4, f5);
    new Philosopher(f5, f1);
}
```

Running the basic implementation of the dining philosophers from the command-line is not very rewarding; The program merely runs 5 simultaneous, infinite loops, but not much appears to happen. The philosophers' main loop in Listing 3.1 contains two comments, on lines 6 and 13. These comments are placeholders for some imaginary `think` and `eat` routines respectively.

To show what is actually happening while the program is running, the `run` method can be augmented with some code that prints status messages to standard output. To tell the output from the different philosophers apart, each message should be prefixed with a string identifying the philosopher. For this purpose the built-in function `toString()` can be used with the philosopher object as parameter.

For code reuse, and to keep non-essential logic out of the `run` method, an auxiliary method `tell()` will be defined in the `Philosopher` class. The `tell` method will take a string and prefix it with the string representation of the philosopher object itself, before passing it to the built-in function `println()`.

Listing 3.3: A simple method for printing status messages

```
Unit tell(String s) {
    Unit p = println(toString(this)+" "+s);
}
```

With the `tell` method in place, the placeholder comments can now be replaced with calls to `tell()` with appropriate messages such as `"is thinking"`, and `"is eathing"`. To show more details of the philosopher's progress, status messages can also be printed after each time the philosopher grabs a fork and after he releases both forks. The output from a single, isolated philosopher process would look like the following, repeated over and over:

```
Philosopher 1 is thinking
Philosopher 1 grabbed left fork
Philosopher 1 grabbed right fork
Philosopher 1 is eating
Philosopher 1 released both forks
```

The full ABS implementation of the Dining Philosophers is listed on on page 105 in the appendix.

3.3 Deadlock

Because all the philosophers grab their respective forks in the same order and one at a time, a deadlock is likely to happen. As Dijkstra [8, p.15] explains, all philosophers may get hungry and try to grab their left-hand forks simultaneously. If they all operate roughly in the same speed, so that each philosopher successfully acquire his left-hand fork, then every philosopher will be waiting for his right-hand fork to be available. They are thereby all stuck in a cycle.

This can easily be demonstrated with the naive ABS implementation. No philosophers are programmed to give up a fork before they have picked up both of them, so they are all stuck waiting for each other. Running the full ABS program will – eventually – show that no philosophers get past the *thinking* state

and the program stalls. The following is the output from a sample run of the ABS implementation, with status messages, where the system enters a deadlock.

```
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is thinking
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 5 grabbed left fork
Philosopher 2 grabbed left fork
Philosopher 4 grabbed left fork
Philosopher 3 grabbed left fork
Philosopher 1 grabbed left fork
```

3.4 Starvation

As the philosophers run in a loop, they go into the thinking state directly after putting down the forks. If one philosopher is waiting for another to put down his fork, he is also depending on the scheduler to give him processing time at the right moment. The philosopher currently holding said fork could be a "fast thinker", meaning that he finishes thinking and has time to pick up the forks again within the time given to him by the scheduler.

Even in the ABS implementation there is no guarantee that waiting tasks will be given access at the right time. Below is outlined a course of events, showing two philosophers `A` and `B` competing for a fork `F`.

```
Philosopher A calls F!grab();
Philosopher B calls F!grab();      // B's task must wait
Philosopher A calls F!release();
Philosopher A calls F!grab();
```

Intuitively one could assume that after `A` releases the fork, `B`'s `grab` would always be granted before `A` manages to sneak in yet another `grab`. This does not need to be the case though; ABS will rely on the scheduling of the running backend, and philosopher `B` could be left starving.

This type of starvation is arguably more of a problem in a setting where the philosophers are local processes running on a single computer, all being handled by the same scheduler; and the forks represent some sort of local resource which require exclusive access, such as a `volatile` variable or a `synchronized` code block in Java. In such a setting, the act of grabbing a fork would be entirely performed by the philosophers, while the forks are not actually active processes that *do* anything.

In the ABS model however, the forks are active objects, each running in a separate COG. Logically these are to be regarded as entities independent of the rest of the system, and they actively react to the incoming requests. It is likely that a separate system handling remote procedure calls – which is what the COGs represent – has a scheduler, or internal queuing system, that would try to handle incoming messages roughly in the same order as they arrived.

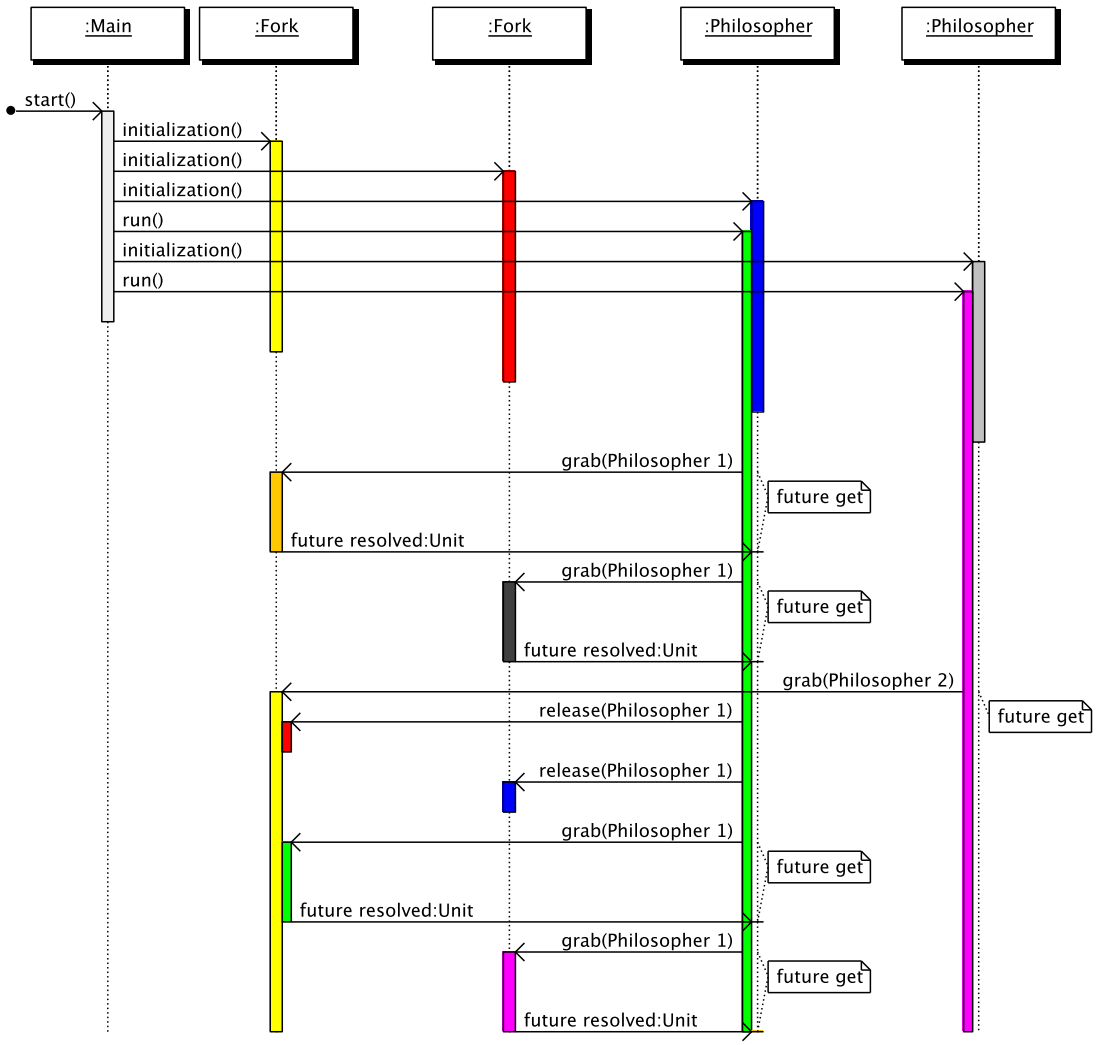


Figure 3.2: Sequence diagram illustrating the starvation of a process

Still, the basic principle of starvation can be demonstrated with the ABS model. Figure 3.2 shows a sequence diagram which is generated from an altered version of the implementation of the Dining Philosophers. For clarity only two philosophers were initialized in the main block, both with the same forks as arguments, and in the same order. The leftmost fork in the diagram is the one that the philosophers are competing for, corresponding to fork `A` in the example that was outlined in the example on page 16.

The sequence diagram shows the first philosopher successfully grabbing both the forks. This can be told from the arrows leading back from the fork processes, and the labels telling that the `future` has been resolved. This means that the `grab` tasks have terminated and returned to the philosopher. When a `future` has been resolved, the **`get`** statement in the `Philosopher` class no longer block the COG, and the philosopher can proceed.

After the first philosopher has acquired the forks, the second philosopher issues

a `grab` request to the first fork. This request cannot return at this time because the first philosopher already has grabbed it. But after Philosopher 1 releases both the forks, it immediately sends out new `grab` requests. The request to the first fork then competes with the existing request from Philosopher 2. In this example Philosopher 1 is granted the forks *again*, and we see that Philosopher 2 is still unable to advance.

3.5 Solutions

3.5.1 Anti-symmetry

It has been covered that when the deadlock appears, all the philosophers are waiting in a cycle; They are all programmed the same way and they are all in the same state (waiting for fork number two). Lehmann and Rabin [18] refer to this as *symmetry*.

They argue that when the initial configuration is symmetric, the scheduler may choose to advance each philosopher one step at the time in a circular order. After the scheduler has finished one round, the configuration is again symmetric. This is part of their proof to the following theorem [18, p.135]:

> There is no deterministic, deadlock-free, truly distributed and symmetric solution to the dinings philosophers problem.

Andrews [1, p.166] suggests simply letting one philosopher pick up the right-hand fork first. This breaks the symmetry and eliminates the possibility of circular waiting, as the right-handed philosopher will have to compete with his right-hand side neighbor for the *first* fork. One of these two philosophers will have to wait, and while that philosopher is waiting he will not pick up his *second* fork, which in the meantime will be available for the opposite neighbor.

Instead of a separate implementation, letting one philosopher pick up right fork first is easily achieved in the ABS model by flipping the order of fork arguments as the last philosopher is created. The program could then potentially run forever without stalling. There is however no guarantee against starvation; Each philosopher is still competing with its neighbors for access to the forks.

3.5.2 Servant process

Another solution, which at first glance may seem like the most obvious one, is to declare that the philosophers must grab both the forks at the same time, instead of one by one. This would prevent the philosophers to get stuck with one fork in the left hand, waiting for the right-hand fork to become available.

In the current setup, however, the philosophers have no means to grab two forks in one atomic operation. The forks are running is separate COGs, independent of each other. As the COGs communicate through *asynchronous* message calls, clearly an atomic operation like that would involve some sort of synchronization.

The simplest form of synchronization is arguably to have the forks handled by another process – a servant – which will grant a philosopher access to both his forks simultaneously. The philosophers would then communicate only with this servant process, rather than with the forks directly.

In order to prevent that the servant faces the exact same problems as the philosophers do on their own, the system should have only one servant which handles all the forks. As its task is to make sure that two forks are handed to a philosopher simultaneously, it must leave a philosopher waiting if one of the forks the philosophers is requesting is not currently available.

The servant solution prevents deadlocks in the system, at the cost of being centralized. All the communication goes through the same servant, as opposed to the original scheme where the communication was spread evenly among the different philosophers and forks.

3.6 Chandy and Misra's "hygienic solution"

A more complete and fully decentralized solution to the dining philosophers, that also fits nicely with the cooperative scheduling of the ABS language, is *A hygienic solution to the diners problem* by Chandy and Misra [4]. This section gives a brief explanation of the solution and walks the reader through an ABS implementation.

3.6.1 Solution overview

Chandy and Misra's solution differs from the original problem by Dijkstra in that no fork ever lies on the table, but is always being held by some philosopher. When a philosopher wants a fork that he is not currently holding, he can therefore not simply pick it up – He must instead send a *request* to the current holder of that fork.

To ensure that the forks are not simply sent back and forth between two philosophers, so that none of them have time to use it, the forks will be marked as `clean` or `dirty`: When a philosopher hands over a fork to a neighbor, he will clean it first. After a philosopher has finished eating, the forks he is holding will be marked as dirty. The key point is that a philosopher will only grant a request for a fork if the fork is dirty (and he is not currently eating). This guarantees that a philosopher will always get to eat at least once before giving up his forks, so the problem of starvation is eliminated.

The configuration is initially set up so that one of the philosophers is holding two forks while his right-hand neighbor is holding none. The rest are holding one fork each in their left hand. For each fork shared between two philosophers, the philosopher not currently holding the fork will instead hold a *request token* for that fork.

A philosopher sends a request token to another philosopher to indicate that he wishes to use the fork and that he wants the other philosopher to send the fork to him. When one philosopher is holding both a fork and the request token for that fork, then the other philosopher has an outstanding request for that fork [4, p.638].

3.6.2 ABS implementation overview

Two of the first problems that have to be considered when implementing the Chandy/Misra solution is how to realize the forks and request tokens, and how these should be sent between philosophers.

The latter almost solves itself; The only way two ABS objects may exchange information across COGs is by asynchronous method calls. The most obvious way to realize the sending of request tokens and forks is therefore to have a request token sent by an asynchronous method call that – while the philosopher holding it is not eating – returns the fork upon termination.

An outstanding request for a fork will then occur in the ABS model as a method call that has not yet terminated. This means that one philosopher will not be holding both a fork and its request token at the same time, which is something that occurs in Chandy and Misra's original solution.

Still, two neighboring philosophers will have different relationships to the fork they share between them: One philosopher is holding the fork while the other is holding a request token for the fork. This prompts the philosophers to keep two pieces of data for each fork; They must know whether they are currently holding the fork in question, and they must remember the fork's ID so that they can request the correct fork from their neighbor. This problem is minimized by a few simplifications in the model.

3.6.3 Forks as an algebraic data type

In this implementation, the forks will be realized as an algebraic data type. The main reason for this is that the philosophers will not actually *use* the forks for anything, except as tokens indicating that they may start their `eat` routine. Neither is there any logic that needs to be placed in the forks – the philosophers are the ones handling everything.

All that the philosophers really need to know is whether they are currently holding both forks so they can start to start to eat, and whether a fork is clean or dirty when they receive a request for it. This information will be stored in the forks. The forks are thereby nothing more than containers of data, which is best represented in ABS as algebraic data types.

It has been covered that one philosopher will not be holding both a fork and its request token simultaneously. Moreover, the actual request for a fork will be done through a asynchronous call to a method defined specifically for this purpose. The only real use left for the request token is to indicate that the philosopher is *not* holding a fork, analogue to a **`null`** pointer for objects.

The request token is then, from a single philosopher's point of view, merely another state of the fork, along with `clean` and `dirty`. These three values can thereby be merged into a single data type, which will be called `FState`.

```
data FState = Clean | Dirty | RequestToken;
```

With this simplified representation, and because algebraic data types are immutable, there is no coupling between a request token and one unique fork object. There must be a way to tell the forks apart, and to identify them when sending a fork request. This can be achieved by giving the forks an identifying integer argument. For clarity a `ForkID` type synonym is declared.

```
type ForkID = Int;
data Fork = Fork(ForkID forkID, FState state);
```

With the algebraic data type `Fork`, philosopher processes can now easily pass information about the forks among each other. The immutability of the three different states, as well as the fork IDs, make reasoning simple and clear. If for instance a philosopher is holding fork 4, and fork 4 is clean, he will store the information about this fork as the expression `Fork(4,Clean)`. The neighbor he is sharing fork 4 with will at the same time know it as `Fork(4,RequestToken)`.

3.6.4 `Philosopher` class overview

Unlike in the naive solution, the philosophers can not start running right upon their creation. They all need to know about their neighbors so that they can pass request tokens and forks to each other. As the neighbor relationship is circular, all the philosophers must be initialized before each is given references to his neighbors. A method `start` will be used to pass information about a philosopher's neighbors and indicate that the main routine may begin.

As has been covered, these requests will be done through asynchronous method calls. The method `request` takes a `ForkId` argument and returns a `Fork`.

```
interface Philosopher {
    Unit start(Philosopher left, Philosopher right);
    Fork request(ForkID i);
}
```

The forks, on the other hand, can be passed directly to the philosophers as the philosophers are created, just like in the implementation of the naive solution Either way, there is no *initialization* of an algebraic data type. For brevity the left and right forks are simply called `forkL` and `forkR` respectively, and the fields holding the philosopher's neighbors are similarly called `philL` and `philR`.

Listing 3.4: Beginning of the Chandy/Misra Philosopher class

```
class Philosopher(Fork forkL, Fork forkR)
implements Philosopher {
    Philosopher philL;
    Philosopher philR;

    Unit start(Philosopher left, Philosopher right) {
        philL = left;
        philR = right;
```

Once its main routine begins, a philosopher will cycle trough the states *thinking*, *hungry* (waiting for one or two forks) and *eating*. Except for when in the eating state, a philosopher should be able to handle incoming fork requests. But, with the cooperative scheduling in ABS, a fork request will not run in parallel with the philosopher's thinking routine. Instead, explicit scheduling points must be inserted at convenient locations in the code.

3.6.5 Scheduling points

If the philosopher was a part of an actual, distributed system, and its thinking routine would run for a long time, it would probably be in the philosopher's own interest, as well as the system as a whole, that the philosopher would check for incoming fork requests as often as the thinking routine would allow. The reasoning is that the sooner a philosopher is given a fork, the sooner it will be done with it and thus ready to give it back.

If for instance two philosophers *A* and *B* would only handle fork requests at the very end of their thinking routines, then neither one of *A* and *B* could eat while the other was thinking; They both would have had to pass the forks while they were hungry and actually wanted the forks for themselves. Optimally two neighbors would pass the fork between them so that one was always eating while the other was thinking. In practice this would probably seldom be the case. Besides, the Dining Philosophers problem involves five processes, and no two neighbors can eat at the same time. Therefore at most two philosophers can eat simultaneously and there is at least one pair of neighbors where neither is eating.

What the philosophers are in fact doing during the eating and thinking routines is irrelevant to this model. But to make it appear as if *something* is happening, each state is realized by a corresponding method. The transition from one state to another takes place when the method corresponding to the first state calls the method corresponding to the second state.

In addition to `thinking` and `eating` there will be a state called `hungry` to clarify that philosopher is done thinking and will try to acquire the forks. The philosopher is then completely oblivious to the forks during the thinking state.

The calls to these state methods are done asynchronously[1]. Each transition to a new state is therefore manifested by a new task in the COG before the task corresponding to the old state terminates. This automatically generates a scheduling point between each state.

Another benefit is that the transitions between states will become apparent in sequence diagrams, so that the diagrams give a more detailed picture of the model.

3.6.6 The `thinking` and `eating` states

An ABS task is unaware of the other tasks running on the same COG. Therefore, an incoming fork request cannot know the state of a philosopher without this being stored in a field on the `Philosopher` object. A simple algebraic data type `Pstate` is defined, enumerating the states `Thinking`, `Hungry` and `Eating`, and the philosophers are given a field of this type, called `state`. This field is updated to the next state at the end of the method corresponding to the *previous* state.

The states *thinking* and *eating* are quite simple. Besides changing state, the method `think` does not do anything. After a philosopher is done eating, his forks should be marked as `dirty`, and the appropriate place to do this, is the `eat` method.

[1] An infinite loop of recursive, synchronous method calls, would sooner or later cause the Java backend to to crash with a *stack overflow*.

Listing 3.5: The thinking and eating states of a hygienic philosopher

```
Unit think() {
    state = Hungry;
    this!hungry();
}

Unit eat() {
    forkL = Fork(forkID(forkL), Dirty);
    forkR = Fork(forkID(forkR), Dirty);
    state = Thinking;
    this!think();
}
```

3.6.7 The `hungry` state

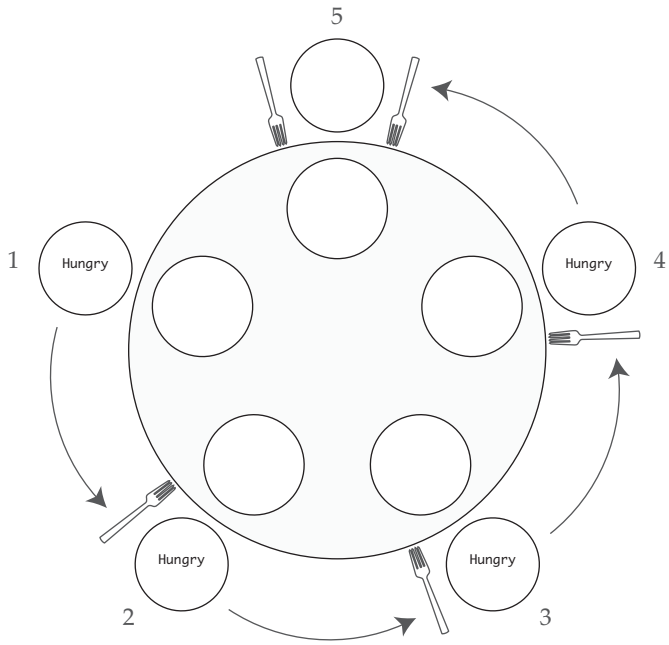
The `hungry` state is where a philosopher will try to acquire the forks if he is not already holding them. Branches in the code are needed to cover each of the following possibilities, and in each case, requests must be sent to the appropriate neighbor(s): The philosopher may be missing *both* forks, only the *left* fork, or only the *right* fork. If none of these conditions are true, then the philosopher is already holding both the forks.

What is important to note, however, is that while the philosopher is waiting for one neighbor to pass a fork, the philosopher may himself receive a request from the opposite neighbor to pass the other fork. While it may seem inefficient, it is actually crucial for the algorithm that a philosopher will give up the fork he is holding while he is waiting for the other, unless, of course, the fork he is holding is clean.

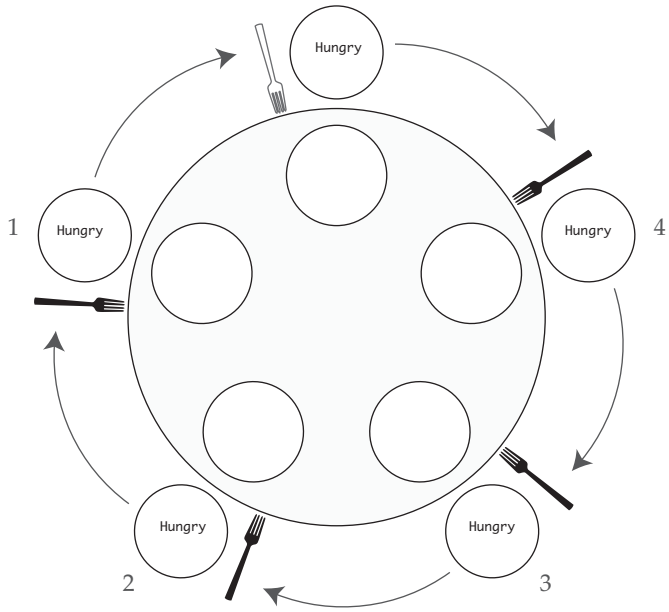
If the philosophers refuse to give up their forks while they are waiting for their own fork request to resolve, a deadlock similar to the one in the original Dining Philosophers problem may occur. A sequence of events leading up to such a situation is illustrated with Figure 3.3 on the next page.

The topmost part of the figure shows the philosophers in the initial configuration. They are getting hungry one by one, starting to send fork requests to their neighbors. The first philosopher to get hungry is Philosopher 1, the one that initially did not hold any forks. He sends the fork requests to his neighbors, philosophers 2 and 5. The first request reaches Philosopher 2, which sends over the fork just before getting hungry himself. Philosopher 2 then sends a fork request back to Philosopher 1, and one request to Philosopher 3, which in turn repeats this pattern. Philosopher 4 follows, successfully acquiring one of the two forks that Philosopher 5 started with.

After this, the configuration of forks and philosophers is as shown on the bottom part of the figure. Philosophers 1 through 4 are holding a clean fork in their right hand, waiting for the left fork. The only dirty fork, and thus the only fork that may change hands at this point, is the one held by Philosopher 5. If Philosopher 5 gets hungry and sends a request for his left-hand fork before receiving the request from Philosopher 4, then all the philosophers are waiting



(a) The initial configuration. The diagram is augmented with arrows indicatig fork requests being sent after the philosophers are getting hungry. An arrow pointing from a philosopher to a fork indicates a request *from* the philosopher *for* that fork.



(b) The configuration after the fork requests in the previous subfigure have been resolved, and the philosophers are waiting in a cycle, as described on the previous page. Clean forks are shown in solid black to indicate that they currently cannot be taken from the philosopher holding them.

Figure 3.3: The initial configuration of Chandy and Misra's *hygienic solution to the diners problem* and how the philosophers can enter a cyclic configuration, exhibiting the need for a philosopher to give up a (dirty) fork while waiting for another.

in a cycle, similar to the deadlock situation explained in Section 3.3. From this it is clear that the philosophers need to accept incoming fork requests while they themselves are waiting for a fork.

This is easily solved by using the await statement. Unlike in the implementation of the naive solution in Listing 3.1, the remote procedure calls will be followed by await statements before using the **get** statement to obtain the return value[2]. The process corresponding to the philosopher's `hungry` state is thereby suspended until the fork requests are resolved, allowing other tasks to be executed in the meanwhile.

Because there will be repeatedly need for checking whether a philosopher needs to send a fork request a simple helper function `needsFork` is defined to shorten the code and make it more readable.

```
def Bool needsFork(Fork f) = state(f) == RequestToken;
```

To ensure that a philosopher does not try to eat with only one fork, the forks must be checked again after the `hungry` method resumes from awaiting a fork request. An extra while loop wrapped around the main part of the `hungry` logic will suffice.

Listing 3.6: The `hungry` state of a hygienic philosopher

```
1  Unit hungry() {
2      while (needsFork(forkL) || needsFork(forkR)) {
3
4          if (needsFork(forkL) && needsFork(forkR)) {
5              Fut<Fork> reqL = philL!request(forkID(forkL));
6              Fut<Fork> reqR = philR!request(forkID(forkR));
7              await reqL? & reqR?;
8              forkL = reqL.get;
9              forkR = reqR.get;
10
11         } else if (needsFork(forkL)) {
12             Fut<Fork> req = philL!request(forkID(forkL));
13             await req?;
14             forkL = req.get;
15
16         } else if(needsFork(forkR)) {
17             Fut<Fork> req = philR!request(forkID(forkR));
18             await req?;
19             forkR = req.get;
20     } }
21     state = Eating;
22     this!eat();
23 }
```

[2]When the philosopher only need to send one fork request, the RPC immediately followed by **await** and **get** could normally be replaced by the shorthand *await statement* which joins the three statements into one. This is not done here because the Eclipse plug-in, which is used for making the sequence diagrams, has not yet been updated to support this syntax.

3.6.8 The fork request

With the logic of the philosophers' states *thinking*, *hungry* and *eating* in place, the only remaining piece is the handling of fork requests. Most of the foundation of this implementation has been laid, so the `request` method is quite straight-forward.

The method takes a `ForkID` argument, which the philosopher will need to check against the IDs of the two forks he is holding. In a more general solution the philosophers might have access to a larger, possibly variable, number of forks. It would then need a better facility for looking through the set of forks, such as a map structure. For this model, an if-statement for each of the forks will do. For simplicity it is assumed that a philosopher will not receive requests for other forks than the two that he has access to.

The procedure of the fork request is the same for both forks, but different variables need to be accessed. Therefore the logic is duplicated in the two branches of the if-statement.

Having sorted out which of the two `Fork` fields the request is for, the method must wait for that fork to be `dirty` and for the philosopher to not be `eating`. These two conditions must be checked in one atomic operation; If the two conditions were instead checked in two individual steps, the process could pass the guards one by one, without both the conditions being true at the same time. The philosopher could then for instance end up giving away a fork right before it began eating.

To handle the request correctly and not break any invariants of the problem, a single **`await`** statement checks that the philosopher is not eating and that the fork is dirty. When the process has passed this guard, the field corresponding to the fork in question is changed to a request token, before a clean fork is returned to the caller of the fork request.

Listing 3.7: The fork request

```
Fork request(ForkID i) {
    if (i == forkID(forkL)) {
        await this.state != Eating && state(forkL) == Dirty;
        forkL = Fork(i, RequestToken);

    } else if (i == forkID(fr)) {
        await this.state != Eating && state(forkR) == Dirty;
        forkR = Fork(i, RequestToken);
    }
    return Fork(i, Clean);
}
```

3.6.9 Testing the model

The last adjustment to the model before initializing the philosophers, is to make sure that the philosophers enter the cycle of thinking, getting hungry, and eating. In the bottom of the `start` method in Listing 3.4 on page 21 an asynchronous call to `think()` must be added.

In the main block the five philosophers are created, named `p1` through `p5`. To match the initial configuration as it was shown in Figure 3.3a, each philosopher has

the fork with the number corresponding to his name, on his left-hand side. And for each pair of neighbors, the one with the highest number will be the one to hold the fork they share between them. Thus `p5` is the one initialized with two forks, and `p1` is initialized without any forks.

After all philosophers have been initialized, their `start` routines can be called. The order of the parameters to this method will determine which philosophers that are neighbors. To match with the mapping of the philosophers' names and the forks' IDs as defined above, the neighbor relationship is set up as follows.

```
// Abbbritated to make each statement fit on one line
Phil p1 = new Phil(Fork(1, Token), Fork(2, Token));
Phil p2 = new Phil(Fork(2, Token), Fork(3, Token));
Phil p3 = new Phil(Fork(3, Token), Fork(4, Token));
Phil p4 = new Phil(Fork(4, Token), Fork(5, Token));
Phil p5 = new Phil(Fork(5, Token), Fork(1, Token));

p1!start(p5, p2);
p2!start(p1, p3);
p3!start(p2, p4);
p4!start(p3, p5);
p5!start(p4, p1);
```

Deadlock

It was explained in Section 3.6.7 how the system could enter a configuration where all the philosophers were waiting in a cyclic arrangement. A deadlock similar to the one in the naive solution of the Dining Philosophers would occur if the philosophers would not be able to handle fork requests while they were in the *hungry* state waiting for another fork.

With the interactive scheduler in the Eclipse plug-in, such a situation could be reproduced. Figure 3.4 on page 29 shows that the `hungry` method works as intended, allowing concurrent fork requests to be handled. For clarity the two other philosopher not touched by the two fork requests were edited out of the figure.

The figure also shows the different states of the two leftmost philosophers, thanks to the states being their own processes, and it can be seen that the final fork request is not granted until after the philosopher is done eating.

It has been shown that a dirty fork can always change owner. Therefore any configuration with at least one dirty fork can always change into another configuration; There cannot be a deadlock in the system as long as there are dirty forks.

Figure 3.3b shows a configuration where all forks but one are clean. This configuration is reached from the initial configuration by having the philosophers 2 through 5 give up their left-hand forks. In order to reach a configuration where all forks are clean, Philosopher 5 must also give up his right-hand fork. No forks can then immediately change hands, but Philosopher 1 is now holding two forks, and may eat, so the configuration still advances. After Philosopher 1 has eaten, his forks are again dirty. Because Philosopher 5 is initially holding two forks, a

configuration where each philosopher is holding one clean fork cannot be reached. Deadlock is therefore impossible.

Fairness

Section 3.4 covered the principle of process starvation, and it was shown how the scheduler could choose

XXX Skip this?
This is a stricter requirements
TODO

- Scheduler seems to be weakly fair
  - Show that running the model on the Java backend tends to end in a situation similar to Figure 3.3, having one philosopher thinking and eating non-stop, the rest locked.
  - Show how to solve that with an additional “fairness guard”
  - Ref to Section 3.4 – without additional fairness, this model is actually a better example of starvation!
  - Ref [1, p.74]

<!– # draft/some ideas for the rest of the chapter

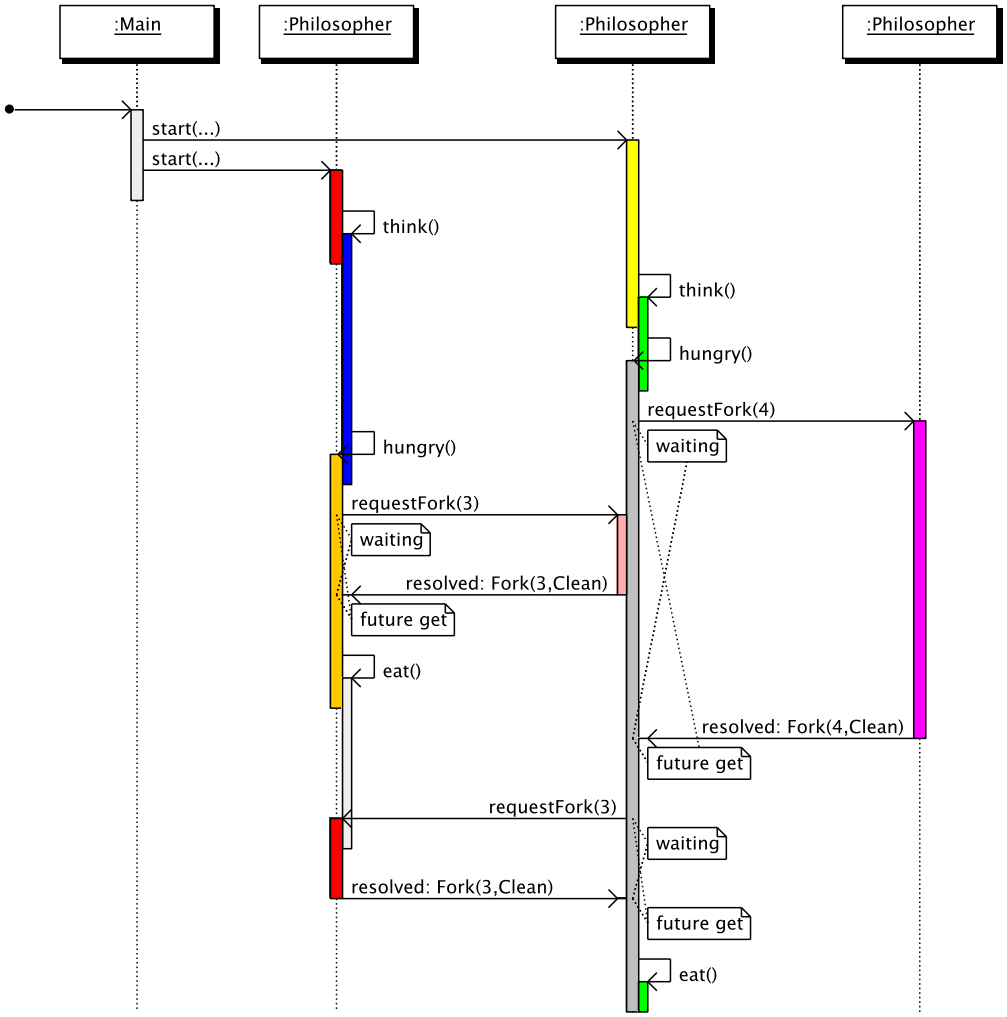


Figure 3.4: A simplified sequence diagram showing a hungry Chandy/Misra philosopher (in the middle) waiting to recive a fork from his right-hand neighbor. In the meantime the opposite neighbor sends a request for the other fork. In order to prevent a possible deadlock the philosopher gives that fork away, and must wait until the left-hand neighbor is done eating

Chapter 4

Paxos

Paxos is a fault-tolerant algorithm for reaching consensus among processes working together in a network. The algorithm was suggested by Leslie Lamport in 1990, but not published until 8 years later [13, 15]. Paxos is used in services like Apache ZooKeeper [3] and Google Chubby [2].

A distributed system running the Paxos algorithm will be modeled in ABS. Later another module is introduced that tries to mimic faulty communication in the implementation with an experimental approach to intercept messages between the concurrent object groups.

4.1 Algorithm overview

Given a network of processes that can propose values to each other, the purpose of the Paxos algorithm is to have the processes cooperating on picking one, and only one, of the proposed values. This value is said to be *chosen*.

What the actual value is, has no importance to the algorithm itself. Lamport [14, p.8] describes a distributed state machine like a banking system, as an example.

Having only a single central server may not prove to be robust enough for such a system – if the server fails, everything fails. But with more than one server, the risk of such failure is reduced. It is important however, that all these servers are up to date as often as possible, and that none of them contain false information.

With a deterministic state machine implementation, the state of the server can be exactly replicated on another server, given the correct sequence of commands [16]. The Paxos algorithm can be used to ensure that all the servers execute the same commands in the same order. In this case the state machine commands would be the values to be chosen by Paxos. To determine a sequence of multiple commands, the algorithm must be executed as many times as there are commands.

Safety requirements

The Paxos algorithm is based on asynchronous communication, with the following assumptions [14, p.2]: Messages are sent asynchronously, and can therefore be delayed, lost or duplicated, but it is assumed that they will not be corrupted. Even under these conditions the Paxos algorithm guarantees to preserve the *safety*

requirements that only one value can be chosen, and that a chosen value is one that has been proposed by an agent. Also, an agent will never learn a value that has not actually been chosen.

It is also assumed that an agent can fail by stopping and then restart. The agent must then be able to remember some information after it has restarted.

4.2 Agents

There are three types of agents involved in Paxos.

- The **proposers** are the ones that propose values.
- The **Acceptors** receive the proposals from the proposers and may or may not accept these proposals, depending on factors that will be covered later. It is important to note that a value is not necessarily *chosen* even though it has been *accepted* by one or more acceptor.
- The **Learners** will hear what values the acceptors have accepted, and will deduce from this what value has been *chosen*.

Each process may act as more than one type of agent. At the very least it is likely that all processes want to learn the chosen value, and would therefore act as a learner.

4.3 Quorum of majority

The Paxos algorithm takes advantages of a simple principle regarding majorities of a set. It will be shown later in this chapter that not all acceptors necessarily accept the same values. So to learn what value has been chosen, an agent cannot trust only one acceptor. However, it is not necessary to have the value confirmed by *all* the acceptors, only a majority, i.e. more than half, of the acceptors is enough.

This is safe because, given a set of agents, any two subsets that consist of more than half of those agents must have at least one agent in common. Given that an agent cannot have accepted more than one value, there cannot exist two majorities that have chosen different values.

This reduces some of the overhead that comes along having multiple servers in a system instead of just one, with the benefit of improved robustness; A system could tolerate the failure of F processors if there were $2F$ 1 processors in total [12].

The next section describes the two phases that the proposers and acceptors go through before any value is accepted, and subsequently chosen. It will be shown that the proposers also rely on the majority of acceptors before they advance to the second phase.

4.4 Reaching consensus

Before a value can be chosen it obviously needs to be proposed, so the proposers are the first ones to take action in the Paxos algorithm. They will not start

proposing values right away though; The interaction between proposers and acceptors happens in two phases before a value is chosen.

In the first phase a proposer tries to have the acceptors promise to accept a proposal that it will send later. In the second phase it will determine whether it is allowed to issue a proposal, before potentially doing so. The two phases will be referred to as the *prepare phase* and the *accept phase* respectively. The following describes the two phases more in detail.

4.4.1 The prepare phase

Before issuing an actual proposal, a proposer will send a *prepare request* to a majority of the acceptors. This request contains a *proposal number n* which will be used to determine which proposer may eventually issue a proposal. In a way the proposers are going through as sort of "bidding round" before the actual proposals take place.

The proposers may send more than one prepare request, but must use different proposal numbers each time. Also, two different proposers must never use the same numbers. To prevent different proposers from using the same proposal number, the proposers must somehow agree in advance on the set of numbers that each proposer is allowed to use. How a proposer chooses a proposal number in practice will be covered during the ABS implementation of the algorithm.

Each acceptor keeps track of the highest proposal number it has received. If the proposal number n of an incoming prepare request is the highest so far, or if it is the first prepare request it has received, an acceptor should send back a response. This response informs the proposer of the current state of the acceptor.

It is important to note that while a proposer is still in the prepare phase, an acceptor may have been communicating with other proposers, which may have completed both the phases. When receiving a prepare request, an acceptor may therefore already have accepted a value from another proposer.

When an acceptor responds to a prepare request, it tells the proposer about the highest proposal number it has received (which should be one that the proposer sent), and whether is has already accepted a value. If it has accepted a value, it will include the proposal that it has accepted, in the response that it sends to the proposer.

The response to the prepare request is also a promise to the proposer to not accept any other proposals numbered less than n.

If the proposal number is lower than the highest it has received, an acceptor may simply ignore that prepare request.

4.4.2 The accept phase

If a proposer receives responses from more than half of the total number of acceptors, it is ready to send the accept request. This is the actual proposal. However, it must first decide what value to propose. To do so it must take into consideration the responses it got from the acceptors.

If some of the acceptors reported back that they had already accepted a proposal, the proposer must propose the value of the highest-numbered proposal

among these. If no accepted proposal was reported, the proposer is free to use its own value.

4.4.3 Progress

When a proposer receives a response to a prepare request, the acceptor responding has promised the proposer to ignore any future requests numbered lower than a given proposal number n. After receiving enough responses the proposer may send the accept requests, but it is entirely possible that another proposer in the meanwhile has sent prepare requests with a number greater than n. The accept request from the first proposer is then ignored by the acceptors – the request can be said to have been *canceled out* be the other proposer.

Moreover, a proposer can pick a new, greater proposal number and start the prepare phase again. If the requests of proposer A has been canceled out by proposer B, then A may start over and, it turn, cancel out B's requests. This may continue back and forth, both proposers canceling out each other's requests. Neither proposer will then be able to complete the accept phase, and no value will be chosen.

Two proposers that keep canceling out each other's requests will be said to be *racing* against each other.

Paxos itself does not have a mechanism for sorting out a race between two proposers: The algorithm guarantees that only one value can be chosen – However, there is no general guarantee that a value *will* be chosen.

To guarantee progress, Lamport [14, p.7.] suggests selecting a *distinguished proposer* and let it be the only one allowed to issue proposals. Other agents with values to propose will then suggest these to the distinguished proposer rather than propose the values themselves. The distinguished proposer can be determined with any leader election algorithm.

4.5 Modeling Paxos in ABS

The ABS implementation will focus entirely on the basic consensus algorithm of Paxos. Concerns such as electing a distinguished proposer are considered irrelevant and not implemented. Instead, the model will focus on preserving the safety properties, which is the purpose of Paxos, and it will be assumed that progress is always possible.

The Paxos model will be carried out in two steps. First a module `Paxos` is presented, implementing the different agents. This module does not take faulty communication into consideration, but provides agents that will behave correctly regardless. After the implementation of `Paxos` another module will be defined which expands the previous and introduces faulty communication.

Running the algorithm

The module `Paxos` will contain all the logic needed to run an instance of the algorithm: data types for values and proposals, interfaces for all the types of agents, and classes implementing these interfaces.

Any configuration of agents and values can be set up. The algorithm is run by instantiating agents through the classes provided, and giving the agents the necessary information. When the proposers are initialized, the algorithm starts.

The implemented agents will only run one instance of the algorithm, so that it can be properly examined and tested. Running another round of Paxos will require to initialize the agents again.

Only having to run the algorithm once also helps simplifying the model, in that the agents will have fewer internal states. There will not be needed any logic to keep track of each round, nor to synchronize the agents between the rounds.

Disregarding distinguished proposers

In an ideal situation without faulty communication, a leader election algorithm would successfully elect a distinguished proposer, and no other agents would try to propose a value. Hence the prepare and accept requests from the distinguished proposer would not have been canceled out by any other proposer. As long as the distinguished proposer does not fail, and enough messages get through, there are no obstacles to stop the distinguished proposer from successfully proposing a value.

This scenario is arguably not very interesting, and the Paxos algorithm would not even be needed if such ideal circumstances could be guaranteed. It is when the proposer fails that Paxos become crucial for preserving the safety:

The distinguished proposer could experience message loss to the extent that it effectively was isolated from the rest of the network. To the other agents it would appear as if the distinguished proposer had failed. Without the distinguished proposer's knowledge, the other agents could start a new round of leader election to find a new distinguished proposer. If the communication was restored between the old distinguished proposer and the other agents, there would be two proposers simultaneously believing themselves to the rightfully distinguished one. This shows that with faulty communication, using a distinguished proposer is alone no guarantee for safety.

Because a distinguished proposer makes no difference towards the safety requirements of the system, the model will not include a leader election algorithm. Instead it is assumed that a distinguished proposer can eventually be determined if needed, so that progress is guaranteed.

Simplified time model

Without faulty communication in the model, the proposers in the basic implementation can assume that messages will arrive in time. The model can be simplified if the proposers do not need to consider time-outs. They will rather be implemented only to react upon receiving the acceptors' responses. If a proposer does not receive enough responses to enter the accept phase, it will not restart the prepare phase by itself.

A proposer that is waiting for a respond cannot know *why* the message has not appeared. It may be caused by faulty communication, the acceptor failing, or another proposer canceling out the request. Modeling proposers that time out is best left to the implementation of faulty communication.

Having the basic proposers only send one batch of prepare requests makes the model a lot more predictable. It also ensures that a run of the model will terminate, even without a distinguished proposer.

Racing proposers

Given two proposers that have picked their respective proposal numbers and are in the beginning of their prepare phases: In order to complete both the phases of the algorithm without restarting, the one proposer that has the smallest proposal number of the two would need its *accept* request to reach the acceptors before the other proposer's *prepare* request. Granted, if the two proposers are operating at the same speed, the one with the greatest proposal number does have an advantage. Still, either one of the proposers could complete the accept phase.

Assuming that there will be progress in the model means to assume that, even if two proposers are racing each other, still one of the two proposers will eventually complete the accept phase by successfully issuing enough accept requests before the other proposer's next prepare requests.

When two proposers A and B are racing against each other, they will keep picking new proposal numbers. In order to cancel out A's requests, B must pick a new proposal number greater than the previous number used by A, and vice versa. As long as the proposers correctly picks numbers from non-overlapping pools and the race is eventually terminated, one proposer will have a proposal number greater than and not equal to the other's.

Having the proposers pick one proposal number and stick with it, is a way of predetermining which proposer that comes out of the race with the highest proposal number; The model skips directly to the end of the race. It will be shown that with two proposers starting the prepare phase one time each, finally a value will be chosen.

4.5.1 Data types

All the data types, interfaces and class definitions in this module will be needed to create an actual running instance of the algorithm or to expand the model. Therefore everything is exported so it can be imported in another module.

```
module Paxos;
export *;
```

The actual values that will be chosen by the Paxos algorithm has no real purpose in this model, besides being proposed and chosen. But because the module is made to support any configuration of agents, it must be possible to produce arbitrarily many different values, so that the values can be identified and traced along the run of the algorithm.

With infinitely many possible values, the integer is a fitting data type to represent a value. An algebraic data type `Value` is defined as a wrapper for this integer, to make it stand out from the rest of the numbers that is sent to and from the agents. This will be useful when debugging the model and when reading sequence diagrams.

The information stored in the different agents will mostly be various integers. A type synonym `Pnum` for proposal numbers makes the purpose of certain fields and the parameters in method signatures more self-explanatory.

In this implementation it will be made sure that 0 is never used as a proposal number in an actual proposal; The value 0 will be reserved as the default value indicating that an acceptor has not received any prepare requests. This makes certain reasoning simpler.

An algebraic data type is defined for the proposals. The `Proposal` type will have two constructors: The first, simply called `Proposal`, pairs a proposal number with a value; The second constructor, `None`, is a default value used for indicating that no proposal has been accepted or received yet.

```
data Value = Value(Int);
type Pnum = Int;
data Proposal = Proposal(Pnum, Value value) | None;
```

By applying the name `value` to the second parameter of the `Proposal` constructor, a function called `value` is automatically generated for extracting the value from a proposal. It is important that this function is only called when it is absolutely certain that the proposal in question is not `None`. The value will only need to be extracted from a proposal at one place in the implementation, so this is easily controlled.

The agents will however need to extract the numbers from proposals on multiple occasions. The numbers are, among other things, the entities used for comparing two proposals. An automatic `number` function on the proposals would require a lot of prior checks for `None` throughout the whole implementation for safety to be guaranteed. It is therefore better to define the `number` function manually.

No actual proposals will be made with the proposal number 0. Consequently, 0 can safely be defined as the proposal number of `None`. This is also convenient because `None` will rank lower than any other proposal, which simplifies some later parts of the implementation.

The function is realized using the **`case`** statement on the proposal, and both the constructors of the data type are mapped to a return value. Some simple pattern matching is used to extract the proposal number of an actual proposal: The `Proposal` constructor is used as a pattern with an unbound variable `n` for the proposal number, as well as the `_` wildcard to express that the value is ignored.

```
def Pnum number(Proposal p) = case p { None => 0;
                                       Proposal(n, _) => n; };
```

4.5.2 Agents

Three different interfaces will be needed in the model – one for each type of agent. Each interface will have methods that correspond to the different kind of messages that it will receive.

Because the proposers are not guaranteed a response to every prepare request, the response to a request will not be given as a return value, but rather as a separate method defined in the `Proposer` interface which will be called by the acceptors.

In most of the remote procedure calls the receiver needs to identify the caller, to respond or for some other purpose. The methods corresponding to these RPCs will include a parameter `caller`.

Acceptors need a method for each of the two types of requests they will receive from the proposers. Both types of messages that are sent to the acceptors are a type of requests, so the simple method names `prepare` and `accept` should be clear enough. Acceptors that have accepted a value will notify the learners by calling the learners' `report` method.

The proposers only receive one type of messages, so they are given the method named `response`, which will be called by an acceptor responding to a prepare request.

The Paxos algorithm starts with the proposers, so it is assumed that when the proposers are initialized, the algorithm can begin. The basic class implementing the proposers will be made active by declaring a `run` method and putting the logic for starting the prepare phase inside said method. The basic proposer class will not concern itself with timeouts, so it will not be restarting the prepare phase on its own. But to make it possible to restart the proposer externally, the `run` method can made visible to the public through the interface.

However, it is not desirable that other agents in the model see this method, nor that other classes implementing the interface have to define a `run` method. Therefore `ActiveProposer` is defined as a sub-interface of `Proposer`. It will be made sure that just the process controlling the proposer will see the `ActiveProposer` interface, while the other agents only see a regular `Proposer`. Thus the `run` method remain hidden to the rest of the system.

For the implementation of faulty communication later, it will also become handy to know the proposer's current proposal number, so the function `currentPnum` is also defined on `ActiveProposer`.

Listing 4.1: Interfaces for Paxos agents

```
interface Learner {
    Unit report(Acceptor caller, Proposal p);
}

interface Acceptor {
    Unit prepare(Proposer caller, Pnum n);
    Unit accept(Proposal p);
}

interface Proposer {
    Unit response(Acceptor caller, Pnum n, Proposal accepted);
}

interface ActiveProposer extends Proposer {
    Unit run();
    Pnum currentPnum();
}
```

As mentioned earlier, a process may act as more than one type of agent. For each of the 3 different types of agents, a process may or may not play that role, giving a total of 2^3 8 combinations. The combination that does not includes any of the types is obviously disregarded, so there are in total 7 different possible variations of processes. For simplicity this model will use one class per type of agent, each implementing only that one type.

First of all, the ABS language does not offer much facility for reuse of code, so only implementing three distinct classes requires less code and makes the demonstration simpler.

More importantly, this will make the model easier to analyze and verify, but it will not limit the model or affect its correctness in any way. The behavior of each type of agent is strictly defined separately from the others. If one process should act as multiple types of agents, the correct behavior for each of its roles towards all other agents in the network is still defined independently and not overlapping.

If the logic of the different types of agents was kept separated on the process, the different parts could communicate through the same means as they communicated with other processes. To the model, it would then not make any difference, and the logical presentation of that process could be split into one for each type of agent.

Internally, a multi-agent process could of course do certain optimizations. For instance it would not be particularly efficient for a process acting as a proposer to start sending out prepare requests with a proposal number n if it was also acting as an acceptor and it had already responded to a prepare request with a number higher than n.

For a correct implementation however, the optimizations should not change a process's general behavior towards the rest of the network. It would still have to regard itself merely as one among the other agents and obey the majority. Therefore, the various parts acting as different types of agents on a correctly implemented multi-agent process could still be logically regarded as separate processes.

Through Sections 4.5.3 and 4.5.4 the individual methods of the different classes, and the classes themselves, will be covered in the order they are needed during a run of the algorithm.

4.5.3 The prepare phase

This section covers the details of the implementation that is related to the prepare phase of the algorithm. First is the part of the implementation that is needed to initialize a proposer and let it start sending prepare requests to the acceptors. The acceptor class, and how it handles the prepare requests with the `prepare` method, is then covered.

Sending prepare requests

All the information specifically needed by the `Proposer` class in the prepare phase, is passed as class parameters, and described in the following paragraphs. Because the basic proposer implementation will be reused in the module modeling faulty communication, the class implements the extended interface `ActiveProposer`.

Each proposer that is instantiated will be given a value which it will try to propose. This is the only class parameter of the proposer class that is of type `Value`, so its purpose should be quite clear even though, for brevity, it is only given the name `v`.

As all proposers must use different proposal numbers, a way of describing the disjunct set of proposal numbers available to the proposer is necessary. To solve this, the proposer is given two numbers as class parameters. The first number is a unique initial proposal number. To uphold the safety of the algorithm, the proposer will also use this number as a reference to the current proposal number, so the first of the two parameters will be called `current`. The second number is the total number of proposers in the network, `nProposers`.

The unique initial proposal number given to each proposer is taken from a consecutive range of integers, starting at 0. Each time a proposer needs a new proposal number, it will increase the previous number by `nProposers`. As `nProposers` is the same for all proposers, this is enough to guarantee that no two proposers will ever use the same proposal number, as long as they were fed different values for `current`: The proposal numbers from each individual proposer is always the same, modulo `nProposers`.

The proposers obviously need to know how they can contact the acceptors, so they are given a list of acceptors, just called `acceptors`. When testing the model, it may not always be desirable that every proposer knows about every acceptor. Only letting a proposer know about a subset of the acceptors can serve as a simple way of modeling faulty communication in a controlled manner, and can be useful for testing specific corner cases of the algorithm. Later however, when the proposer starts to receive responses to the requests, it must know exactly how many acceptors there are in the system. The total number of acceptors will be passed to the proposer through `nAcceptors`, the last of the class parameters.

There are a few more fields that the proposer will need later in the next phase of the algorithm. In ABS all fields must be declared in the beginning of the class, before the first method declaration. Therefore the remaining fields will briefly be introduced here and included in the following listing. More details about these fields will be given in Section 4.5.4 on page 42.

For robustness against duplicated messages, the proposer uses `responders`, a set of acceptors, to remember the ones that have already responded to the prepare requests. When acceptors respond, they might report that they have already accepted a proposal. The proposer is obliged to remember the highest-numbered proposal reported among the responses, and this proposal will be stored in `maxResponse`. The last field is a boolean value called `cutoff` and indicates whether the proposer has finally sent out the accept requests.

To satisfy the `ActiveProposer` interface, the proposer also needs to define the method `currentPnum`. This method simply returns the proposal number `current` and is not shown in the listings.

Listing 4.2: Declaration of the Proposer class and its fields

```
class Proposer(Value v, Pnum current, Int nProposers,
               Int nAcceptors, List<Acceptor> acceptors)
implements ActiveProposer {
```

```
Set<Acceptor> responders = EmptySet;
Proposal maxResponse = None;
Bool cutoff = False;
```

It has been established that the basic proposer in the `Paxos` module will not by itself abandon a proposal and start over with the prepare phase. However, the `run` method can be triggered manually, so the proposer must make sure that no previous run can interfere with the prepare and accept phases it is about to enter. It picks a brand new proposal number by adding `nProposers` to `current`, and resets its progress by setting the `responders` field back to `EmptySet` and `cutoff` to false.

As long as a proposer is not passed a negative number for its initial value of `current`, the proposer stays clear of the proposal number 0, which should not be used in this model.

To iterate through the list of acceptors, a while-loop is used. Because of the structure of the list, an efficient way to extract each element is to treat the list as a stack, *popping* off one value at a time. The list structure is defined as an algebraic data type, so it is side-effect free. The standard stack operation *pop* is therefore imitated by replacing the list with the *tail* of the list. Being side-effect free also ensures that popping off elements of a copy of the list does not affect the original.

Listing 4.3: The proposer's run method – starting point of the Paxos algorithm

```
Unit run() {
    current = current + nProposers;
    responders = EmptySet;
    cutoff = False;

    List<Acceptor> stack = acceptors;

    while (stack != Nil) {
        Acceptor next = head(stack);
        next!prepare(this, current);
        stack = tail(stack);
    }
}
```

Responding to prepare requests

Similarly to how the proposers are initialized with a list of acceptors, the acceptors will be initialized with a list of learners. When an acceptor has accepted a value, it will notify the learners. The list of learners is the only class parameter of the acceptors.

Regarding the actual algorithm, there are only two things that an acceptor needs to remember: The highest proposal number it has received, and the proposal it has accepted, if any. These two values are stored in the field `max` and `accepted` respectively.

The acceptor's role in the prepare phase is quite simple, and the `prepare` method equally so. The parameter `caller` tells the acceptor which proposer it

should respond to, and `n` is a proposal number. If the incoming request has a higher proposal number than the one the acceptor has received so far, it will respond to the proposer and update the field `max`.

It has been established that 0 will not be used as a proposal number in a request in this implementation. With the initial value of `max` is set to 0, no special logic is needed to handle the first prepare request an acceptor receives.

Likewise, the initial value of `accepted` is `None`, which should not be used for an actual proposal. The acceptor includes the value of `accepted` in the response directly, and it can easily be interpreted by the proposer.

Listing 4.4: Head of the Acceptor class and the prepare method

```
class Acceptor(List<Learner> learners) implements Acceptor {
    Proposal accepted = None;
    Pnum max = 0;

    Unit prepare(Proposer caller, Pnum n) {
        if (n > max) {
            max = n;
            caller!response(this, n, accepted);
        }
    }
```

4.5.4 The accept phase

The accept phase starts after the proposer has received responses to the prepare requests. The accept requests will therefore be sent from within the `prepare` method. This section covers this method, as well as the acceptors' `accept` method, which in turn calls the learners' `report` method.

Handling responses to the prepare requests

In the accept phase the three final fields of the `Proposer` class in Listing 4.2 come into use.

The proposer cannot start sending accept requests until it has received responses from a majority of the acceptors. Because it is assumed that messages may be duplicated, the proposer needs to keep a list of all the responders so that one acceptor is not erroneously counted twice. The predefined `Set` is a data type well suited for this use. With the function `size` the set is also used as a counter, which is compared to the total number of acceptors in the system.

The responses may reveal that one or more acceptors already have accepted a value. If so, the proposer is obligated to take the value of the highest-numbered proposal from these responses, and use it as the value in the accept requests. The field `maxResponse` is used for remembering the highest-numbered proposal the proposer has encountered.

As soon as a proposer has received responses from more than half of the acceptors, it can start sending accept requests. It should be perfectly safe for the proposer to send the requests to all the acceptors it knows about, also those that have not responded.

But if the proposer is going to send accept requests to everyone as soon as it has received responses from a majority of the acceptors, it should not repeat this for every subsequent response. The boolean `cutoff` is used as a flag to indicate whether or not the proposer has sent out the accept requests.

The method `response`, shown in Listing 4.5 on the next page, takes 3 parameters. Again, `caller` is used to identify the agent sending the message. The proposal number `n` tells the proposer what prepare request the acceptor is responding to. The parameter `reported` is either `None` or a proposal, and tells the proposer whether the acceptor already had accepted a value.

If the proposer starts the prepare phase more than once, it must completely abandon the old proposals. The `run` method in Listing 4.3 showed the proposer emptying the set of responders and picking a new proposal number as the current prepare phase was initiated. Calls to the `response` method that includes a different proposal number than the current, are outdated and will be ignored.

A large if statement is wrapped around the body of the method. It compares `n` to `current` and checks that the caller is not already registered a responder; No outdated or duplicated responses can affect the proposer. Inside the if statement, the set of responders is updated immediately. Then, up to two of the following things may happen: The proposer may update the `maxResponse` field, and it may start sending the accept requests to the acceptors.

The first of the two things is handled in a one-line if statement. With the function `number` which was defined in Section 4.5.1, comparing two proposals is easy and safe. The function extracts the proposal number from an actual proposal, or returns 0 if the proposal is `None`. The highest-numbered proposal that the proposer has yet encountered, and the proposal reported in the current response, can be be compared through the `number` function even if at least one of them is `None`.

The proposer will send the accept requests only if a majority of the acceptors has replied, and if the requests have not already been sent. The `cutoff` flag is checked, and the size of the set `responders` is compared against the total number of acceptors. The proposer puts together a new proposal, `p`, with the current proposal number and the value which was initially passed to the proposer, `v`. If any existing proposals were reported by the acceptors, then the highest-numbered of these will have been stored in `maxResponse`, and the proposer uses the value of this response instead of `v`.

To send the accept requests, the proposer loops through the list of acceptors, similar to how it sent the prepare requests in the `run` method. After the requests have been sent, the `cutoff` flag is set to `True`, indicating that the proposer's part of the accept phase is completed; It will not act again unless it should abandon the proposal and start over with the prepare phase.

Accepting a proposal

By the time a proposer sends the accept requests, it has acquired an overview of the system through its prepare requests and based a decision on these. Trusting that the proposers are following protocol, the acceptors only need to do as they are told. No more logic is needed, except for one thing: An acceptor must of course honor the promise it made when it responded to the prepare request earlier – that it

Listing 4.5: The proposer's response method

```
Unit response(Acceptor caller, Pnum n, Proposal reported) {
    if (n == current && ~contains(responders, caller)) {
        responders = insertElement(responders, caller);

        if (number(reported) > number(maxResponse))
            maxResponse = reported;

        if (~cutoff && size(responders) > nAcceptors / 2) {
            Proposal p = Proposal(n, v);

            if (maxResponse != None)
                p = Proposal(n, value(maxResponse));

            List<Acceptor> l = acceptors;

            while (l != Nil) {
                Acceptor next = head(l);
                l = tail(l);
                next!accept(p);
            }

            cutoff = True;
        }
    }
}
```

Listing 4.6: The acceptor's accept method

```
Unit accept(Proposal p) {
    if (number(p) >= maxPrepare) {
        accepted = p;

        List<Learner> l = learners;

        while (l != Nil) {
            head(l)!report(this, value(p));
            l = tail(l);
        }
    }
}
```

would not accept any proposals numbered less than that prepare request.

A simple if statement is enough to honor this promise. The proposal is accepted if its number is greater than or equal to `maxPrepare`. When the acceptor changes `accepted` to the new proposal, the actual accept phase is over.

The only thing left to do is to notify the learners. The acceptor loops over the list of learners and sends them the newly accepted value through an asynchronous call to their `report` method.

It is important to remember that even if a proposal has been accepted by an acceptor, the value is not necessarily *chosen*. As the short, almost anticlimactic third line of the above code listing demonstrates, the state of being *accepted* is merely a local property of one acceptor. How the chosen value is learned, is covered in the next section.

4.5.5 Learning the chosen value

For a value to be chosen, a majority of all the acceptors in the network must have accepted a proposal with this value. Thus accepting a proposal is a kind of vote for the value to be chosen. The fact that a value has been chosen, is not in itself dependent on any agents knowing it; It is however the learners' job to find out.

There are different ways of letting the chosen value to be known to every learner, and there are cons and pros associated with each strategy. Lamport [14, p.6] discusses the different approaches, which range from using a single distinguished learner, to having every acceptor notify every learner.

A distinguished learner would have the accepted values reported from the acceptors, and when it had learned the chosen value, it would let the other learners know. This minimizes the traffic in the network, at the cost of having a single point of failure. Having every acceptor communicate with every learner, on the other hand, causes much larger number of messages to be sent, but each learner could potentially learn the value sooner, and it should be more robust.

There are also of course many possible solutions between the two extremes, but exactly how the chosen value reaches each learner is less important in this model; The acceptors will simply report to all the learners they know whenever they accept a proposal. The implemented learner concludes that a value has been chosen when a majority of the acceptors have reported that they have accepted it.

The learners take no part in the actual consensus algorithm of Paxos. Therefore an agent that is only acting acting as a learner knows nothing about the events leading up to the moment when an acceptor finally reports a proposal. It will only have to trust that the proposers and acceptors are doing their part to uphold the safety requirements which were explained on page 31. These requirements state that only one value should be chosen, and that an agent will never learn an incorrect value. When the learner learns the chosen value, it will therefore never need to re-evaluate it, and subsequent proposal reports can be ignored.

The `Learner` class is given an integer parameter `nAcceptors` so that it can determine when a value has reached a majority of the votes. The number of votes for each value is kept in the map `counts`. The learner also keeps a map called `votes` which contains the latest proposal accepted by each acceptor. The field `learned` is a `Maybe` with a `Value` type parameter. Until the chosen value has been learned, this field remains `Nothing`.

When a newly accepted proposal is reported, the learner checks if a proposal has already been reported by the caller. This is crucial both because messages can be duplicated, and because an acceptor may accept more than one proposal. Only the latest accepted proposal from each acceptor counts as a vote. If the acceptor has already accepted a proposal, the old proposal will be found in the `votes` map. Because messages can be delayed, the proposer must also check that the current report is sent later than the already registered vote. Two proposals are compared by their proposal number, and the one with the highest number is deemed the newest. If the current report includes an old proposal, the report is dismissed.

Listing 4.7 on the next page shows the realization of the `report` method. The body of the method will only be executed if the chosen value has not been learned. A boolean variable `valid` is declared to indicate whether the incoming report should count as a vote for the given value. The predefined function `lookup` is used to see whether a proposal is already registered to the caller. The return type of said function is a `Maybe` which either holds the existing proposal or `Nothing`.

In case the lookup did *not* return `Nothing`, it means that a vote already is registered on the caller. The registered vote is extracted from the `Maybe` type with the `fromJust` function, and the two proposals are compared on proposal numbers.

If the current report contains a newer proposal, the number of votes for the new value will be increased in the next step of the method. But first the number of votes for the *old* value needs to be *decreased*, so that one acceptor does not get two votes. Because the proposer was registered with an existing vote for a proposal, the value of that proposal has at least one vote, and it must be present in the `counts` map. The lookup is therefore performed with the `lookupUnsafe` function, which unpacks the `Maybe` type and returns its content directly. The number that was looked up is decreased by one and put back into the map.

If instead the current proposal was older than the existing one, the `valid` flag is set to `False`, so that the next part of the method is skipped, and the state of the learner remains unchanged.

When the number of votes for the current proposal is increased, a third type of map lookup is used. The `lookupDefault` takes an extra parameter which specifies a default value in case the lookup fails. The function can therefore, like `lookupUnsafe`, omit the `Maybe` wrapper and return an integer directly. The `counts` map is updated with the correct number of votes for the value in the reported proposal, and the proposal is mapped to the caller in the `votes` field.

Finally, the new count is checked against the number of acceptors. If the count is greater than half of the acceptors, the learner knows that the value has been chosen. Finding a chosen value is celebrated with an asynchronous call to a little private method, `learn`, which simply sets the `learned` field accordingly. The extra method call provides visual confirmation in the sequence diagrams.

4.5.6 Testing the model

The basic implementation of Paxos is tested in a separate module. All the data types are imported, so that the different agents can be initialized. Listing 4.8 illustrates how to set up a test program for the `Paxos` module.

Because the different type of agents keep references to other agents, the classes have to be instantiated in the following order: learners – acceptors – proposers. The

Listing 4.7: The Paxos Learner class

```
class Learner(Int nAcceptors) implements Learner {
    Maybe<Value> learned = Nothing;
    Map<Value, Int> counts = EmptyMap;
    Map<Acceptor, Proposal> votes = EmptyMap;

    Unit report(Acceptor caller, Proposal p) {
        if (learned == Nothing) {
            Bool valid = True;
            Maybe<Proposal> oldVote = lookup(votes, caller);

            if (oldVote != Nothing) {
                if (number(p) > number(fromJust(oldVote))) {
                    Value v = value(fromJust(oldVote));
                    Int count = lookupUnsafe(counts, v);

                    counts = put(counts, v, count - 1);

                } else
                    valid = False;
            }

            if (valid) {
                Value v = value(p);
                Int count = 1 + lookupDefault(counts, v, 0);

                counts = put(counts, v, count);
                votes = put(votes, caller, p);

                if (count > nAcceptors / 2)
                    this!learn(value(p));
            }
        }
    }

    Unit learn(Value v) {
        learned = Just(v);
    }
}
```

learners are only told how many acceptors there are in the system, but do not need any references to actual acceptors. The acceptors are given a list of learners, and the proposers are given a list of acceptors.

The proposers take 4 additional parameters before the list of acceptors. Some arbitrary value is chosen for each proposer, and they are given a unique integer on which to base their respective proposal numbers. These integers must be chosen from a range from 0 and up to the total number of proposers. The two final parameters are the number of proposers and the number of acceptors.

Competing proposers

As a simple test case, a system is set up with two proposers, one acceptor and one learner. This configuration is sufficient to demonstrate the two possible outcomes when two proposers are racing against each other in a scenario where progress is assumed. This was discussed in Section 4.5.

Listing 4.8: A simple test of the Paxos module

```
module PaxosTest;
import * from Paxos;

{   // main:
    Int nAccs = 1; // number of acceptors
    Learner l = new Learner(nAccs);
    List<Learner> learners = list[l];

    Acceptor a1 = new Acceptor(learners);
    List<Acceptor> accs = list[a1];

    Int nProposers = 2;
    new Proposer(Value(10), 0, nProposers, nAccs, accs);
    new Proposer(Value(99), 1, nProposers, nAccs, accs);
}
```

Figure 4.1 shows the sequence diagrams of two different runs of the test program in Listing 4.8.

The upper diagram shows the *accept* request from the first proposer being sent to the acceptor before the *prepare* request of the second proposer. The second proposer is then informed of the accepted proposal, and must reuse it in the accept request.

In the lower diagram, the first proposer's accept request comes later than the second proposer's prepare, and is thereby ignored.

Accepting different values

In the previous test case there was only one acceptor. Had a value first been accepted by that acceptor, it would be impossible for a proposer to finish the prepare phase afterwards without hearing about the accepted value. The same is true if there were two acceptors, because if only one of them had an accepted value, a proposer would need both acceptors' responses to advance to the accept phase.

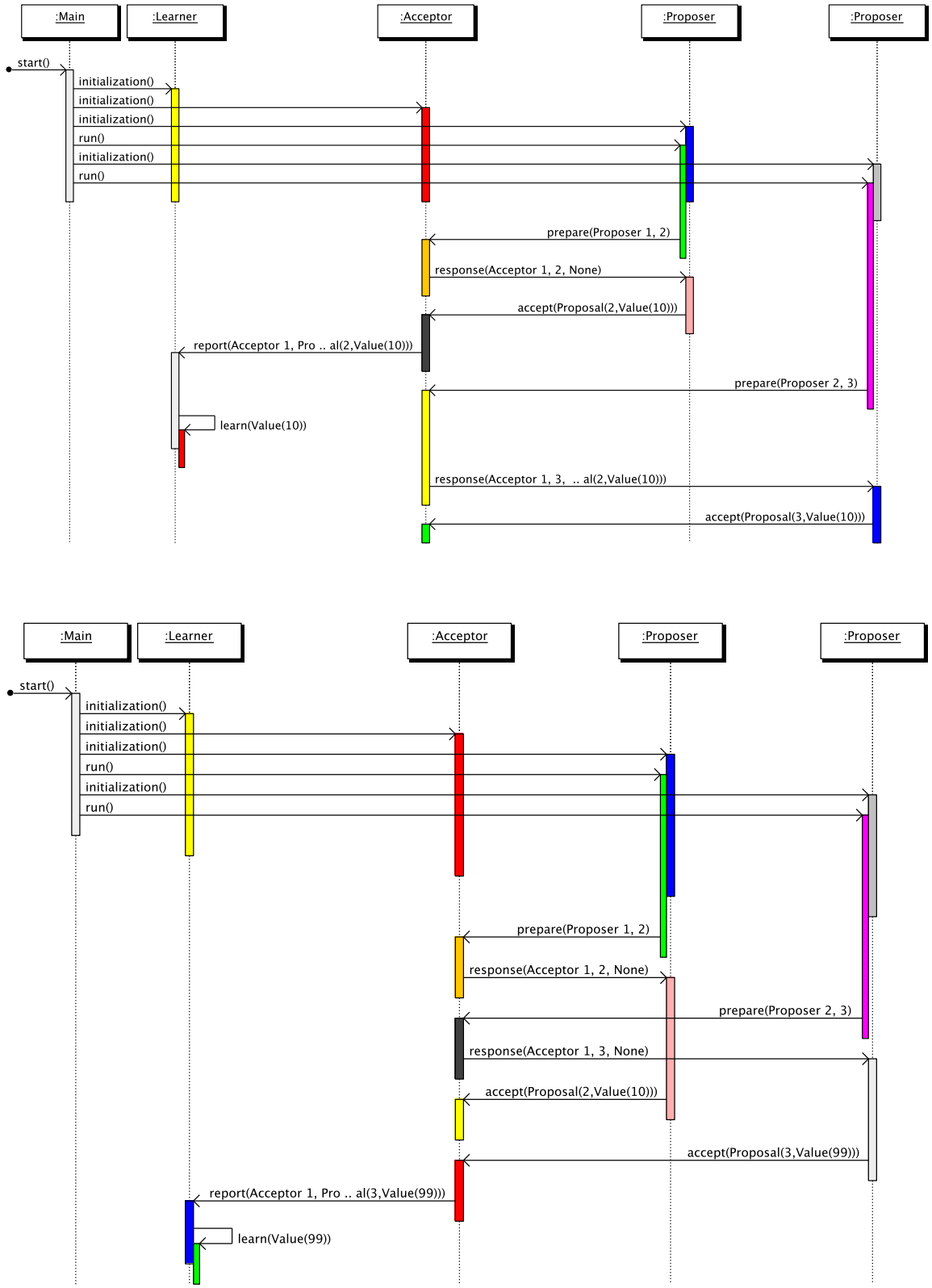


Figure 4.1: Two test simulations of the Paxos algorithm in which two proposers compete to have a value accepted.

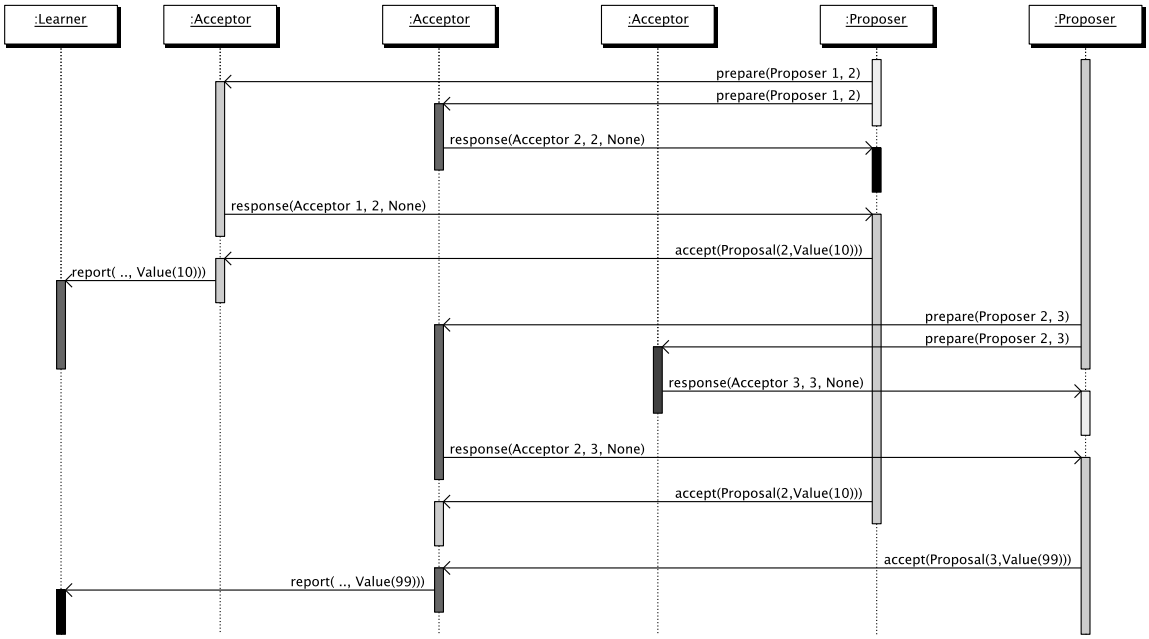


Figure 4.2: Two different values being accepted

If however there are three or more acceptors, it is possible for a proposer to have a proposal accepted even if there are other proposals accepted (however, if there was a value *chosen* – simultaneously accepted by more than half the acceptors – the proposer would hear about it during the prepare phase).

In order to have two different values being accepted at once, a configuration of three acceptors and two proposers can be set up. To create the situation depicted in Figure 4.2, the first proposer was initialized and given a list containing the two first acceptors, while the second proposer was similarly given the two last acceptors.

With the interactive scheduler in the Eclipse plug-in, the first proposer is advanced as far as to the accept phase, just having time to send the first accept requests. The second proposer is then "woken up" and advanced to the accept phase up until to the moment where it has sent its first accept request.

Being the basic implementation, the last proposer would of course succeed in having its value accepted by its second acceptor as well, had the model been ran further. But as the situation stands right there, only two acceptors have accepted a value, and two different values at that.

4.6 Modeling faulty communication

This section explains a way of adding faulty communication to the Paxos module. A new module `FaultyAgents` will be created, importing all the types from `Paxos`. There will be defined new classes implementing the interfaces of the three different types of agents. These new classes will cause duplication, delay, and loss of messages, while the old classes from the previous module will be reused for the logic of the actual Paxos algorithm.

Intercepting messages

The classes provided in the `Paxos` module will hereafter be referred to as the *basic* classes. The new implementations of the agent interfaces provide a sort of simulated environment for the basic classes.

Faulty communication will be simulated by having instances of the basic classes kept as objects local to the new classes: When initialized, the basic agents will not be given references to other basic agents, but rather to agents from the new module. Oblivious to the difference, the basic agents will act and send messages as normal, but they will be communicating solely to the objects of the new classes: The `FaultyAgents` module provides a way to *intercept* the messages between the basic agents.

To intercept the messages, there will be two different kinds of classes implemented. The classes called *proxy classes* are a sort of auxiliary class that will receive *outgoing* messages from the basic classes and *redirect* these messages to other agents. The other type of classes defined is the main classes that will hold the objects of the basic classes, as well proxy classes. These main classes are just called the *faulty* counterparts to the basic classes. The faulty agents intercept the *incoming* messages and redirect them to the basic agents.

For instance, when the basic `Proposer` class starts its `run` method, it sends asynchronous message to the acceptor objects it was given upon its initialization. In the faulty module, the list of acceptors given to the basic proposer contains only references to acceptors of the proxy type.

Each proxy corresponds to an actual agent, which is initialized and running somewhere else in the system. When the proxy receives a function call from the basic agent, it asynchronously calls the same function on the *remote* agent – the agent that the proxy corresponds to. But, in the outgoing method call, the proxy replaces the `caller` argument with a reference to a faulty counterpart of the basic agent.

The faulty agents keep track of both basic agents and agent proxies. A faulty agent initializes a basic agent of the same type, and one proxy for each of the remote agents to which the basic agent will be sending messages. Like the proxies intercept and redirect the basic agents' outgoing messages, the faulty agents intercept and redirect the incoming messages, replacing the `caller` argument with a corresponding proxy.

Figure 4.3 on the following page illustrates the flow of messages when a basic agent – under a faulty agent's control – sends a message to a remote agent and the remote agent sends a reply back. The main faulty agent is in control of the others, and is therefore called the *master*. The agents under the master's control are the master's *slaves*. The slaves are initialized by the master and run in the same COG, so the slaves are positioned below the master. The remote agent runs in another COG, and its internals are unknown to the faulty agent; The remote may itself be a faulty agent comprised of separate agent components.

Faulty communication

In Paxos there are three different ways in which it is assumed that the communication can fail: Messages can be duplicated, delayed, and lost. This is

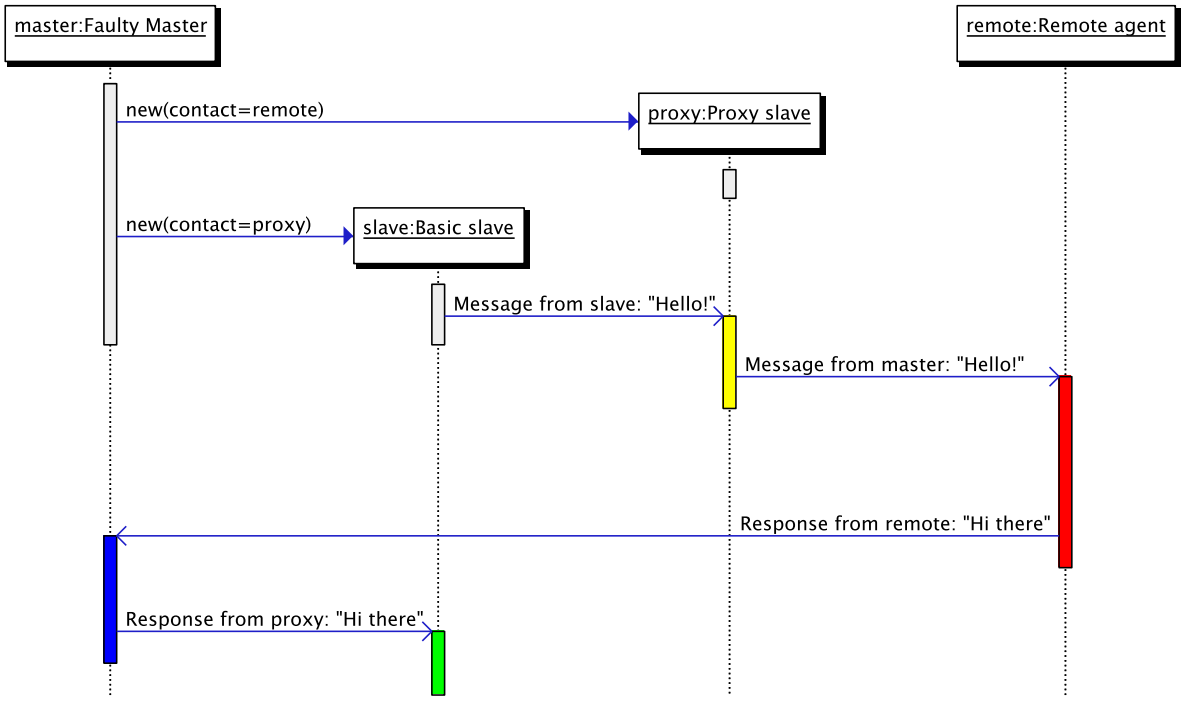


Figure 4.3: A faulty agent and a proxy agent intercepting a basic agent's messages.

simulated by the classes introduced in this the `FaultyAgents` module when they intercept the method calls to and from the basic agent classes.

The duplication and loss of outgoing messages is easily produced using the proxy classes. When the proxies intercept a method call, they can simply send it twice to the intended target in order to duplicate it; Message loss is achieved by not sending a message at all. Message delay is more complicated, and may have different consequences, depending on an agent's role.

In a sense, acceptors and learners play *passive* roles in Paxos, in that they only act when they receive messages from other agents – It is the proposers that initiate the algorithm and start sending messages, while the other agents merely *respond*.

Thus the delay of a single message has in it self no impact on the acceptors or the learners. They will simply react when the messages arrive, and the duration between each message is irrelevant. However, if some messages are delayed, while other are not, then messages can arrive in an arbitrary order. The proxies can mimic this aspect of message delay by holding on to an outgoing message until after another message has been intercepted and redirected.

The proposers are not only reacting to the incoming responses from the acceptors – they can also decide to start over with the prepare phase and send new requests. Parallel to the well known *halting problem*, an agent cannot know whether a message has truly been truly lost or simply is delayed. To avoid waiting forever, the agent will at some point have to assume that the message is lost. At that time the message is said to *time out*.

In the ABS model, the time-out of responses will not actually be triggered by a lapse of time. Instead, a faulty proposer that receives a response from a

remote acceptor can pretend that the message did not arrive, and instead initiate a new prepare phase by calling the basic proposer's `run` method. In that event, the incoming message call does not indicate the reception of a response, but rather signifies the time at which the proposer considered the pending responses to have timed out. Other responses may still arrive after the time-out was decided, but these should be disregarded if the basic proposer has been implemented correctly.

4.6.1 Implementation

Like the basic `Paxos`, the new module `FaultyAgents` defines classes that can be imported and initialized in another module. In addition to the new faulty classes, the basic data types from `Paxos` is still needed to run the simulation. Everything in the `Paxos` module is therefore imported *and* exported. From the new module though, only the classes for the faulty proposers and acceptors are exported; These classes will initialize and handle the proxies, which is therefore hidden from the outside. Each of the new classes, both *faulty* and *proxy* variants, will implement a corresponding interface from `Paxos`, so they are all fully compatible with the classes from the old module.

```
module FaultyAgents;
export FaultyProposer, FaultyAcceptor;
export * from Paxos;
import * from Paxos;
```

Whether messages are lost, delayed, duplicated, or just delivered without any trouble, is determined at random. A function `chance` will be used for declaring various probabilities. For instance, `chance(3,5)` means a 3/5 probability.

```
def Bool chance(Int c, Int n) = random(n) < c;
```

Acceptor proxies

The class responsible for intercepting the basic proposer's outgoing messages is `AcceptorProxy`. Objects of this class will be a link between a local basic proposer and a remote acceptor, as shown in Figure 4.3. The class is given two parameters: `owner`, a reference to the master proposer class; and `target`, a reference to a remote acceptor.

The class implements the `Acceptor` interface and the two methods that the interface defines. These methods will be called by the local basic proposer, and the acceptor proxy will, with a certain probability, call the respective methods on `target`, only with the `caller` parameter replaced with `owner`. Except for the method names, and the list of parameters, the methods `prepare` and `accept` are implemented identically.

An if statement with four branches determines what will happen with the outgoing message. The following probabilities given to the different branches are completely arbitrary.

Firstly an 80% chance is given for the message to be delivered successfully without any hiccups. The remaining 3 outcomes are given the same probabilities,

Listing 4.9: AcceptorProxy's method for intercepting the prepare request from a basic proposer.

```
1  Unit prepare(Proposer caller, Pnum n) {
2      Bool furtherDelay = False;
3
4      // Normal message delivery
5      if (chance(8, 10)) {
6          target!prepare(owner, n);
7
8      // Message duplication:
9      } else if (chance(1, 3)) {
10         target!prepare(owner, n);
11         target!prepare(owner, n);
12
13     // Message delay:
14     } else if (chance(1, 2)) {
15         if (isJust(delayed)) {
16             target!prepare(owner, n);
17
18         } else {
19             delayed = Just(Left(n));
20         }
21
22         furtherDelay = True;
23
24     // Message loss;
25     } else
26         skip;
27
28     if (~furtherDelay)
29         this.sendDelayed();
30 }
31
32 Unit sendDelayed() {
33     if (isJust(delayed)) {
34         case fromJust(delayed) {
35             Left(m) => caller!prepare(owner, m);
36             Right(p) => caller!accept(p);
37         };
38     }
39 }
```

one third each. In the second branch the message is duplicated by sending the method call to the remote acceptor twice.

The third branch handles message delays. Instead of passing the method call on to `target`, the acceptor proxy stores the parameters in a field called `delay`, and the method terminates. At the end of the *next* method invocation, the acceptor proxy will see the value of `delayed` and make the appropriate method call to `target`.

This scheme allows the effective order of two method calls to be swapped without any timing features needed. The message is also delayed without suspending the process, so that the message can terminate.

A delayed method call relies on a subsequent method call to be sent. If there is no more method calls to the acceptor proxy after a method call is delayed, the message is instead effectively lost.

If two subsequent method calls end up in the "delayed branch" of the if statement, the latter method call is redirected normally, and the first one is further delayed. A boolean variable `furtherDelay` is set to `True` when the current process should not send the delayed message.

Listing 4.9 shows the `prepare` method of the `AcceptorProxy` class and the helper method `sendDelayed`. The class also keep a field `delayed` which is of the type `Maybe<Either<Pnum, Proposal>>`. `Maybe` is a predefined parametric data type that either is `Nothing` or holds a value in the constructor `Just`. `Either` is another predefined parametric data type, which holds a value either in the `Left` constructor or the `Right` constructor.

The other method from the `Acceptor` interface, `accept`, is implemented identically to `prepare`, except for that `accept` is called on `target` rather than `prepare`. Also, when delaying the message, the value of `delayed` (line 19) is instead set to `Just(Right(p))`, where `p` is the method's `Proposal` parameter.

Faulty proposers

The faulty proposer class takes the same list of parameters as the basic implementation in Listing 4.2 on page 40. These parameters, except for the list of acceptors, will be passed on as is to a basic proposer object which the faulty proposer will initialize locally. The acceptors passed to the faulty proposer are replaced by corresponding proxy acceptors.

The faulty proposer is given two field: An `ActiveProposer` reference to a basic proposer, `slave`; and a list of acceptor proxies, `proxies`. The faulty proposer is made active so that it can initialize the two field after its own initialization. In the `run` method the faulty proposer loops through the list of acceptors and initialize corresponding acceptor proxies which are appended to the `proxies` field. The basic proposer `slave` is then `initialized` with the new list. The basic proposer class is also active, so `slave` will start right away, sending prepare requests to the proxies.

The basic proposer and the list of acceptor proxies are initialized with the keywords **`new local`**. They are then running in the same as the `FaultyProposer`, so externally the faulty proposer does not seem any different than other agents.

In the `response` method the time-out of messages is simulated. Because field `slave` is of type `ActiveProposer`, the basic proposer's `run` method is visible to the master. With an arbitrary 20% probability the faulty proposer will restart the

basic proposer's prepare phase instead of delivering the incoming response from the remote acceptor.

Timing out will however only be done if the responses that contain the proposer's current proposal number; Lower numbered responses must hail from a previous accept phase and are already outdated and timed out.

Proposer proxies

The proposer proxy class is implemented similarly to the acceptor proxy. It is given one reference to it's `FaultyAcceptor` master class, and one to `Proposer` target. The logic of the redirection of the `response` method calls is copied and pasted from the `AcceptorProxy`'s methods, off course only that the name of the called method and its parameters are changed appropriately to match the `response` method in Listing 4.5 on page 44.

The class's `delayed` field it also changed to a `Maybe` holding a `Pair` of `Pnum` and `Proposal`, which are the response method's parameters. Since the `Proposer` interface only has one method, less logic is needed when sending the delayed message, because there is no question as to what method that needs to be called.

Learner proxies

Again, the learner proxy class is implemented in a copy-and-paste fashion; The logic as the same as in the other proxy classes.

The learners do however not respond to any messages, so there is no need to change the `caller` parameter of the method calls. The basic learner class only use the callers as keys in a set to keep track of the reported proposals and avoid counting duplicate messages (Listing 4.7 on page 47).

Since the learners' `report` method only take one parameter except for `caller`, the `delayed` field of the `LearnerProxy` class only need to keep one value, so the field's type is just `Maybe<Proposal>`.

Faulty acceptors

Like the `FaultyProposer` class keeps one local basic proposer and multiple local acceptor proxies, the `FaultyAcceptor` class will keep one local basic *acceptor* and multiple *proposers*. However, the basic acceptors do not concern themselves with which proposers that are sending them messages, so it is the faulty acceptor that must keep track of the remote proposers and the corresponding local proposer proxies. It therefore keeps a map from remote proposers to local proxies.

The acceptors play a passive role in Paxos, and in the basic implementation, the acceptors do not know about the specific proposers until the proposers send them asynchronous method calls. The map of remote and local proposers is therefore populated as the `prepare` method is called by the remote proposers.

The list of learner proxies is however initialized upon the faulty acceptor's initialization, similar to how the faulty proposer initialized the acceptor proxies. The basic acceptor `slave` is then initialized with the list of learner proxies.

The `accept` method does not take a `caller` parameter, so this method is redirected to `slave` without needing to look up a corresponding proxy to the caller.

Listing 4.10: The FaultyProposer class

```
1  class FaultyProposer(Value v, Pnum init, Int nProposers,
2          Int nAcceptors, List<Acceptor> acceptors)
3          implements Proposer {
4
5      ActiveProposer slave;
6      List<Acceptor> proxies = Nil;
7
8      Unit run() {
9          List<Acceptor> ax = acceptors;
10
11         while (ax != Nil) {
12             Acceptor proxy = new local AcceptorProxy(this, head(ax));
13             proxies = appendright(proxies, proxy);
14             ax = tail(ax);
15         }
16
17         slave = new local Proposer(v, init, nProposers, nAcceptors, proxies);
18     }
19
20     Unit response(Acceptor caller, Pnum n, Proposal reported) {
21         Int current = slave.currentPnum();
22
23         // Timeout:
24         if (n == current && chance(2, 10))
25             slave!run();
26
27         // Normal delivery:
28         else
29             slave.response(caller, n, reported);
30     }
31
32 }
```

Listing 4.11: The FaultyAcceptor class

```
class FaultyAcceptor(Int nAcceptors, List<Learner> learners)
implements Acceptor {
    Acceptor slave;
    List<Learner> learnerProxies = Nil;
    Map<Proposer, Proposer> proposerProxies = EmptyMap;

    Unit run() {
        List<Learner> iterator = learners;

        while (iterator != Nil) {
            Learner proxy = new local LearnerProxy(head(iterator));
            learnerProxies = appendright(learnerProxies, proxy);
            iterator = tail(iterator);
        }

        slave = new Acceptor(learnerProxies);
    }

    Unit prepare(Proposer caller, Pnum n) {
        Proposer proxy;
        Maybe<Proposer> lookup = lookup(proposerProxies, caller);

        if (lookup == Nothing)
            proxy = new local ProposerProxy(this, caller);
        else
            proxy = fromJust(lookup);

        slave.prepare(proxy, n);
    }

    Unit accept(Proposal p) {
        slave.accept(p);
    }
}
```

Listing 4.11 on page 58 concludes the implementation of the `FaultyAgents` module. The full implementation is listed in its entirety in the appendix. The next section provides a test program and a brief analysis at the workings of the module.

4.6.2 Testing

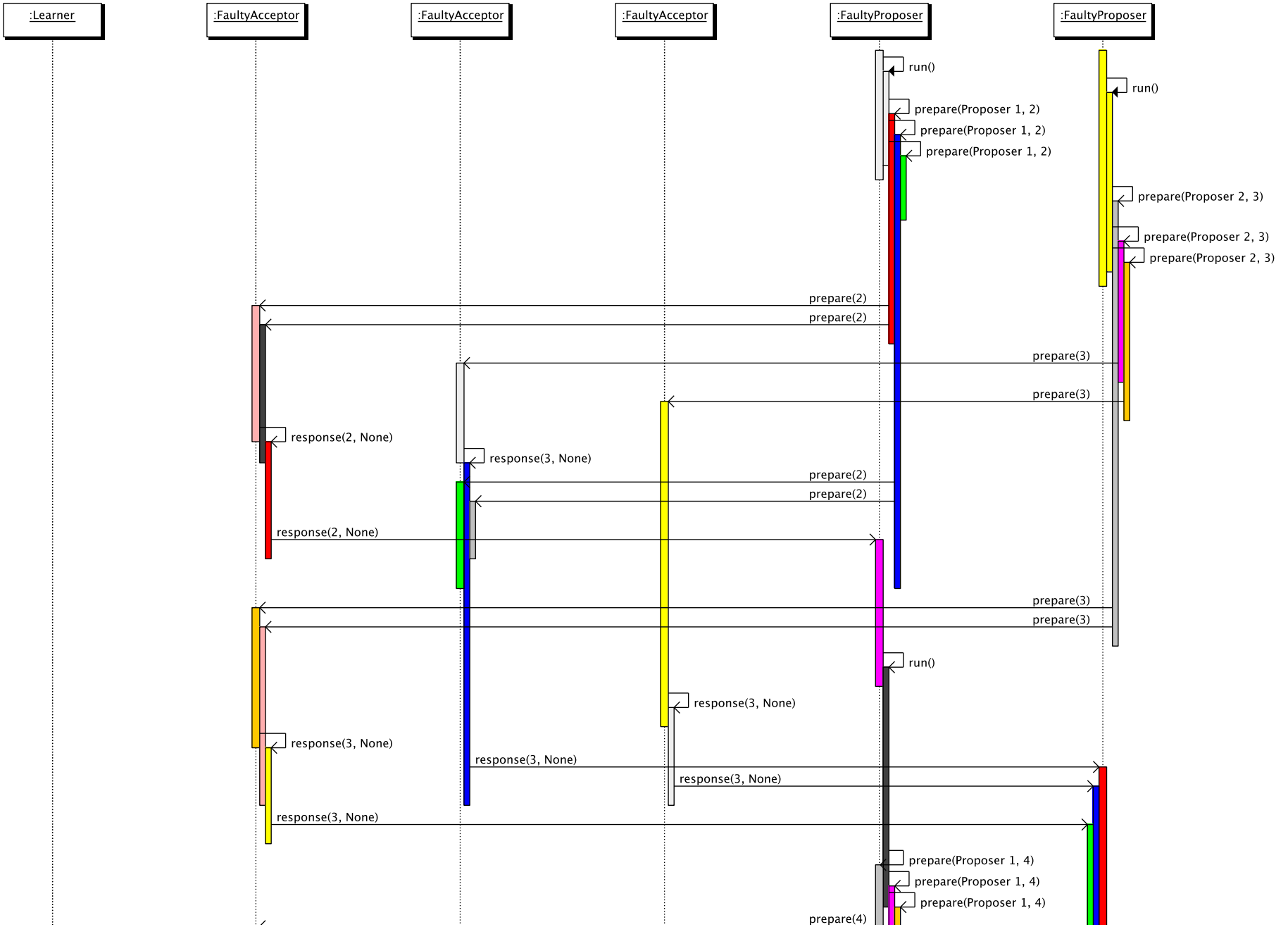
A simple module is set up, similar to the test of the basic `Paxos` module in Listing 4.8 on page 48, with three faulty acceptors, two proposers and one learner.

```
module FaultyTest1;
import * from FaultyAgents;

{
    Int nAccs = 3; // number of acceptors
    Learner l = new Learner(nAccs);
    List<Learner> learners = list[l];

    Acceptor a1 = new FaultyAcceptor(nAccs, learners);
    Acceptor a2 = new FaultyAcceptor(nAccs, learners);
    Acceptor a3 = new FaultyAcceptor(nAccs, learners);
    List<Acceptor> accs = list[a1, a2, a3];

    Int nProps = 2; // number of proposers
    new FaultyProposer(Value(7), 0, nProps, nAccs, accs);
    new FaultyProposer(Value(9), 1, nProps, nAccs, accs);
}
```

A test run in the Eclipse plug-in produced the diagram in Figure 4.4 (slightly edited later to fit inside the pages). The method calls that loop out from the COG and back again, are the basic agents calling their proxies. It can be seen that the first proposer wants to send prepare requests two the three acceptors. Of the three processes that are created on the COG, one just ends abruptly, signifying that the message was lost. The two other in this examples were duplicated, as can be seen from the two method calls going out from the same process to the same destination. When the first response reaches the proposer, the `run` method is started anew, meaning that the response timed out.

Figure 4.4 (on the following page): Sequence diagram showing an example run of the Paxos model with simulated faulty communication.

prepare(4)
prepare(4)
response(4, None)
response(4, None)
response(4, None)
response(4, None)
accept(Proposal(3,Value(9)))
accept(Proposal(3,Value(9)))
accept(Proposal(3,Value(9)))
accept(Proposal(3,Value(9)))
accept(Proposal(4,Value(7)))
accept(Proposal(4,Value(7)))
accept(Proposal(4,Value(7)))
accept(Proposal(4,Value(7)))
accept(Proposal(4,Value(7)))
accept(Proposal(4,Value(7)))
report(Proposal(4,Value(7)))
report(Proposal(4,Value(7)))
report(Proposal(4,Value(7)))
report(Proposal(4,Value(7)))
report(Proposal(4,Value(7)))
learn(Value(7))


:Learner
:FaultyAcceptor
:FaultyAcceptor
:FaultyAcceptor
:FaultyProposer
:FaultyProposer
run()
run()
prepare(Proposer 1, 2)
prepare(Proposer 1, 2)
prepare(Proposer 1, 2)
prepare(Proposer 2, 3)
prepare(Proposer 2, 3)
prepare(Proposer 2, 3)
prepare(2)
prepare(2)
prepare(3)
prepare(3)
response(2, None)
response(3, None)
prepare(2)
prepare(2)
response(2, None)
prepare(3)
prepare(3)
run()
response(3, None)
response(3, None)
response(3, None)
response(3, None)
response(3, None)
prepare(Proposer 1, 4)
prepare(Proposer 1, 4)
prepare(Proposer 1, 4)
prepare(4)

Chapter 5

Distributed Hash Tables: The Kademlia algorithm

Kademlia is a distributed hash table that was introduced by Petar Maymounkov and David Mazières in 2002 [19]. It is used in many peer-to-peer applications, among other for locating peers in the tracker-less torrents in the BitTorrent protocol.

This chapter covers the ABS implementations of a Kademlia distributed hash table.

5.1 Distributed hash tables

A hash table is a well known data structure, used to store pairs of keys and values, and provides a simple lookup facility. A *distributed* hash table is, as the name suggests, a distributed system that provides a similar functionality, but spreads the information across multiple nodes in a network, usually an overlay network on top of an underlaying system such as the Internet.

In order to find information in a distributed hash table (DHT), a client can join the network, becoming a node itself. The nodes take the place of the *buckets* in a regular hash table; Each of them will have the responsibility of storing some of the key–value pairs. To find information that it does not already know, a node will query other nodes in the network.

Unlike the non-distributed variant however, a DHT does not offer lookup in constant time, as there generally is no node with complete knowledge of the entire system. Instead, the very infrastructure of the system itself is also spread among the nodes, just like the key–value pairs.

This is achieved by letting each node be aware of a subset of the rest of the nodes, along with some sort of metric for a virtual distance within the network. A lookup will therefore most likely happen in multiple iterations, where each iteration should bring the inquirer closer by one order of magnitude – with respect to the chosen metric – to the desired node. An operational DHT is therefore fully decentralized.

A obvious benefit of this scheme is that it eliminates the problems associated with a single point of failure. With the infrastructure of the system spread out among the nodes, no node is more important than another. Still, to join a DHT a node needs to know about another node that is already connected to the network.

For a public service, there would have to exist a "gateway" somewhere that would help new nodes gain access to the service. The failure of such a gateway would deny new nodes to join the service, but it would not affect the nodes already connected.

5.1.1 Consistent hashing

An important property of a distributed hash table is that it is relatively tolerant to nodes joining and leaving at random. This property is called *consistent hashing* [11].

When a node joins or leaves a DHT, this is equivalent to a change in the size of the underlaying array of buckets in a standard implementation of a hash table. The hash function of such a hash table involves a modulo operation of the array size to avoid mapping a key to an index outside of the array. Changing the number of buckets renders the indices of most of the stored key–value pairs invalid, and each of these keys must be moved to a new index – an operation called *rehashing*.

With consistent hashing the amount of key–value pairs that must be moved when the amount of buckets changes should be minimal. The lookup of a hash is done in iterations, moving from node to node increasingly closer to the target. If for example a key–value pair P were to be moved from node B to node A, this would be due to a change in the set of nodes so that A now was the closest node to P – either A newly joined, or B left, the network. Still, most of the other nodes in the route from an arbitrary point in the DHT to P should stay untouched, similar to how the first elements in a linked list is not affected by a change at the end of the list.

5.2 The Kademlia protocol

Kademlia nodes communicate with each other though four different remote procedure calls: `PING`, `STORE`, `FIND_NODE`, and `FIND_VALUE`. The protocol dictates how the nodes should respond to these messages, and specifies a *node lookup* routine which uses these RPCs and is central to completing the basic tasks of a node.

The protocol also defines an XOR metric for distance between nodes, and uses this metric as the basis of the node's routing table. The routing tables are the nodes' tools for finding their targets during a lookup.

5.2.1 The XOR metric

When joining a Kademlia DHT, a node chooses a random 160 bit integer as ID. The keys stored in the DHT are hashed to the same range of integers, or ID space. When searching for a given key, a node will try to locate the nodes with IDs closest to the hash of the key. The distance between two IDs is the bitwise exclusive or (XOR) of their IDs [19, p.56].

With the XOR metric, a node's distance to itself is always 0. Because XOR is symmetric, the distance from A to B is the same as the distance from B to A for any two IDs A and B.

Figure 5.1: The longest common prefix of the binary representation of two IDs

$$\begin{array}{l} \mathbf{1\,0\,1}\,0\,1\,0\,1\,1\,0\,0 \\ \underbrace{\mathbf{1\,0\,1}}_{lcp}\,1\,1\,1\,0\,0\,1\,0 \end{array}$$

Regarding the ID space as a binary tree, Maymounkov and Mazières [19] define the *magnitude* of the distance between two IDs as the height of the smallest subtree that contain both IDs. A perhaps simpler, but equivalent, way to find this magnitude is by comparing the binary representation of the two IDs: Take the length of the longest common prefix of the IDs and subtract it from the number of bits in total.

As an example, figure 5.1 shows the binary representation of two 10-bit integers. The length of their longest common prefix (marked *lcp*) is 3, so the magnitude of their distance is 7.

5.2.2 The routing table

All the contacts that a Kademlia node knows about are stored in the node's routing table. With the distance metric explained above, the routing table partitions the ID space and maps each contact to a predictable location in the data structure, even as the structure expands. The routing table does not have room for all contacts that a node comes across, and this section will explain this dynamic, as well as how the routing table is used to quickly locate contacts that are close to a given hash.

Structure

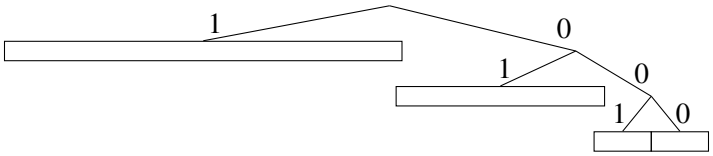
The routing table is a tree structure where each leaf holds a list of contacts. Each such leaf is called a k-bucket and it will hold at most k contacts, where k is a system-wide parameter that all nodes should adhere to. The routing table is an unbalanced binary tree which only branches off to the right-hand side so that each level of the tree has a left leaf.

Initially the routing table is flat, only consisting of a single leaf, or k-bucket. The first k unique contacts that are inserted into a k-bucket are simply appended to the back of the list in the bucket. What will happen the next time a contact is to be inserted into a bucket after it has been filled up, will depend on whether the bucket is a left leaf or a right leaf. The one k-bucket of the initial flat tree is considered a right leaf.

If the full bucket is a left leaf, the node must decide whether it should drop the left-most contact of that bucket and insert the new contact, or keep the old contact and drop the new one. The node should favor the old, existing contact and only decide to drop it if the node suspects that the old contact is no longer connected[1]. The topic of evicting contacts will not be discussed in greater detail here.

[1] Investigating whether or not a contact still is active typically involves probing the contact in question with `PING` requests. To reduce network traffic the node can rather place an old node in a *replacement cache* for contacts which are eligible for conviction, and delay the `PING` until another time [19, p.63].

Figure 5.2: The structure of a Kademlia routing table



When a contact is attempted to be inserted in a right-leaf k-bucket which is full, the k-bucket will first be divided into one left leaf and one right leaf k-bucket. Each of the contacts that were already stored in the bucket that was divided, will then be moved into one of the two new buckets, depending on its distance to the ID of the node that owns the routing table. The ID of the owner of the routing table will from now on be called the *base ID*.

The following example illustrates what happens when the initial routing table, a single k-bucket, is divided. The existing contacts in the k-bucket will be put into the new left leaf if and only if they have an ID which does not share any prefix with the base ID. That is to say, the contacts in the uppermost – zeroth – level of the tree, holds contacts whose ID has a longest common prefix with the base ID of length zero. The remaining contacts that are put in the right-hand side of the tree will then have a longest common prefix with the base ID of length 1 or more.

In general, a left leaf on level i of the tree, starting from the top and counting from 0, has a longest common prefix of exact length i. Half of all the possible IDs in the ID space disagrees with the base ID on the most significant bit. The first left leaf therefore covers half the ID space, and the second left leaf covers one quarter, and so on. The bottommost right leaf will always be the k-bucket containing IDs closest to the base ID; The topmost left leaf will always contain the IDs that are the furthest away. Each step down the tree is *one order of magnitude* closer to the base ID.

As there is a fixed limit to the amount of contacts in a k-bucket, and the branches on the right-hand side of the tree covers an exponentially smaller partition of the ID space, a node has better overview of the IDs closer to its own. At one level, k will be larger than the size of the remaining fraction of the ID space, and the tree will not be split any further.

With a 160-bit ID space, the tree of the routing table could in theory reach 159 levels, if k was 1. The right, bottom leaf if the tree is always the bucket that is responsible for the remainder of the ID space, after the ID space has been divided into exponentially smaller partitions for each level of the tree. The base ID is the only ID that could have been placed in the right-hand leaf on the 160th level, but the base ID is never inserted into the routing table, and therefore the last right leaf would not have been split.

Within the bucket, the contacts are ordered after the time they were last heard from. The left-most contact inside a bucket is the one of which there was last recently registered activity. When a node registers activity from an already existing contact, it will push the contact to the back of the k-bucket it belongs to.

Exceptions to the structure rules

Until now the data structure of the routing table has been described as a unbalanced tree in which each subtree only contains further branching on the right-hand side while the depth of the left-hand side is zero. This structure is effectively a linked list where each element is also a list.

However, Maymounkov and Mazières [19, p.60] mention one exception to the rules of the branching in the routing table. The exception is for handling a situation which can arise if there is an uneven distribution of IDs among the nodes.

A node may encounter many contacts with IDs that fall into the same left-leaf k-bucket. If the node has not come across more than k contacts in this subtree, it is allowed to split the left leaf k-bucket in order to keep an overview of the smallest subtree in the ID space around its own ID that contains at least k nodes. This scheme is called a *relaxed routing table*, and should ensure that the node can let itself be known to all the other contacts in the subtree during the *refresh* phase while joining a network, which is described in section 5.2.5. Relaxed routing tables is merely an optimization, and does not affect any other workings of the protocol.

Searching the routing table

To find the correct bucket for a contact, a path can be taken from the top of the tree in the routing table by considering the XOR of the contact's ID and the base ID. Starting with the most significant bit in the result of the XOR, a zero-bit means choosing the right-hand subtree, and a one-bit means the subtree to the left. In a tree without the optimization of split left-leaves, the search terminates as soon as a 1-bit is encountered in the XOR.

Figure 5.2 shows a routing table of a node whose ID has a binary representation starting with `000`.

For purposes that will be described under the paragraphs about the remote procedure calls and the *node lookup*, the node will have to pick out the k closest nodes to a given ID. In this search, the corresponding bucket may be empty, or may not contain as many as k contacts.

To find the closest contact to an ID when the appropriate k-bucket to that ID is empty, Maymounkov and Mazières [19] suggest to take the ID and reverse the bits in the positions that correspond to the empty k-buckets in the tree, and then search for the result of that calculation instead.

5.2.3 Remote procedure calls

There are four remote procedure calls defined in the Kademlia protocol.

- **PING** is the typical signal used for checking that a node is still online and also used for checking its response time. In the ABS model it is assumed that nodes stay connected, so `PING` is not paid much attention.

- **STORE** instructs the node to store a key–value pair that other nodes may retrieve later.

- **FIND_NODE** is used for asking the node to return a list containing the IP addresses, port numbers and node IDs of the k nodes from its routing table that are closest to a given node ID.

- **FIND_VALUE** is a request for the value associated with the given key. If the node has not stored any value with that key, it instead replies with the k closest nodes to the hash value of the key, equivalent to the response to `FIND_NODE`.

With each of the remote procedure calls, the sender attaches a random *RPC ID* which the responder must echo in the response. This helps the nodes to verify that the responses they receive originate from an actual request and are not forged.

5.2.4 The node lookup procedure

The *node lookup procedure* is how a node locates an ID in the network.

For two arbitrary IDs there is a 50% chance that their binary representation disagree on the most significant bit. This means that the uppermost k-bucket of a node's routing table covers half of the entire ID space; When the numbers of nodes and possible IDs are significantly greater than k, the chances that a node has stored a far away ID in the routing table, are therefore slim.

The closer two IDs are though, the better are the chances that a node knows the other ID. The node lookup procedure exploits this principle, and consists of the two following steps.

1) The node starts by searching the routing table for the α closest nodes to the ID it wants to locate. It then sends an asynchronous `FIND_NODE` call to each of the nodes it found in the routing table. α is a system-wide constant, for instance 3.

2) The node lookup procedure continues when it has received the responses to the `FIND_NODE` calls. Of all the returned nodes, the α closest ones are picked, and Step 2 is repeated until a total of k nodes have responded.

If the routing tables of the nodes in the network are properly populated, the `FIND_NODE` RPCs should always bring the node closer to the desired ID, because each new node returned has a more detailed knowledge about the IDs closer to it.

In step Step 2 a new round may be started before all the nodes have responded to `FIND_NODE`. If no nodes in one round answers to the RPCs, it is sent to the k closet nodes instead of the α closest nodes.

After the last repetition of Step 2, the node has probably received a rather large total of nodes. Each of these are attempted added to the routing table. The list of nodes may or may not contain a node with the exact ID that one was trying to locate. Either way, the k closest nodes to the ID is considered the result value of the procedure.

5.2.5 Joining a network

In order to join the network, a node must already have contact with one existing, connected node. Once that somehow has been obtained, joining the network is quite simple.

The node wanting to join chooses its own ID randomly, and adds one contact it knows to the routing table. The node then proceeds to to perform a node lookup procedure *for its own ID*.

During this lookup, the node will automatically fill up its routing table, and it will make itself known to the k nodes that it sends the `FIND_NODE` RPCs to.

After the node lookup for its own ID is complete, the node *refreshes* all its k-buckets that are further away than the node's closest neighbor. Refreshing a k-bucket means to pick a random ID that would be covered by that k-bucket and perform a node lookup on that ID.

5.3 ABS implementation

Simplification, see Section 5.3.1

we will focus on operations on the routing table (see **??**) and

Assumptions

In this implementation it will be assumed that all nodes are following the protocol and do not have malicious intents. There is thus no use for the random RPC IDs which were briefly covered in section 5.2.3.

It also will be assumed that nodes do not fail, and that messages arrive without problems. This enables a few simplifications and optimizations, but it will be made clear where the implementation differs from the original design.

Because all the nodes in the model will stay connected, there will not be any candidates for eviction from the routing tables. Instead new nodes will simply be rejected when the appropriate left-leaf k-bucket is full. Probing agents and deciding which ones to keep would only complicate the model and is not really relevant for the overall purpose of the algorithm.

Design

The key functionality of a Kademlia node is the node lookup. Node lookup proceeds in rounds. For each iteration, the node sends remote procedure calls to the closest nodes it has found, and must await their replies before continuing. This requires some degree of synchronization.

A challenge in itself is keeping an overview of all the different nodes that have been encountered during the lookup. The node will have to keep tabs on which nodes it has visited, what nodes it will visit next, and which nodes to ultimately keep when the lookup terminates. This, as well as other issues that will be uncovered later, will require a lot information to be stored during the lookup. Some redundancy is inevitable, and it will prove crucial to keep the different pieces of information up to date, across iterations and between the incoming replies.

Besides synchronization issues, correctly handling the nodes' routing tables is another challenge. The data structure will be realized as an algebraic data type, and all of the operations on it are implemented by pure functions. This design decision allows a very clean separation between the two major aspects of the implementation, and the routing tables can be tested independently. All the logic that is left inside the `Node` class is defining how the node behaves towards the rest of the swarm when sending and receiving remote procedure calls, and how its internal state changes over time as a result of this.

5.3.1 Basic data types

In this implementation the values stored in the DHT will be strings. No type-specific operations are ever performed on these, so changing the type alias `Value` to something else will not effect the model at all.

For simplicity's sake, we abstract away from the concrete implementation of keys. The keys would actually not be used for anything besides being passed to a hash function in the initial stage of the lookup procedure, and in a last equality check before a node would return the value it has stored for the key. Instead, all remote procedure calls and local methods will rather take parameters of a type `Hash`, which is an alias for `Int`. This represents the output of an imaginary hash function on keys.

A data type `Key` and a fitting hash function could have been provided in an extended implementation, but it is not significant to the actual workings of the DHT – the important functionality is the ability to locate nodes whose IDs are closest to the given hash[2].

The ID of a Kademlia node is in the same range as the hashes, so to make this semantically clear, the type `ID` is aliased to `Hash`. The ABS language has arbitrary length integers, so there is no real restriction, but IDs and hashes will be treated as if they contain the same fixed number of bits, and that they only hold non-negative numbers. The number of bits will not be hard-coded, but left as an option when the nodes are being initialized. Using full 160 bits IDs and hashes as in the Kademlia specification will unnecessarily slow down the model because the bitwise operators are recursive functions. Also, the length of the IDs dictates the depth of the routing table.

```
type Value = String;
type Hash = Int;
type ID = Hash;
```

A Kademlia node keeps the IP address, UDP port and Node ID for each of the nodes in its routing table. In the ABS model, a reference to an object of type `Node` (listing 5.1) replaces the IP address and UDP port as the means to contact another node, so only a Node–ID pair is needed as contact information. This coupling is merged into a record type named `Contact`.

```
data Contact = Contact(ID id, Node node);
```

[2]Another view on the choice of omitting the actual keys, could be that the keys in this particular implementation of the DHT are in fact of the same type as the hashes, and that the hashing function returns its input unaltered.

5.3.2 The `Node` interface

A basic interface `Node` is set up to only contain methods corresponding to the four RPCs defined in the Kademlia protocol, as well as methods for replying to these. Making sure that nodes in the model only will see each other as objects of this type, is limiting all communication between nodes to the use these RPCs. The `Node` interface therefore also works as a sort of formal specification of the protocol.

Listing 5.1: The Kademlia protocol expressed as an ABS interface

```
interface Node {
    Unit ping(Contact caller);
    Unit pong(Contact responder);

    Unit store(Hash key, Value v, Contact caller);

    Unit findNode(ID node, Contact caller);
    Unit findNodeReply(List<Contact> nodes, Contact responder);

    Unit findValue(Hash key, Contact caller);
    Unit findValueReply(Value v, Contact responder);
}
```

When responding to a `FIND_VALUE` RPC, a node will either return a list of nodes or the value of associated with the given key. In this implementation the node will use `findNodeReply()` to reply when it does not know the correct value. An alternative approach would be to let the first parameter to `findNodeReply()` be of type `Either<Value, List<Contact>>`.

As briefly covered in the introduction to this section, it is assumed that the nodes are cooperating with pure intents. This is why the methods in the interface does not include the extra parameter known as a RPC ID which is used to protect against phony replies. Under section 5.5 it is shown how this scheme can be incorporated into the implementation.

Not needing to echo the RPC ID, it is not clear how or if a node needs to reply to a `STORE` RPC. If the sender needs some acknowledgment that the `STORE` has been received, the `pong()` should suffice. As this model assumes non-faulty communication, the reply is not needed.

5.3.3 Extended node interface

Before starting on an implementing class, another interface is defined, extending `Node` with useful methods for interacting with the node beyond the four basic RPCs in the protocol. This interface includes methods for manipulating the DHT through tasks such as publishing values or finding values. It must also provide a method to tell the node to join a network through a bootstrap node.

To make matters more simple, the node will only be able to perform one lookup at a time. Therefore procedures that require a node lookup must wait until the node has successfully joined a swarm. For the first node that will be created, there is obviously no swarm to join, so a method `init` will tell a node to be ready right away.

A class correctly implementing the Kademlia protocol will repeatedly need to update its routing table, adding new nodes or moving existing ones around. This is a routine that will be invoked from many different places in the code and should be placed in a separate method for reuse. This method, called `insert`, is made visible through the interface so that it can tested thoroughly.

Listing 5.2: Extended Kademlia node interface

```
interface AdvancedNode extends Node {
    Unit join(Node node, ID id);
    Unit insert(Node node, ID id);
    Unit publish(Hash key, Value v);
    Maybe<Value> find(Hash key);
    Unit init();
}
```

5.3.4 Beginning of the `Node` class

For the reminder of section 5.3, the run-down of the node implementation will swing back and forth between logic put inside the `Node` class and operations on the routing table, presented in the order in which it is needed.

There are two constants defined in the Kademlia protocol: k is denotes the size of the buckets in the routing tables, the number of nodes that should be returned from a `FIND_NODE` RPC, and how many nodes to visit before terminating a node lookup; α is the *concurrency parameter*, determining the amount of simultaneous RPCs sent during each round of the lookup. For simplifying tests and examples, the implementation will allow an arbitrary range for the hashes and node IDs. For this purpose a system-wide parameter w is defined as the number of bits of a hash, meaning that valid hashes and IDs range from 0 inclusive to 2^w exclusive.

Listing 5.3: Header of the Node class

```
class Node(ID ownID, Int k, Int alpha, Int w)
implements AdvancedNode {
    Map<Hash, Value> storedValues = EmptyMap;
    Table contacts = EmptyTable;
```

Due to the lack of global variables, the logical system-wide constant must be kept locally in each instantiated node, so these are passed as class parameters. The node ID of will also be passed as a parameter, leaving the task of picking the ID to the process instantiating the node.

Two things essential for a node to work properly are a map for storing keys and values, and a routing table to keep contact information about other nodes. This concludes the very basic properties of the `Node` class. The next thing needed is the routing table data structure, `Table`.

5.3.5 Routing table data structure

The optimization with "relaxed" routing tables which also allows for the left k-buckets to be split, will not be used in this implementation. Dropping this

optimization is partly justified by the fact that the current backends of ABS can't support that many nodes running at a time; There will simply not be enough nodes to make it problematic that nodes might have to drop some of their closest encountered contacts – A manageable number of concurrent nodes lies between 100 and 1000. Even with low values for k a lookup for a value seems to succeed within two or three iterations, probably because each node is aware of a relatively large part of the swarm. More about the performance of the model will be discussed in section 5.4.

Also, being merely an optimization, the relaxed routing table scheme does not change the run of the algorithm in any way except that a node ends up keeping in touch with more contacts and may return a slightly different set of contacts upon a `FIND_NODE` request or the initial part of a node lookup. Once a set of contacts have been extracted from the routing table, the behavior stays the same.

As it will not have any branching in the left-side buckets, a Kademlia routing table without the relaxed optimization is practically a linked list. Still a custom data type that supports a tree structure will be used. Relaxed routing tables could then be implemented later if needed. Being an algebraic data type, the structure can easily be reused with new pure functions in addition to the pure functions that will be defined in this implementation.

In order to try to keep certain lines of code within a reasonable length, the data type representing the routing table will just be called `Table`. The `Table` data type will have a constructor for an empty table, similar to how `Nil` is a constructor for an empty list. As the `KBucket` constructor takes a list, which could in fact be empty, this may seem redundant. Despite this, a separate constructor for an empty table should make certain code snippets more readable and clear, for instance when investigating the contents of a routing table via the **`case`** statement and for stating the base cases in which recursive functions should halt. Listing 5.3 shows the `EmptyTable` constructor succeeding an instance of the `EmptyMap` constructor of the `Map` type, and the two certainly goes together better than, say, `EmptyTable` and `KBucket(Nil,0)`.

The briefly mentioned `KBucket` constructor is of course representing a k-bucket, which contains list of contacts. The third constructor of the `Table` type is `Split` which simply contains two other routing tables. No nodes are kept on a routing table of type `Split` itself, which makes it different from the `Cons` constructor of the list data type, and makes sure that the structure of the overall routing table is not restricted to be a linked list, even though it effectively will be in this particular implementation.

Both the `KBucket` and the `Split` constructors of the `Table` data type are augmented with an integer that denotes the number of contacts stored in that particular sub-tree. For the `KBucket` this number equals the length of the local contact list, while for `Split` it is the sum of contacts in the left and right sub-trees. Even though these numbers can easily be extracted with a recursive call down the tree and across each list, it is much faster to have this number cached in each subtree. It is of course important to keep these numbers correctly updated, and this will be taken properly care of during the various recursive operations performed on the routing tables.

Being able to look up the number of contacts in each sub-tree in constant time will also dramatically increase efficiency when performing a local lookup for the k

closest IDs to a given hash, as will be shown later.

Listing 5.4: Data structure for Kademlia routing tables

```
data Table = EmptyTable
           | KBucket(List<Contact> contacts, Int)
           | Split(Table, Table, Int);
```

Preserving the correctness of a routing table during operations such as insertion of contacts and splitting of k-buckets requires some knowledge of the system. An essential piece of information to the routing table is the ID of the node which it belongs to, hereafter called the *base ID* or simply denoted b. Every time a node is inserted into the tree, or when a k-bucket is split, it is the base ID that determines which k-bucket each contact will be put into. The base ID and the ID of the node to be inserted are compared bitwise by the longest common prefix, but this number cannot be decided without knowing w, which is actually the maximum value for this metric; Without an upper bound to the amount of bits in a number, the number can be considered to have an arbitrary amount of zero-bits in front of it, and comparing the prefix of two numbers would not make any sense. Once the appropriate bucket has been located, the parameter k is needed to determine whether there is room for a new contact in the bucket.

It has been covered that despite being system-wide constants, k and w have to be passed as parameter to each instantiated node. Likewise, a way to let these values, as well as the node-wide constant b, be known when operating on the routing table is needed.

Under this section there has already been presented arguments for keeping some redundant data in the routing table, namely the size of each sub-tree. However, storing k, w and b inside each sub-tree of the routing table seems excessive. Even more so since this information has already been stored in the node object itself (see listing 5.3) and the values are not going to change – unlike the number of stored contacts, which *was* decided to be kept in the routing table.

The alternative of coupling k, w and b to the routing table by storing it in a special root constructor is not much of a solution because the values would still have to be passed down the tree as parameters of recursive functions. The extra root node is then nothing but an extra layer complicating the model. Instead the recursive functions will be called directly from the node object which has all the parameters stored.

Also, if the root node were to be realized as a constructor of the `Table` type, there would for instance be no built-in, syntactical means to prevent a root node to be used as a parameter to a `Split` constructor, so extra logic would have to be introduced to solve such inconsistencies.

Because k, w and b are not stored in the routing table's data structure, these parameters will have to be passed to several of the functions that will be defined to operate on the routing table.

5.3.6 Comparing hashes

The longest common prefix and unique suffixes of two hashes

A central metric in Kademlia is the length of the longest common prefix of two hashes (figure 5.1). As mentioned in the previous section, this number is dependent on w, the number of bits in the hashes, since an positive integer can be regarded as having an arbitrarily long prefix of zeros. The simplest way to calculate the longest common prefix is probably to perform the opposite function, which will be called *the longest unique suffix*, and subtract it from w.

The search for the longest unique suffix terminates when the two integers are equal – only their common prefix remains. It shifts both the numbers one place to the right by dividing them by two, and calls itself recursively. This will always terminate because integer division rounds towards zero. The number of times the function was called recursively is the length of the suffixes that was removed from the integers in order to find two identical prefixes.

Listing 5.5: Longest Unique binary Suffix of two integers

```
def Int suffix(Int a, Int b) =
    if a == b then
        0
    else
        1 + suffix(a/2, b/2);
```

Xor function

Currently the ABS language does not offer any built-in bitwise operators. The metric for actual distance in Kademlia, XOR, must therefore be implemented manually.

During the implementation of this model a bug which caused the Java backend of the ABS compiler to crash, as well as another case of strange behavior, were encountered. The first version of the `xor` function was affected by this, which caused a number of problems. To circumvent these problems, another much slower `xor` function was written. These problems and their temporary solution, are described in section 5.4. The function described here does not cause any such problems.

Similar to `suffix`, the function `xor` calls itself recursively for each bit of the parameters. The least significant bit of the numbers are extracted by taking the value of the numbers modulo 2. The `xor` function can terminate when the two parameters are equal, as any integer's XOR of itself is zero.

Listing 5.6: The Bitwise Exclusive Or of two integers

```
def Int xor(Int a, Int b) =
    if a == b then
        0
    else
        let (Int x) = if a%2 == b%2 then 0 else 1 in
        x + 2 * xor(a/2, b/2);
```

5.3.7 Inserting contacts into the routing table

Because the implemented nodes do not fail, it was decided to disregard evicting nodes from the routing tables. In Kademlia the contacts within a *k*-bucket are sorted by the time they were last heard from. This is for deciding which nodes to probe and potentially kick out. When no nodes will be evicted from the *k*-bucket, the contacts do not need to be sorted in any particular order.

When inserting a contact, the contact needs to be sent down to the appropriate level recursively. For each level is passes, the counters in the `Split` constructors must be updated, but only if the contact is finally inserted. Also, while in the same chain of recursive calls, one or more *k*-buckets may need to be split up, and the existing nodes reassigned between the old and new *k*-buckets. To ease the implementation of the main insert function, several auxiliary functions are defined.

Inserting a contact to a *k*-bucket

For inserting contacts into a local *k*-bucket, two different functions are created. The first one, called `pushContact`, will simply push the contact to the head of the *k*-bucket's list without checking for duplicates or whether the list is full. The second function, `addContact` is slightly more sophisticated, only inserting the contact into the given *k*-bucket if the *k*-bucket is not already present, and only if there are less than *k* contacts in the bucket. These function is only meant to be used on a local *k*-bucket, not a full full routing table structure.

Listing 5.7: Helper functions on *k*-buckets

```
def Table initBucket(Contact c) =
    KBucket(Cons(c, Nil), 1);

def Table pushContact(Contact c, Table t) =
    case t {
        EmptyTable => initBucket(c);
        KBucket(list, n) => KBucket(Cons(c, list), n+1);
    };

def Table addContact(Table t, Contact c, Int k) =
    case t {
        EmptyTable => initBucket(c);
        KBucket(list, n) => if includes(list, c) || n >= k then
                                t
                            else
                                KBucket(Cons(c, list), n+1);
        _ => t;
    };
```

Splitting *k*-buckets

When a full right-leaf *k*-bucket is encountered, it must be split before the contact can be inserted. There will be two functions dedicated to the task of splitting a

k-bucket. The function that is called first is called `split`. This function is simply for preparing the values that is sent to the second function, `split2`, which does the actual work.

The `split` function is called with a parameter `d` which is the depth in the routing table at which the *k*-bucket is located. The parameter `w` is number of bits in the IDs, and `b` is the base ID; These are passed on to `split2` and there used to determine which of the two new *k*-bucket that each contact should be reassigned to.

The `split2` function moves each contact from the given list into one of two *k*-buckets which is built up during the recursive steps. The *k*-buckets are temporarily stored as parameters to the recursive calls to the function, and they are finally placed in a `Split` constructor and returned when the end of the contact list is reached.

The longest common prefix of the contact in question and the base ID is calculated as described on page 75. If the longest common prefix is equal to the current depth `d`, the contact should remain on this level of the routing table, and is inserted into the left-hand *k*-bucket parameter as the function is called recursively. Otherwise it is inserted on the right-hand side.

The listing below also includes the use of `tSize`, which is a simple function that extracts the integer that are stored in a `KBucket` or `Split`, or returns 0 for `EmptyTable`.

Listing 5.8: Splitting a *k*-bucket

```
def Table split(Table t, Int d, Int w, Hash b) =
    split2(contacts(t), d, w, b, EmptyTable, EmptyTable);

def Table split2(List<Contact> q,
        Int d, Int w, Hash b, Table l, Table r) =
    case q {
        Nil => Split(l, r, tSize(l)+tSize(r));
        Cons(head, tail) => if w-suffix(id(head),b) == d
          then
            split2(tail, d, w, b, pushContact(head, l), r)
          else
            split2(tail, d, w, b, l, pushContact(head, r));
    };
```

The main insertion function

The function which will be used to initiate the whole process of inserting a contact into a routing table, is `insertContact`. Like with the splitting of the *k*-buckets, the main work is handled by another function that that is called by the first.

In `insertContact` the longest common prefix of the to-be-inserted contact `c` and the base ID `b`, is calculated. The result equals the appropriate depth in the routing table for `c` to be inserted, which will be called the *target depth*. The next function, `insert_` is then called with all the parameters of `insertContact`, plus

the target depth `dC`, and a counter variable `d` that keeps track of the current depth, starting at 0.

The `insert_` method takes all possibilities into consideration when it traverses the given routing table. Whenever it encounters a `Split` constructor, it checks with the current depth `d` against the target depth `dC` whether the contact should be inserted here or that recursion should continue.

When a non-empty k-bucket in encountered, it is first checked whether the contact already is in the list with the simple function `includes`. If the contact already is stored, the function terminates. If not, wither the contact is pushed into the k-bucket, or the k-bucket is split and the function called again.

Each time the function calls itself, it checks the return value afterwards and updates the counter on the current k-bucket appropriately.

Listing 5.9: Inserting a contact in the routing table

```
def Table insertContact(Table t, Contact c,
                        Int k, Int w, ID b) =
  let (Int depth) = w - suffix(b, id(c)) in
  insert_(t, c, k, w, b, depth, 0);

def Table insert_(Table t, Contact c,
                  Int k, Int w, ID b, Int dC, Int d) =
  case t {
    Split(left, right, _) =>
     if d == dC then
       let (Table l) = addContact(left, c, k) in
       Split(l, right, tSize(l)+tSize(right))
     else
       let (Table r) = insert_(right, c, k, w, b, dC, d+1) in
       Split(left, r, tSize(left)+tSize(r));

    KBucket(list, n) => if includes(list, c) then
                          t
                        else if n < k then
                          pushContact(c, t)
                        else
                          let (Table s) = split(t, d, w, b) in
                          insert_(s, c, k, w, b, dC, d);

    EmptyTable => initBucket(c);
  };
```

5.3.8 Searching through the routing table

The function `findContacts` is the implementation of the lookup scheme described on page 67. It takes a routing table and a node ID and returns the n closest contacts to the ID. The function calls itself recursively and picks out contacts from multiple k-buckets if needed.

But to make is a bit simpler, it is assumed that at the initial call has an `n` parameter equal to k, and the returned list will not be sorted. Therefore, if the k-bucket corresponding to the given ID is full, the list of contacts can be returned without further ado.

Because each level of the routing table data structure keeps the number of remaining nodes in the sub-tables, it can be easily detected that a sub-table contains fewer contacts than what has been requested. Thereby, extra nodes can be appended to the returned list of contacts before the next recursive function call is made. The final list that is returned by the function `findContacts` is therefore built up during only one pass down the deeps of the routing table.

The k-bucket which is closest to the requested ID is prioritized so that as many contacts as possible is chosen from that bucket. Because `n` is assumed to be equal k, *all* the nodes from the bucket which is closest to the target ID, and which contains any contacts, is chosen. Only if this bucket is not full, will other k-buckets be used.

When selecting nodes from other k-buckets, the ones located in the deepest levels of the routing table will be picked over the ones near the top. For a given k-bucket the IDs of the contacts in the bucket above itself are one order of magnitude further away from its own contacts, than the contacts of the k-bucket below.

As said, the list of returned contacts will not be sorted, and when taking out only a subset of the contacts of one k-bucket, the contacts' distance to the target ID is not considered – the function simply takes the first ones in the list. This simplification is justified by the fact that if not all the contacts are taken from a k-bucket, it means that the k-bucket is not the closest bucket to the target ID anyway; The contacts are then at least one order of magnitude further away than other contacts that have already been selected.

The announced `findContacts` function will actually just take some parameters from the calling node and do a quick calculation before passing on some values to the function that will do the actual work. It has been explained what criteria the contacts are chosen on, and that the contacts that are returned are not compared directly against the target ID: When it has been decided which k-bucket that ideally would cover the target ID, the contacts from this k-bucket have priority, then the k-buckets below it, and lastly k-buckets above.

Given the target ID, the base ID b, and w which is the number of bits in the IDs *and* the deepest possible level of the routing table, the appropriate k-bucket can be located. The closer the target ID is to the base ID, the deeper down it belongs in the routing table. The length of the longest common prefix of the two IDs is the same as the depth of the target IDs k-bucket. The length of the common prefix is found by subtracting the length of the longest unique suffix (Listing 5.5 on page 75) from w. This depth, along with the desired number of contacts and a reference to the routing table, is passed on to the function that does most of the work,

```
def List<Contact> findContacts(Table t, ID target, Int n,
                               Int w, ID b) =
    let (Int depth) = w - suffix(target, b) in
    getContacts(t, n, depth);
```

With the depth of the target ID's location `d`, the function `getContacts` traverses the routing table down to this k-bucket. It is not really known to the function what

the current depth is, `d` simply shows how many levels there are left. Some default tests are done before the main logic of the function so that the recursion will halt given an empty routing table or if `n` is zero.

If the current routing table `t` is a *k*-bucket, the recursion has reached the bottom of the routing table. In that case the case branch extracts as many contacts as it can. `keep` is a helper function that was defined for taking the `n` first elements of a list.

The recursive step of the function is the case branch matching a `Split` constructor. A helper variable `m` holds the number of contacts that will be extracted from the current level of the routing table: If `d` is zero, we are at the ideal depth, and take and many contacts as possible. If the right height has not yet been reached, but the lower routing table does not contain enough contacts, then the appropriate amount of contacts to take from the current is calculated too.

```
def List<Contact> getContacts(Table t, Int n, Int d) =
    if n > 0 then
        case t {
            EmptyTable => Nil;
            KBucket(list,k) => if n >= k then list
                               else keep(list, n);
            Split(l, r, _) =>
                let (Int m) =
                    if d == 0 then
                        tSize(l)
                    else if tSize(r) < n then
                        intMin(n-tSize(r), tSize(l))
                    else
                        0
                in
                let (List<Contact> ll) =
                    getContacts(l,m,0)
                in
                  concatenate(ll, getContacts(r, n-m, d-1));
        }
    else
        Nil;
```

5.3.9 The node lookup procedure

This section covers the implementation of the node lookup procedure, which is the key functionality of the Kademlia node. The method is so long that not all of it will be shown, only the key parts.

The method is made so that it can be used for both finding a list of nodes, and to search for a given value in the DHT. The method returns *either* a value or a list of contacts, which is apparent by its return type: `Either<Maybe<Value>, List<Contact>>`.

A boolean parameter `findValue` is used to signify whether one want to locate a value on the DHT, or find a list of nodes. Either way, the parameter `hash` is the value that will be searched for; In this model the hashes are also used for keys.

Synchronization

In the `Node` interface there were methods defined for the all responses to the remote procedure calls in the Kademlia protocol, as well as methods for the RPCs themselves. Using separate methods for replies, instead of having the value returned directly, leaves the node some freedom. The system-wide parameters k and α are not hard-coded, so it is not known how many concurrent RPCs a node will send at a time. In a real system, the number of concurrent RPCs may also vary for each iteration of the lookup, depending on how quickly remote nodes manage to respond.

Not knowing the exact number of concurrent asynchronous method calls that will be made, it is not possible to set up a correct guard for an **`await`** statement. Cf. line 7 of listing 3.6 on page 25, showing an **`await`** statement for two futures. In the case of the Kademlia node, there is no fixed number of futures. It is not possible to await a variable-length list of futures in one statement, so another type of condition is needed.

The `Node` class is given an integer field `pendingReplies` which is initially 0 and increased once for each outgoing asynchronous method call that is made. After the method calls are sent, the `nodeLookup` method will suspend and wait for `pendingReplies` to reach 0 again. Each call that is made back to the node's `findNodeReply` method will decrease the counter by one.

Main loop

The logic above is wrapped inside a while loop which continues until k nodes in total have responded, or as many nodes as has been encountered if that number is less than k. The number of nodes encountered is updated for each incoming reply.

Similarly, the concurrency parameter α decides how many concurrent RPCs to send for each iteration, but a smaller number is used when less than α nodes are known. Sending the concurrent asynchronous method calls is done in an inner loop. A field `queue` holds a list of contacts, sorted by their proximity to the *target hash* – the hash of the node or key that is being looked up in the lookup procedure. After each call that is sent, the callee is added to a set of visited nodes, the field `tried`, so that the same node is not contacted twice in the same node lookup.

The `queue` is initially set to the k closest nodes to the target hash, found using the `findContacts` function defined in Section 5.3.8, before the node lookup begins. The queue is updated for every contact that is reported back to the node between each iteration of the main loop in `nodeLookup`, unless a reported contact is found in the `tried` set.

Among all of the logic covered above, is also an adjustment for when the method is used for finding a value instead of a list of contacts. First of all, if the `valueLookup` parameter is true, then the asynchronous method calls being sent are for `findValue()` rather than `findNode()`. Also, the main while loop has an additional condition that makes it terminate if a field `valueFound` contains a value. The value is set if a remote node has responded with the `findValueReply` method.

The key part of the `nodeLookup` method is found in Listing 5.10 on the next page. To differentiate between local variable and class field among the references that has not been explicitly listed, the fields are prefixed with "**`this`**.".

Listing 5.10: The main logic of the lookup procedure

```
Int nContacts = tSize(this.contacts);
Int kk = min(k, nContacts);
Int nRPCs = min(alpha, nContacts);

Int responses = 0;

while (responses < kk && this.valueFound == Nothing) {
    Int i = 0;

    while (i < nRPCs && this.queue != Nil) {
        Contact c = head(this.queue);
        Node node = node(c);

        this.tried = insertElement(this.tried, c);
        this.pendingReplies = pendingReplies + 1;

        if (findValue) {
            node!findValue(hash,Contact(this.ownID, this));

        } else {
            node!findNode(hash, Contact(this.ownID, this));
        }

        i = i + 1;
        this.queue = tail(this.queue);
    }

    await this.pendingReplies == 0 || this.valueFound != Nothing;

    // Updating kk and nRPCs as if timeouts were possible
    responses = responses + nRPCs - this.pendingReplies;
    kk = min(this.k, kk + length(this.queue));

    nRPCs = if this.pendingReplies == nRPCs then
                kk       // whole round failed – try more!
            else
                alpha;
}
```

The value of the local variable `kk` is the lowest of k and the total amount of contacts, and affects the termination of the main loop. Likewise `nRPCs` is the minimum of α and the total number of contacts, and dictates how many asynchronous method calls that will be sent each round. Both `kk` and `nRPCs` are updated each round of the loop.

Accounting for nodes not responding

The await statement shown on line 28 of the listing only depends on `pendingReplies` or `valueFound`, so if a node would not respond, the method would never continue. Had the Java backend supported timers however, there could have been added an time requirement on the await guard.

The nodes in this model are always responding so there is of course no problem, but the values of `kk` and `nRPCs` are nevertheless updated in a way that assumes that the process could have awoken from the await statement without `pendingReplies` being 0. The specification of the node lookup procedure states that a node should proceed to send the RPCs to k nodes, rather than α, in the next round if no nodes seem to be responding. It must also be considered that the responding nodes may not know as many as k other nodes, and thus reply with fewer than that. The queue of contacts is thereby not necessarily long enough to send as many RPCs as wanted.

5.4 Performance of the implementation

Because the Kademlia protocol is made to support an almost arbitrarily large number of nodes, it would be interesting to see how many nodes it would be possible to run in the ABS model.

An early version of the implementation had several functions that were hastily implemented as placeholders for functions that would come later. The first time it was attempted to run the model on the Java backend, the program used almost an hour to terminate when 10 nodes were set up to join each other and create a swarm.

Gradual changes were made which made the model slightly faster, until a swarm of a hundred nodes could be set up and run reasonably fast. However, the most significant speed improvement was made by a changing a function that had been completely overlooked – the `xor` function, further described in Section 5.4.1.

The rudimentary testing was done by initializing one node that would act as the bootstrap node. Then, a loop would create a certain number of node objects and have them join the bootstrap node by calling the `join` method which was defined in the `AdvancedNode` sub-interface.

The `AdvancedNode` interface also defined a method `printContacts` that would pretty-print the structure of a node's routing table. It would seem that, because all the nodes were joining more or less simultaneously, the `findNode` method calls would return shorter lists of contacts, because each node had not yet completed the node lookup.

The tests were performed on the same, reasonably modern computer, and in the record run it took 53 minutes for the program to terminate when 10 000 nodes were joining the swarm.

5.4.1 The `xor` function

Before arriving at the current implementing of the `xor` function in listing 5.6, various other versions were attempted. These versions were affected by a bug in the Java backend of the ABS compiler as well as some unexpected behavior.

The bug which causes the compiler to crash appears in some circumstances where numbers of type `Rat` are mixed in with numbers of type `Int` in the same expression[3].

Due to the rather incomprehensible output from the compiler, it was not at first determined which exact fragment of the code that caused it to crash. The wrongfully suspected offender was the part where each of the arguments was divided by two in the recursive call to the function itself. The suspicion was based on the assumption that this division operation was the only place where the integers could be turned into rationals.

Since the `suffix` function in listing 5.5 was using integer division but somehow did not cause any trouble, an alternative version of the `xor` function was written, utilizing the `suffix` function and completely avoiding further use of division and thus rational numbers.

The elaborate scheme was to use several helper functions to extract the value of the bits in each of the positions in the two original parameters. The `xor` function would first use `suffix` to determine the index of the most significant bit that would need to be checked. This is sound because the XOR of any common prefix would turn into a prefix of only zeros, so only the part where the integers start to disagree is interesting. The helper function would then work its way towards the least significant bit by decrementing the index.

The helper function would extract the bits from each index by using the modulo of some power of 2, and each time using the same procedure to mask out other bits. The whole arrangement resulted in four functions being called by each other, each one also calling itself recursively. In retrospect it is quite obvious that this scheme could not be very efficient, but it did work as intended and avoided crashing the compiler. It also remains an interesting display of arithmetics. The code in its full can be seen in section E.1 in the appendices.

The other unexpected behavior that caused some confusion is that parentheses in pure expressions do not always seem to actually be evaluated. The ABS reference manual does not mention usage of parentheses, but if they are used in an expression, it will go through the compiler without any warnings or errors. It is therefore easy to think that the parentheses have some purpose here, and that for instance an expression like `1 * (2 + 3)` would return `6`. However, that expression returns `5`, which is what one would expect if the parentheses were not there.

5.5 Potential improvements of the model

A minor, deliberate deviation from Maymounkov and Mazières [19]'s specification is that the model does not take into account the RPC IDs that is supposed to sent

[3]The bug was reported on https://envisage.ifi.uio.no:8080/redmine/issues/140 – last accessed 2014-03-30.

with each RPC. It has rather been assumed that all nodes have pure intents and that all messages passed between nodes are genuine.

Implementing the scheme using RPC IDs would not have been very complicated, but would have added another layer of complexity which is not strictly relevant to the protocol. A brief run-down of an implementation is as follows.

Algorithm for the *RPC ID* scheme

Augment all methods in the `Node` interface as well as any implementing class with an additional parameter `rpdID`, for instance like so:

```
Unit ping(Contact caller, Hash rpcID);
```

Let the implementing class keep a mapping from hashes to contacts. For each new round of outgoing RPCs and for each node the RPC is sent to, create a random and unique hash and store it in the mapping along with the node in question. This algorithm can be expressed in the following method, where `this.rpcIDs` is a mapping with type `Map<Hash,Contact>`.

```
Hash newRpcID(Contact contact) {
    Hash hash = random(pow(2, this.hashLength));

    while (lookup(this.rpcIDs, hash) != Nothing)
        hash = random(pow(2, this.hashLength));

    this.rpcIDs = InsertAssoc(Pair(hash,contact), this.rpcIDs);
    return hash;
}
```

Chapter 6

BitTorrent

BitTorrent is a peer-to-peer protocol introduced by Bram Cohen in 2003 [6]. It is mainly used for distributing and sharing files over the Internet. Unlike downloading files with the traditional server–client approach, files are not only sent from its original source to the users, but also spread among the different users. This potentially allows for fast downloads and greatly reduced server traffic.

This chapter covers an implementation of the *rarest piece* algorithm which is used to maximize the replication of a file in a BitTorrent network.

6.1 BitTorrent Overview

Users sharing files over BitTorrent are called *peers*. A set of peers that are sharing the same file is a *swarm*. To find a swarm that shares a file that a user is interested in, the user must contact a *tracker* – a server which keeps track of the peers – or with tracker-less torrents it can lookup the list of peers in a Kademlia distributed hash table. The torrent also contains meta-information about the file.

A file that is shared in BitTorrent is divided into many small pieces. Among the file's meta information are hashes of each piece that is used to verify that the received data is correct. When a peer has successfully downloaded one piece, it can share this piece with other peers, and thus participate in the distribution of the file even though it does not have downloaded all of it.

A peer has the whole and continues to share is known as a *seed*. While the peer is still trying to download the file, it is called a *leecher*. When an aspect of the BitTorrent protocol is explained from one peer's perspective, this one peer will be referred to as the *local peer*, and any other peer than the local peer, is called a *remote peer*.

The *rarest first* algorithm in the BitTorrent protocol aims to maximize the number of *distributed copies* of the file across the swarm. That there is a distributed copy of a file does not necessarily mean that there is a single seed, but instead that each piece of the file is available from some peer in the swarm [20]. In general, the rarest first algorithm is simply to choose to download the rarest pieces of a file first.

Each piece is divided into *blocks*. Each block is downloaded from just one peer, but multiple blocks of the same piece can be downloaded from different peers. The *strict priority policy* states that a peer should try to complete the pieces that it has

already started downloading before any requesting blocks from other pieces.

The four first pieces that a peer downloads are chosen at random in what is known as the *random first policy*. Selecting initial pieces that there may exist more copies of, improves the peers chances to receive those pieces quickly, so that it has some pieces to share with others.

BitTorrent uses an algorithm called the *choke algorithm* as an incentive for peers to shares files. Peers with greater upload rates are also prioritized for downloads; Peers only shares its pieces with a certain number of peers at a time, and the choke algorithm is a way to reward the most generous uploaders. That the local peer *chokes* a remote peer means that the remote peer (temporarily) is not allowed to download from the local peer.

The statistics of the upload and download rates of different peers and the precise mechanics of the choke algorithm will not be included in the ABS model. Instead choking and unchoking will be done somewhat randomly.

6.2 BitTorrent peer protocol

There are 11 different remote procedure calls defined in the BitTorrent protocol [7, 17].

Handshake This the initial contact between two peers. When the first peer has sent a handshake to the other, and the other has sent one in return, a connection has been established.

Bitfield After the two peers have been formally introduced, they tell each other about the pieces of file that they already have. This message is called the bitfield.

Interested If a remote peer *B* has a piece of the file which peer *A* does not have, then *A* is said to be *interested* in *B*. Peer *A* can let peer *B* know this with the `interested` RPC.

Not interested If peer *A* has all the pieces that peer *B* has, then *A* informs *B* of this with the `not interested` RPC.

Choke `choke` is sent to a peer to inform it that it has been *choked* by the caller. When peer *B* is choking peer *A*, it means that *A* will not get to download any pieces from *B* until *A* has been unchoked by *B*.

Unchoke *Unchoked* is obviously the opposite of *choked*, and an `unchoke` RPC is sent to a peer to inform it that it has been unchoked and is now allowed to download from the caller.

Request A request is sent to a peer to tell it that the caller wants the callee to send a given block.

Piece When a peer receives a request from a remote peer that it has currently unchoked, it will send the requested *block* of the file with the `piece` RPC.

Cancel This RPC is used to tell a peer that the caller no longer wants it to send a previously requested piece.

In the following it will be explained in more detail how the protocol defined above works in practice. The ABS model will be explained along the way, as a another way to express the ideas behind the protocol and how the peers ensure the replication of the file.

6.3 ABS implementation

This ABS implementation will only model the communication between the peers; There is no need to have an actual file being sent between the different objects. How the peers join the swarm is also less relevant to the Rarest First algorithm, so the peers will just be given references to each other when the objects are initialized. The purpose of this model is to show how the BitTorrent protocol potentially increases download speeds and maximizes the replication of a file by splitting it into pieces.

6.3.1 Data types

Even though the peers will not send actual files between each other, some data is needed to indicate the progress of the downloads. All that is really needed is a boolean value for each of the blocks that makes up a file. One type synonym is defined for the blocks that are grouped into pieces, and another for the list of pieces that comprise the file.

When a peer receives information about other peers' progress, through the RPCs `bitfield` and `have`, this information must be stored locally so that the peer can estimate the what pieces that are the rarest. For this purpose a set of peers – a *peer set* [17] – is stored for each of the pieces of the file. Each peer set is then stored in a list that will the be called the *piece set*.

Yet another type is defined for the bitfield that is sent between the peers after their handshake session. The bitfield is, like the `Piece`, also a list of boolean values. Different type synonyms helps to clarify the purpose of the various lists. It is important that the length of a bitfield is always the same as the number of pieces in the file, and the length of a `Piece` type is always the same as the number of block pr piece. These numbers will be passed to the peers upon their creation.

```
type Piece = List<Bool>;
type File = List<Piece>;
type PeerSet = Set<Peer>;
type PieceSet = List<PeerSet>;
type Bitfield = List<Bool>;
```

For each of the remote peers that it is connected to, a local peer remember the *choked* and *interested* states, both locally and remotely. These four flags, as

well as the respective peer's bitfield, is kept in record called `PeerInfo`. Chok*ing* and interest*ing* denotes whether the remote peer is choking, or is interesting to, the local peer, while chok*ed* and interest*ed* tells whether the remote peer is choked by, or interested in, the local peer.

```
data PeerInfo =
    Info(Bitfield bf,
         Bool choking, Bool interesting, // remote -> local
         Bool choked, Bool interested);  // local -> remote
```

The last data type definition is a simple enumeration of the different states of a local peer.

```
data State = Init | Leecher | Endgame | Seed;
```

The peer is the only type of actor in this BitTorrent model, and a `Peer` interface is defined with a method for each of the remote procedure calls defined in the protocol. Each method requires the peer to identify itself with the caller parameter. The `handshake` and `bitfield` are given an extra boolean parameter which indicates whether the method call in question was the initial one or the reply to it. This avoids having the peers sending handshakes and bitfields back and forth forever, without needing to store this the state permanently.

```
interface Peer {
    Unit handshake(Peer caller, Bool init);
    Unit bitfield(Peer caller, Bool init, Bitfield have);
    Unit interested(Peer caller);
    Unit uninterested(Peer caller);
    Unit choke(Peer caller);
    Unit unchoke(Peer caller);
    Unit request(Peer caller, Int blockID);
    Unit piece(Peer caller, Int blockID);
    Unit have(Peer caller, Int pieceNum);
}
```

6.3.2 Defined functions

The peers store various information about each other in the form of different algebraic data types, and for handling this information, a set of functions is defined.

Keeping track of pieces available in the swarm

The first piece of information about a remote peer comes in the bitfield exchange after two peers have completed the handshakes with each other. The bitfield is a list of boolean values where each value signifies whether the corresponding peer has the piece of the file with piece ID corresponding the index of the boolean value.

The function `addBitfield` takes the local peer's currently known piece set, and adds the remote peer to the set of peers corresponding to to each of the pieces that the remote peer reported to have.

```
def PieceSet addBitfield(PieceSet ps, Peer p, Bitfield have) =
    case ps {
    Nil => Nil;
    Cons(set, rest) => let (Set<Peer> peers) =
                    if head(have) then
                       insertElement(set, p)
                    else
                       set
                    in Cons(peers,
                            addBitfield(rest,p,tail(have)));
    };
```

The bitfields are only sent between to peers after their handshakes. Subsequent reports about completed pieces are sent individually with the `have` RPC. The local peer adds the remote peer to its list of known holders of the piece with the `addPeer` function.

The sets of peers are listed so that the index of the set corresponds to the piece's ID. The function traverses the list and uses the given piece ID to count down to the correct set.

```
def PieceSet addPeer(PieceSet known, Peer p, Int piece) =
    case known {
        Cons(s, rest) => if piece == then
                            Cons(insertElement(s, p), rest)
                         else
                            Cons(s, addPeer(rest, p, piece-1));
        Nil => Nil;
    };
```

6.3.3 Selecting a piece to download from a remote peer

The `Peer` class has a method called `getFrom`, listed on the following page, which ties together the class' various fields and the functions defined on the algebraic data types that the fields are comprised of.

The method takes a single argument, a reference to a remote peer. All the information that the local peer knows about this other peer is stored in the class fields. From this information the local peer will decide whether the remote peer has any interesting pieces and if so try to download one of them.

The information stored about the remote peer is retrieved from the peer set. Then a list of prioritized pieces is collected: The function `priority` returns the indices of every piece of which the peer has one or more block, but has not completely downloaded. These pieces are prioritized for download according to the *strict priority policy*. The function `findStrict` checks if, according to the information that is stored locally about the remote peer, any of the strictly prioritized pieces are in the remote peer's bitfield. If so, one of these pieces is selected for download.

Listing 6.1: Selecting a piece for download from a remote peer

```
Unit getFrom(Peer peer) {
    PeerInfo info = lookupUnsafe(peers, peer);
    Bool interesting = interesting(info);

    List<Int> priority = priority(downloaded, 0);
    Maybe<Int> piece = findStrict(priority, bf(info));

    if (piece == Nothing) {
        List<Int> want = want(bf(info), have, 0);

        if (state == Leecher) {
            want = rarest(want, available, Nil, -1, 0);
        }

        if (want != Nil) {
            Int rand = random(length(want));
            piece = Just(nth(want, rand));
        }
    }

    if (piece != Nothing) {
        interesting = True;

        if (~choking(info)) {
            Int i = fromJust(piece);
            Int b = index(nth(downloaded, i), False);
            Int block = i*blocks + b;

            peer!request(this, block);
            // should also keep track of outgoing requests

        } else {
            peer!interested(this);
        }

    } else {
        interesting = False;
        peer!uninterested(this);
    }

    PeerInfo update = case info {
        Info(bf, x, _, y, z) => Info(bf, x, interesting, y, z);
    };
    peers = put(peers, peer, update);

}
```

If no piece was selected in the previous step, a function `want` cross references the remote peer's bitfield with the bitfield of the local peer, and returns all pieces that the remote has but the local does not. If the local peer is in the normal state in which it tries to download the rarest pieces, it filters the list of piece IDs with the function `rarest`. Finally, a random piece from the `want` list is selected for download.

If a piece was found that could be downloaded from the remote peer, it means that the local peer is interested in the remote. If the local peer is choked by the remote, it will send a message telling that it is interested. If however the local peer is currently unchoked by the remote, it can proceed to request a block.

First the local peer must find which block to get. The function `index` is defined to return the index of the first occurrence of a value in a list. First the piece is retrieved from the field `downloaded`. The piece is represented by a list of booleans (blocks), so `index` is used to find the first occurrence of `False`, meaning the first block that has not been downloaded in the piece. The block's ID is calculated from the piece's index in the file and the block's index within the piece.

Lastly the local peer updates the *interesting* state of the remote peer and stores it.

6.3.4 Handling requests

The last code listing from the peer class shows the `request` method. In broad terms, the method is called by a remote peer that wants a certain block, and the local peer uploads this block to the remote through the `piece` RPC. However, the local peer will only grant requests from remote peers that are part of the *active peer set* – the set of peers that are unchoked [17, p.2]. Also, it will only send one block at a time: If a remote peer sends multiple requests at once, the requests must be put in a queue.

In the ABS model, the *choked* status of a remote peer is kept in a field, `peers`, which is a mapping from peer references to `PeerInfo` data types. Having a separate field for the *active peer set* is therefore somewhat redundant. However, the active peer set has its use as the place to store the queues of requests from each remote peer: The field `active` is a mapping from peer objects to lists of integers, where the integers are block identification numbers, and each list represents the corresponding peer's request queue.

When a remote peer becomes choked, it is removed from the active peer set, and the queue or requests with it.

The `request` method first checks the `PeerInfo` associated with the calling peer to see if it is currently unchoked. If not, the request is ignored. If proceeding, the ID of the current requested block is appended to the list associated with the caller. The method then waits until that block is in the head of the queue. While waiting, the caller and its queue may however be removed from the active peer set. This calls for a rather complicated await statement, which can be seen in the listing below.

Briefly, the await statement checks if the remote peer has been choked, *or* if the block requested in the current process is in the head of the remote peer's queue.

What makes it complicated, however, is that it turns out that the logical *or* and *and* operators in the ABS language do not short circuit the expression – both sides

Listing 6.2: Handling and enqueuing requests

```
Unit request(Peer caller, Int blockID) {
  if (~choked(lookupUnsafe(peers, caller))) {
    List<Int> q = lookupDefault(active, caller, Nil);

    active = put(active, caller, appendright(q, blockID));

    await choked(lookupUnsafe(peers, caller))
          || let (List<Int> l) =
                  lookupDefault(active,caller,Nil) in
             if isEmpty(l) then True else head(l) == blockID;

    q = lookupDefault(active, caller, Nil);

    if (~isEmpty(q)) {
        await caller!piece(this, blockID);
        active = put(active, caller, tail(q));
    }
  }
}
```

of the `||` are evaluated.

When looking up the remote peer's request queue, it cannot be assumed that the queue exist despite the check that the peer has not been choked. Therefore the safe `lookupDefault` function is used. That in order makes it unsafe to use the accessor function `head` on the looked-up value. Luckily, the if-then-else expression can be used to ensure that `head` is not called on `Nil`.

Another possibility would be to use a populated list as the default value for the lookup so that it was safe to use with `head`, but with a value that could not occur as a block ID:

```
await choked(lookupUnsafe(peers, caller))
      || let (List<Int> l) =
              lookupDefault(active, caller, list[-1]) in
         head(l) == blockID;
```

After the await statement, a fresh copy of the request queue is once again looked up from the active peer set. If the queue is not empty at that point of time it means that the peer is still unchoked and that the current request is indeed first in the line.

The call back to the remote peer contains a reference to the local peer, and the requested block's ID so that the remote peer can cross the off the correct block ass downloaded. However, no actual content is transfered.

6.4 Test run

The following test is set up with three peers. The first one is given the status `Seed`, meaning that it will pretend to have all the pieces of the file, while the two others start as leechers with no pieces, signified by the `Init` status. Each peer is set up with a list containing the previously initialized peers, and will start the handshake with all of them. The number of pieces the file is split into, and the numbers of block pr piece, is set to 5 and 2 respectively.

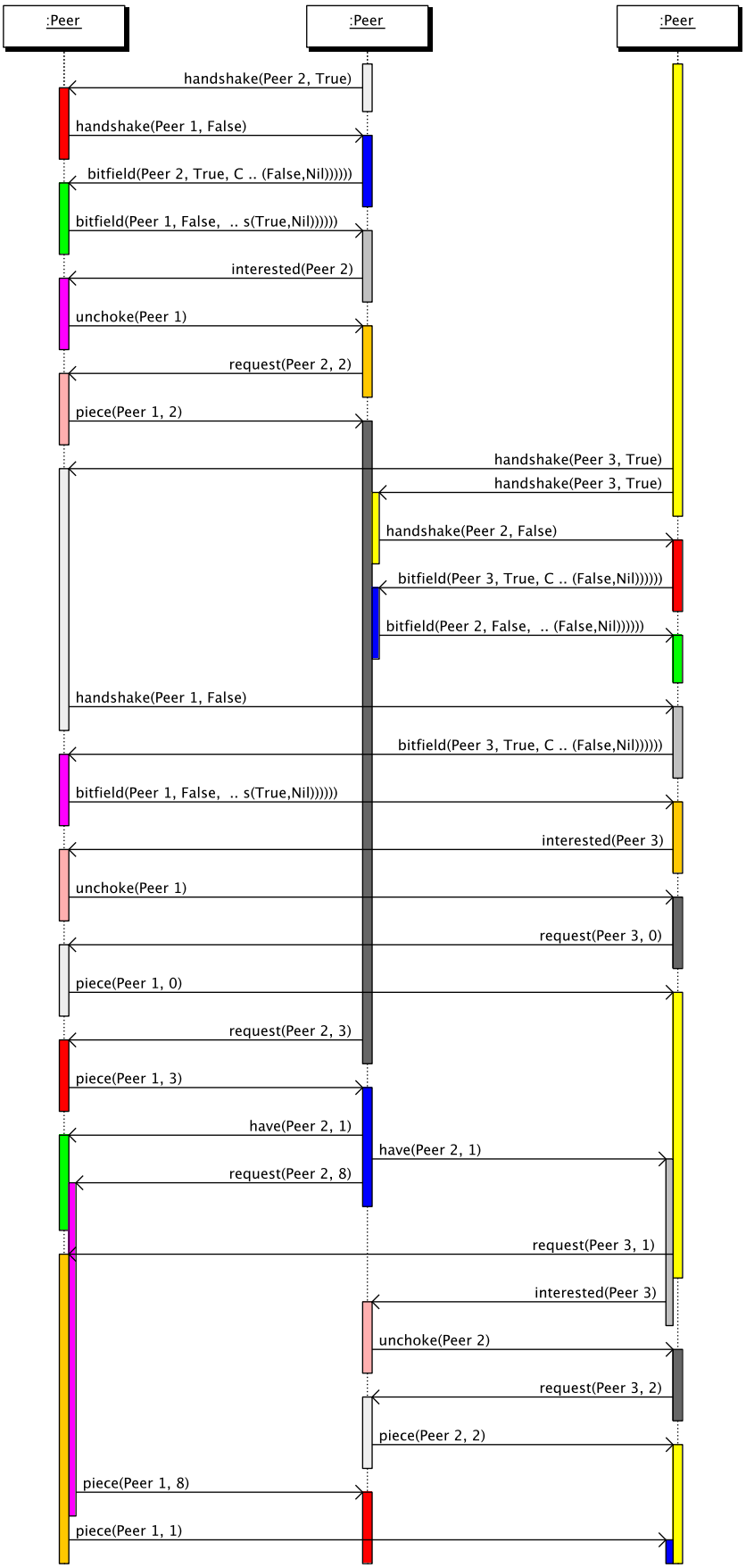
```
module Test;
import * from BitTorrent;

{
  Int pieces = 5;
  Int blocks = 2;

  Peer seed = new Peer(Seed, Nil, pieces, blocks);
  Peer peer1 = new Peer(Init, list[seed], pieces, blocks);
  Peer peer2 = new Peer(Init, list[seed,peer1], pieces, blocks);
}
```

Figure 6.1 shows an example run of the setup. The seed remains passive, and the first leecher initiates the handshake, and then the bitfield. It finds that the seed has pieces that itself does not, so it tells the seed that it is interested. The seed unchokes the first leecher, with proceeds to request and download a piece.

As the piece is being sent to the first leecher, the second leecher starts the handshakes with the two other peers. The two leechers do not have any completed pieces, so they are not interested in each other, but the new leecher is also interested in the seed.

The diagram continues to show that the first leecher completes a piece, and announces this with the `have` RPCs. The second leecher then becomes interested, and by the end of the diagram, it is shown that the second leecher is downloading from both the seed and the first leecher simultaneously, even though the leecher does not have the entire file.

Figure 6.1 (on the next page): Two leechers (to the right) downloading a file from a seed. When the first peers completes its first piece, the other peer starts downloading this piece while also downloading from the seed.


:Peer
:Peer
:Peer
handshake(Peer 2, True)
handshake(Peer 1, False)
bitfield(Peer 2, True, C .. (False,Nil)))))))
bitfield(Peer 1, False, .. s(True,Nil)))))))
interested(Peer 2)
unchoke(Peer 1)
request(Peer 2, 2)
piece(Peer 1, 2)
handshake(Peer 3, True)
handshake(Peer 3, True)
handshake(Peer 2, False)
bitfield(Peer 3, True, C .. (False,Nil)))))))
bitfield(Peer 2, False, .. (False,Nil)))))))
handshake(Peer 1, False)
bitfield(Peer 3, True, C .. (False,Nil)))))))
bitfield(Peer 1, False, .. s(True,Nil)))))))
interested(Peer 3)
unchoke(Peer 1)
request(Peer 3, 0)
piece(Peer 1, 0)
request(Peer 2, 3)
piece(Peer 1, 3)
have(Peer 2, 1)
have(Peer 2, 1)
request(Peer 2, 8)
request(Peer 3, 1)
interested(Peer 3)
unchoke(Peer 2)
request(Peer 3, 2)
piece(Peer 2, 2)
piece(Peer 1, 8)
piece(Peer 1, 1)

Chapter 7

Conclusion

In this chapter I present my findings and experiences with modeling distributed algorithm with the ABS language.

7.1 Summary

Chapter 3 started with two implementations of the Dining Philosophers problem. The original pseudo code of Dijkstra's naive implementation was easily translated into ABS code, and the semaphore operations were realized with a separate fork class with methods fork grabbing and releasing. The principle of deadlock could be demonstrated by running the program, and also starvation was shown through a sequence diagram created with the ABS Eclipse plug-in.

In the implementation of Chandy and Misra's *hygienic solution*, letting the objects send asynchronous method calls to themselves was an experimental approach to create scheduling point. This gave a nice visualization of the internal states of the philosophers along with the rest of the information in in the sequence diagrams.

In chapter 4 it was shown more clearly how defining different interfaces with different methods is a way of specifying a protocol between agents. Also, more data types were defined which proved to be a clear and concise way to express data. Test programs were set up and instances of the algorithm were run successfully.

The experimentation with modeling faulty communication was also quite successful; The basic implementations provided the logic to uphold the protocol, while the additional classes would stop, rearrange, or duplicate the method calls. Also delayed messages and time-outs by the proposers were mimicked by having the proposers restarting the prepare phase.

A larger model was implemented with the Kademlia distributed hash table in chapter 5. A successful run had 10 000 nodes joining one swarm at once, indeed proving some scalability of ABS.

In chapter 6 and the BitTorrent model I got to display some of the workings of the *Rarest first* algorithm and demonstrate how a file is duplicated with a sequence diagram from a test run of the model.

The different models that were created needed different schemes for synchronization. In the naive solution to the Dining Philosophers problem, a functionality resembling semaphores was implemented on the forks. In the Paxos implementa-

tion, the agents had both active and passive behavior, sending messages and waiting silently for replies; Reactions were triggered in the methods corresponding to the different remote procedure calls. In the node lookup procedure in the Kademlia implementation however, the method did not terminate immediately after the outgoing messages were sent; The responses were registered in their respective methods, while the main method was waiting to continue and process the responses.

7.2 Tool evalution

My evaluation of the ABS language and tools.

The functional layer of ABS

The algebraic data types and the functions defined on these, makes for a clean and concise way to express ideas. The data type definitions even let the user define complete data structures, and in a very compact manner.

The fact that algebraic data types are immutable makes modeling easy, as it is guaranteed that no other process may change the content of a local value, even if that value is shared with another object. Thinking about structures like lists as nothing more than values, can be liberating, and especially the pattern matching of the case expression can be used quite elegantly.

However there are admittedly times when the algebraic data types can feel a little awkward, such as in the transition between the functional layer and the imperative layer – for instance iterating a list of objects in order to call some function on them.

Recursive, pure functions, are however, as long as they do not get too complex, a quite clean way of expressing operations on the data types. But when the functionality becomes complicated, one can quickly loose track of it. Some of the functions defined on the Kademlia routing tables demonstrated this. Still, it would seem much more cumbersome to try to implement the data structure with the current object-orientation of ABS; The algebraic data types are certainly the idiomatic way to do this.

However, as was seen with the first version of the XOR function (Section 5.4.1), the recursive calls tend to be quite expensive. I think ABS could benefit from having a data structure like an array, with random access and side effects, even though it may not directly comply with the philosophy of ABS.

The distributed layer

The concurrency model of ABS has made modeling the communication between processes in the distributed algorithms very simple. The ease of implementing each remote procedure call as a function, along with the cooperative scheduling, allowed me for the most part to ignore the distributed aspects of the algorithm while programming the local logic.

The COGs which run alongside each other completely concurrently, act as separate instances of an implementation. This level of abstraction is hard to accomplish in many languages. Typically one needs to actually have multiple

instances of a program running, and the communication realized either through middle-ware, or even low-level constructs such as sockets.

With the functionality of the distributed layer, it was possible to model locking of objects, active and passive behavior, queuing of messages, and even faulty communication.

Concluding remarks

A few problems have been encountered during my work with the ABS language, such as bugs with the Java backend of the compiler. But overall, I find ABS as a good tool for modeling distributed algorithms. With the high abstraction of communication between processes in the language, ABS code expresses the overall behavior of distributed algorithms quite clearly. With a simple object-oriented model and data types without side-effects, ABS does a good job of eliminating surprising behavior, and lets the user focus on the big picture.

During the work on this thesis I have learned a lot, and I must say the working with the ABS language has been instructive.

Bibliography

[1] Gregory R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison Wesley Longman, Inc., 2000. ISBN: 0-201-35752-6.

[2] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems." In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 335–350.

[3] Tushar Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective." In: *In Proc. of PODC*. ACM Press, 2007, pp. 398–407.

[4] K. M. Chandy and J. Misra. "The drinking philosophers problem." In: *ACM Trans. Program. Lang. Syst.* 6.4 (Oct. 1984), pp. 632–646. ISSN: 0164-0925. DOI: 10.1145/1780.1804. URL: http://doi.acm.org/10.1145/1780.1804.

[5] Dave Clarke et al. "Modeling Spatial and Temporal Variability with the HATS Abstract Behavioral Modeling Language." In: *Formal Methods for Eternal Networked Software Systems*. Ed. by Marco Bernardo and Valérie Issarny. Vol. 6659. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 417–457. ISBN: 978-3-642-21454-7.

[6] Bram Cohen. *Incentives build robustness in BitTorrent*. 2003. URL: http://www.bittorrent.org/bittorrentecon.pdf.

[7] Bram Cohen. *The BitTorrent Protocol Specification*. 2008. URL: http://www.bittorrent.org/beps/bep_0003.html (visited on 05/06/2014).

[8] Edsger W. Dijkstra. "Hierarchical ordering of sequential processes." In: *Acta Informatica* 1 (2 1971), pp. 115–138. ISSN: 0001-5903. DOI: 10.1007/BF00289519. URL: http://dx.doi.org/10.1007/BF00289519.

[9] Edsger W. Dijkstra. "Twenty-eight years." circulated privately. Jan. 1987. URL: http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1000.PDF.

[10] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. "ABS: A Core Language for Abstract Behavioral Specification." In: *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*. Vol. 6957. Lecture Notes in Computer Science. Springer-Verlag, 2011, pp. 142–164.

[11] David Karger et al. “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web.” In: *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*. STOC ’97. El Paso, Texas, USA: ACM, 1997, pp. 654–663. ISBN: 0-89791-888-6. DOI: 10.1145/258533.258660. URL: http://doi.acm.org/10.1145/258533.258660.

[12] L. Lamport and M. Massa. “Cheap Paxos.” In: *Dependable Systems and Networks, 2004 International Conference on*. 2004, pp. 307–314. DOI: 10.1109/DSN.2004.1311900.

[13] Leslie Lamport. *My writings*. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html (visited on 05/09/2014).

[14] Leslie Lamport. “Paxos made simple.” In: *ACM Sigact News* 32.4 (2001), pp. 18–25. URL: http://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/past/03F/notes/paxos-simple.pdf.

[15] Leslie Lamport. “The part-time parliament.” In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

[16] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[17] Arnaud Legout, Guillaume Urvoy-Keller, and Pietro Michiardi. *Understanding BitTorrent: An Experimental Perspective*. Tech. rep. 2005, p. 16. URL: http://hal.inria.fr/inria-00000156.

[18] Daniel Lehmann and Michael O. Rabin. “On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem.” In: *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’81. Williamsburg, Virginia: ACM, 1981, pp. 133–138. ISBN: 0-89791-029-X. DOI: 10.1145/567532.567547. URL: http://doi.acm.org/10.1145/567532.567547.

[19] Petar Maymounkov and David Mazières. “Kademlia: A Peer-to-Peer Information System Based on the XOR Metric.” In: *Peer-to-Peer Systems*. Ed. by Peter Druschel, Frans Kaashoek, and Antony Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-44179-3. DOI: 10.1007/3-540-45748-8_5. URL: http://dx.doi.org/10.1007/3-540-45748-8_5.

[20] Arvid Norberg. *Introduction to BitTorrent*. Tech. rep. TDBC85, Distributed systems C, Umeå University, 2006. URL: http://www.libtorrent.org/bittorrent.pdf.

[21] *The ABS Language Specification*. included in the ABS Tools repository.

Appendices

Appendix A

The Dining Philosophers

A.1 Complete implementation of naive solution

```
1  module Philosophers;
2  import println from ABS.Meta;
3
4
5  interface Philosopher { }
6
7  interface Fork {
8      Unit grab(Philosopher owner);
9      Unit release(Philosopher owner);
10 }
11
12
13 class Philosopher(Fork f1, Fork f2) implements Philosopher {
14
15     Unit run() {
16         while (True) {
17             this.tell("is thinking");
18
19             Fut<Unit> f;
20             f = f1!grab(this);
21             f.get;
22
23             this.tell("grabbed left fork");
24
25             f = f2!grab(this);
26             f.get;
27
28             this.tell("grabbed right fork");
29             this.tell("is eating");
30
31             f1!release(this);
32             f2!release(this);
```

```
33
34              this.tell("released both forks");
35          }
36      }
37
38      Unit tell(String s) {
39          Unit p = println(toString(this)+" "+s);
40      }
41  }
42
43  class Fork implements Fork {
44      Philosopher owner;
45
46      Unit grab(Philosopher p) {
47          await owner == null;
48          owner = p;
49      }
50
51      Unit release(Philosopher p) {
52          if (p == owner)
53              owner = null;
54      }
55  }
56
57
58  {   // Main
59      Fork f1 = new Fork();
60      Fork f2 = new Fork();
61      Fork f3 = new Fork();
62      Fork f4 = new Fork();
63      Fork f5 = new Fork();
64
65      new Philosopher(f1, f2);
66      new Philosopher(f2, f3);
67      new Philosopher(f3, f4);
68      new Philosopher(f4, f5);
69      new Philosopher(f5, f1);
70  }
```

A.2 Complete implementation of hygienic solution

```
1  module Chandy;
2  import println from ABS.Meta;
3
4
5  type ForkID = Int;
6  data FState = Clean | Dirty | RequestToken;
7  data Fork = Fork(ForkID forkID, FState state);
8
9  def Bool needsFork(Fork f) =
10     state(f) == RequestToken;
11
12 data PState = Thinking | Hungry | Eating;
13
14
15 interface Philosopher {
16     Fork request(ForkID f);
17     Unit start(Philosopher a, Philosopher b);
18 }
19
20
21 class Philosopher(Fork forkL, Fork forkR) implements Philosopher {
22     Philosopher philL;
23     Philosopher philR;
24
25     PState state = Thinking;
26
27     Unit start(Philosopher left, Philosopher right) {
28         philL = left;
29         philR = right;
30
31         this!think();
32     }
33
34     Unit think(){
35         state = Hungry;
36         this!hungry();
37     }
38
39     Unit hungry() {
40         while (needsFork(forkL) || needsFork(forkR)) {
41
42             if (needsFork(forkL) && needsFork(forkR)) {
43                 Fut<Fork> reqL = philL!request(forkID(forkL));
44                 Fut<Fork> reqR = philR!request(forkID(forkR));
45                 await reqL? & reqR? ;
```

```
46                  forkL = reqL.get;
47                  forkR = reqR.get;
48
49              } else if (needsFork(forkL)) {
50                  Fut<Fork> req = philL!request(forkID(forkL));
51                  await req?;
52                  forkL = req.get;
53
54              } else if(needsFork(forkR)) {
55                  Fut<Fork> req = philR!request(forkID(forkR));
56                  await req?;
57                  forkR = req.get;
58              }
59
60          }
61
62          state = Eating;
63          this!eat();
64      }
65
66      Unit eat() {
67          forkL = Fork(forkID(forkL), Dirty);
68          forkR = Fork(forkID(forkR), Dirty);
69
70          state = Thinking;
71          this!think();
72      }
73
74
75      Fork request(ForkID i) {
76          if (i == forkID(forkL)) {
77              await this.state != Eating && state(forkL) == Dirty;
78              forkL = Fork(i, RequestToken);
79
80          } else if (i == forkID(forkR)) {
81              await this.state != Eating && state(forkR) == Dirty;
82              forkR = Fork(i, RequestToken);
83
84          } else
85              Unit p = println("ERROR!");
86
87          return Fork(i, Clean);
88      }
89
90  }
91
92
93
```

```
94  {   // MAIN
95      Philosopher p1 = new Philosopher(Fork(1,RequestToken), Fork(2,RequestToken));
96      Philosopher p2 = new Philosopher(Fork(2,Dirty),        Fork(3,RequestToken));
97      Philosopher p3 = new Philosopher(Fork(3,Dirty),        Fork(4,RequestToken));
98      Philosopher p4 = new Philosopher(Fork(4,Dirty),        Fork(5,RequestToken));
99      Philosopher p5 = new Philosopher(Fork(5,Dirty),        Fork(1,Dirty));
100
101     p1!start(p5, p2);
102     p2!start(p1, p3);
103     p3!start(p2, p4);
104     p4!start(p3, p5);
105     p5!start(p4, p1);
106 }
```

Appendix B

Paxos

B.1 Complete basic implementation

```
1  /**
2   * Single instance of the Paxos algorithm.
3   * The proposer does not take into consideration the possibility of an
4   * conflict, and only sends one prepare.
5   */
6  module Paxos;
7  export *;
8
9  import println from ABS.Meta;
10
11
12 type Pnum = Int;
13 data Value = Value(Int);
14 data Proposal = None | Proposal(Pnum, Value value);
15
16 def Pnum number(Proposal p) =
17     case p {
18         None => 0;
19         Proposal(n, _) => n;
20     };
21
22 // Interfaces:
23
24
25 interface Acceptor {
26     // Phase 1.a of the consensus algorithm:
27     Unit prepare(Proposer caller, Pnum n);
28
29     // Phase 2.a of the consensus algorithm:
30     Unit accept(Proposal p);
31 }
32
```

```
33  interface Proposer {
34      // Phase 1.b of the consensus algorithm:
35      Unit response(Acceptor caller, Pnum n, Proposal accepted);
36  }
37
38  interface ActiveProposer extends Proposer {
39      // Start the algorithm:
40      Unit run();
41      Pnum currentPnum();
42  }
43
44  interface Learner {
45      // Learns a value from an acceptor. Needs to receive this from a majority
46      // of the acceptors before the value actually is learned.
47      Unit report(Acceptor caller, Proposal p);
48  }
49
50
51  // Classes:
52
53
54  class Proposer(Value v, Pnum current, Int nProposers, Int nAcceptors,
55                 List<Acceptor> acceptors) implements ActiveProposer {
56
57      Set<Acceptor> responders = EmptySet;
58      Proposal maxResponse = None;
59      Bool cutoff = False;
60
61
62      Unit run() {
63          current = current + nProposers;
64          responders = EmptySet;
65          cutoff = False;
66
67          List<Acceptor> l = acceptors;
68
69          while (l != Nil) {
70              Acceptor next = head(l);
71              l = tail(l);
72              next!prepare(this, current);
73              suspend; // Allows quick responses to be handled immediately.
74          }
75      }
76
77      Pnum currentPnum() {
78          return current;
79      }
80
```

```
81
82      Unit response(Acceptor caller, Pnum n, Proposal reported) {
83          if (n == current && ~contains(responders, caller)) {
84              responders = insertElement(responders, caller);
85
86              if (number(reported) > number(maxResponse))
87                  maxResponse = reported;
88
89              if (~cutoff && size(responders) > nAcceptors / 2) {
90                  Proposal p = Proposal(n, v);
91
92                  if (maxResponse != None)
93                      p = Proposal(n, value(maxResponse));
94
95                  List<Acceptor> l = acceptors;
96
97                  while (l != Nil) {
98                      Acceptor next = head(l);
99                      l = tail(l);
100                     next!accept(p);
101                 }
102
103                 cutoff = True;
104             }
105         }
106     }
107
108 }
109
110
111 class Acceptor(List<Learner> learners) implements Acceptor {
112     Proposal accepted = None;
113     Pnum maxPrepare = 0;
114
115     Unit prepare(Proposer caller, Pnum n) {
116         if (n > maxPrepare) {
117             maxPrepare = n;
118             caller!response(this, n, accepted);
119         }
120     }
121
122     Unit accept(Proposal p) {
123         if (number(p) >= maxPrepare) {
124             accepted = p;
125
126             List<Learner> l = learners;
127
128             while (l != Nil) {
```

```
129                 head(l)!report(this, p);
130                 l = tail(l);
131             }
132         }
133     }
134
135 }
136
137
138 class Learner(Int nAcceptors) implements Learner {
139     Maybe<Value> learned = Nothing;
140     Map<Value, Int> counts = EmptyMap;
141     Map<Acceptor, Proposal> votes = EmptyMap;
142
143     Unit report(Acceptor caller, Proposal p) {
144         if (learned == Nothing) {
145             Bool valid = True;
146             Maybe<Proposal> oldVote = lookup(votes, caller);
147
148             if (oldVote != Nothing) {
149                 if (number(p) > number(fromJust(oldVote))) {
150                     Value v = value(fromJust(oldVote));
151                     Int count = lookupUnsafe(counts, v);
152
153                     counts = put(counts, v, count - 1);
154
155                 } else
156                     valid = False;
157             }
158
159             if (valid) {
160                 Value v = value(p);
161                 Int count = 1 + lookupDefault(counts, v, 0);
162
163                 counts = put(counts, v, count);
164                 votes = put(votes, caller, p);
165
166                 if (count > nAcceptors / 2)
167                     this!learn(value(p));
168             }
169
170         }
171     }
172
173     Unit learn(Value v) {
174         learned = Just(v);
175     }
176 }
```

B.2 Faulty communication

```
module FaultyAgents;

export FaultyProposer, FaultyAcceptor;
export * from Paxos;

import * from Paxos;


def Bool chance(Int i, Int n) = random(n) < i;

class AcceptorProxy(Proposer owner, Acceptor target) implements Acceptor {
    Maybe<Either<Pnum, Proposal>> delayed = Nothing;

    Unit prepare(Proposer caller, Pnum n) {
        Bool furtherDelay = False;

        // Normal message delivery
        if (chance(8, 10)) {
            target!prepare(owner, n);

        // Message duplication:
        } else if (chance(1, 3)) {
            target!prepare(owner, n);
            target!prepare(owner, n);

        // Message delay:
        } else if (chance(1, 2)) {
            if (isJust(delayed))
                target!prepare(owner, n);
            else
                delayed = Just(Left(n));

            furtherDelay = True;

        // Message loss;
        } else
            skip;

        if (~furtherDelay)
            this.sendDelayed();
    }

    Unit accept(Proposal p) {
        Bool delay = False;

```

```
46          if (chance(8, 10)) {
47              target!accept(p);
48
49          } else if (chance(1, 3)) {
50              target!accept(p);
51              target!accept(p);
52
53          } else if (chance(1, 2)) {
54              if (isJust(delayed))
55                  target!accept(p);
56              else
57                  delayed = Just(Right(p));
58
59              delay = True;
60
61          } else
62              skip;
63
64
65          if (~delay)
66              this.sendDelayed();
67      }
68
69      Unit sendDelayed() {
70          if (isJust(delayed)) {
71              // new cool case statement:
72              //case fromJust(delayed) {
73              //    Left(m) => caller!prepare(owner, m);
74              //    Right(p) => caller!accept(p);
75              //};
76
77              // old boring ifs for compability with Eclipse
78              Either<Pnum, Proposal> e = fromJust(delayed);
79
80              if (isLeft(e))
81                  target!prepare(owner, left(e));
82              else
83                  target!accept(right(e));
84
85              delayed = Nothing;
86          }
87      }
88
89  }
90
91  // "Main" class:
92
93  class FaultyProposer(Value v, Pnum init, Int nProposers, Int nAcceptors,
```

```
94          List<Acceptor> acceptors) implements Proposer {
95
96      ActiveProposer slave;
97      List<Acceptor> proxies = Nil;
98
99      Unit run() {
100         List<Acceptor> ax = acceptors;
101
102         while (ax != Nil) {
103             Acceptor proxy = new local AcceptorProxy(this, head(ax));
104             proxies = appendright(proxies, proxy);
105             ax = tail(ax);
106         }
107
108         slave = new local Proposer(v, init, nProposers, nAcceptors,
109         proxies);
110     }
111
112     Unit response(Acceptor caller, Pnum n, Proposal reported) {
113         Int current = slave.currentPnum();
114
115         // Timeout:
116         if (n == current && chance(2, 10))
117             slave!run();
118
119         // Normal delivery:
120         else
121             slave.response(caller, n, reported);
122     }
123
124 }
125
126
127 class ProposerProxy(Acceptor owner, Proposer target) implements Proposer {
128     Maybe<Pair<Pnum, Proposal>> delayed = Nothing;
129
130     Unit response(Acceptor caller, Pnum n, Proposal accepted) {
131         Bool delay = False;
132
133         if (chance(8, 10)) {
134             target!response(owner, n, accepted);
135
136         } else if (chance(1, 3)) {
137             target!response(owner, n, accepted);
138             target!response(owner, n, accepted);
139
140         } else if (chance(1, 2)) {
141             if (isJust(delayed))
```

```
142                 target!response(owner, n, accepted);
143             else
144                 delayed = Just(Pair(n, accepted));
145
146             delay = True;
147
148         } else
149             skip; // message loss
150
151
152         //if (~delay) {
153         //   case delayed {
154         //      Nothing => skip;
155         //      Just(Pair(pnum, acc)) => target!response(owner, pnum, acc);
156         //   }
157         //}
158
159         if (~delay && isJust(delayed)) {
160             Pair<Pnum, Proposal> e = fromJust(delayed);
161
162             target!response(owner, fst(e), snd(e));
163             delayed = Nothing;
164         }
165     }
166 }
167
168 class LearnerProxy(Learner target) implements Learner {
169     Maybe<Proposal> delayed = Nothing;
170
171     Unit report(Acceptor caller, Proposal p) {
172         Bool delay = False;
173
174         if (chance(8, 10)) {
175             target!report(caller, p);
176
177         } else if (chance(1, 3)) {
178             target!report(caller, p);
179             target!report(caller, p);
180
181         } else if (chance(1, 2)) {
182             if (isJust(delayed))
183                 target!report(caller, p);
184
185             else
186                 delayed = Just(p);
187
188             delay = True;
189
```

```
190         } else
191             skip;
192
193         if (~delay && isJust(delayed)) {
194             target!report(caller, fromJust(delayed));
195             delayed = Nothing;
196         }
197     }
198 }
199
200 // takes an extra argument nAcceptors because it needs to pass it on to learnerProxies.
201 class FaultyAcceptor(Int nAcceptors, List<Learner> learners) implements Acceptor {
202     Acceptor slave;
203     List<Learner> learnerProxies = Nil;
204     Map<Proposer, Proposer> proposerProxies = EmptyMap;
205
206     Unit run() {
207         List<Learner> iterator = learners;
208
209         while (iterator != Nil) {
210             Learner proxy = new local LearnerProxy(head(iterator));
211             learnerProxies = appendright(learnerProxies, proxy);
212             iterator = tail(iterator);
213         }
214
215         slave = new local Acceptor(learnerProxies);
216     }
217
218     Unit prepare(Proposer caller, Pnum n) {
219         Proposer proxy;
220         Maybe<Proposer> lookup = lookup(proposerProxies, caller);
221
222         if (lookup == Nothing)
223             proxy = new local ProposerProxy(this, caller);
224         else
225             proxy = fromJust(lookup);
226
227         slave.prepare(proxy, n);
228     }
229
230     Unit accept(Proposal p) {
231         slave.accept(p);
232     }
233 }
```

Appendix C

Kademlia

C.1 Full implementation

```
1  module Kademlia;
2  export *;
3
4  import * from ABS.Meta;
5
6  type Value = String;
7  type Hash = Int;
8  type ID = Hash;
9
10 data Contact = Contact(ID id, Node node);
11
12 data Table = EmptyTable
13            | KBucket(List<Contact> contacts, Int)
14            | Split(Table, Table, Int);
15
16 def Int tSize(Table t) =
17     case t {
18         EmptyTable => 0;
19         KBucket(_, n) => n;
20         Split(_, _, n) => n;
21     };
22
23 // Longest Unique Suffix,
24 // reverse of Longest Common Prefix
25 //
26 def Int suffix(Int a, Int b) =
27     if a == b then
28         0
29     else
30         1 + suffix(a/2, b/2);
31         // always terminates: integer division goes towards zero
32         // symmetric
```

```
33
34  def List<A> keep<A>(List<A> list, Int n) =
35      if n == 0 then Nil
36      else case list {
37              Nil => Nil;
38              Cons(last, Nil) => list;
39              Cons(head, tail) => Cons(head, keep(tail, n-1));
40          };
41
42  // Primitive insertion to k−bucket
43  //
44  def Table pushContact(Contact c, Table t) =
45      case t {
46          EmptyTable => KBucket(Cons(c, Nil), 1);
47          KBucket(list, n) => KBucket(Cons(c, list), n+1);
48      };
49
50  // Split a k−bucket (NOT EmptyTable/Split)
51  // d = routing table depth
52  //
53  def Table split(Table t, Int d, Int w, Hash b) =
54      split2(contacts(t), d, w, b, EmptyTable, EmptyTable);
55
56  def Table split2(List<Contact> q,
57          Int d, Int w, Hash b, Table l, Table r) =
58      case q {
59          Nil => Split(l, r, tSize(l)+tSize(r));
60          Cons(head, tail) => if w-suffix(id(head),b) == d
61            then
62              split2(tail, d, w, b, pushContact(head, l), r)
63            else
64              split2(tail, d, w, b, l, pushContact(head, r));
65      };
66
67
68  // Get n closest contacts to target
69  //
70  def List<Contact> findContacts(Table t, ID target, Int n, Int w, ID b) =
71      let (Int depth) = w - suffix(target, b) in
72      getContacts(t, n, depth);
73
74  def List<Contact> getContacts(Table t, Int n, Int d) =
75      if n > 0 then
76          case t {
77              Split(l, r, _) => let (Int m) = if d == 0 then
78                                                  tSize(l)
79                                              else if tSize(r) < n then
80                                                  intMin(n-tSize(r), tSize(l))
```

```
81                                          else 0 in
82                              let (List<Contact> ll) = getContacts(l,m,0) in
83                              concatenate(ll, getContacts(r, n-m, d-1));
84
85              EmptyTable => Nil;
86              KBucket(list,k) => if n >= k then list
87                                 else keep(list, n);
88          }
89      else
90          Nil;
91
92  def Int intMin(Int a, Int b) = if a < b then a else b;
93
94  // The main insert function
95  //
96  def Table insertContact(Table t, Contact c, Int k, Int w, ID b) =
97      let (Int depth) = w - suffix(b, id(c)) in
98      insert_(t, c, k, w, b, depth, 0);
99
100 def Table insert_(Table t, Contact c, Int k, Int w, ID b, Int dC, Int d) =
101     case t {
102         Split(left, right, _) =>
103             if d == dC then
104                 let (Table l) = addContact(left, c, k) in
105                 Split(l, right, tSize(l)+tSize(right))
106             else
107                 let (Table r) = insert_(right, c, k, w, b, dC, d+1) in
108                 Split(left, r, tSize(left)+tSize(r));
109
110         KBucket(list, n) => if includes(list, c) then
111                                 t
112                             else if n < k then
113                                 pushContact(c, t)
114                             else // split and try again:
115                                 let (Table s) = split(t, d, w, b) in
116                                 insert_(s, c, k, w, b, dC, d);
117
118         EmptyTable => initBucket(c);
119     };
120
121 def Bool includes<A>(List<A> list, A e) =
122     case list {
123         Nil => False;
124         Cons(e, _) => True;
125         Cons(head, tail) => includes(tail, e);
126     };
127
128 def Table initBucket(Contact c) =
```

```
129     KBucket(Cons(c, Nil), 1);
130
131 def Table addContact(Table t, Contact c, Int k) =
132     case t {
133         EmptyTable => initBucket(c);
134         KBucket(list, n) => if includes(list, c) || n >= k then
135                                 t
136                             else
137                                 KBucket(Cons(c, list), n+1);
138         _ => t;
139     };
140
141
142 def List<Contact> insertSorted(List<Contact> list, Contact item, Hash
143 target) =
144     case list {
145         Cons(item, tail) => list;        // asume it's in right position
146         Cons(head, tail) => if xor(id(item), target) < xor(id(head), target) th
147                                 Cons(item, list)
148                             else
149                                 Cons(head, insertSorted(tail, item, target));
150         Nil => Cons(item, Nil);
151     };
152
153
154 // xor functions that doen't work:
155
156 // XXX CRASHES COMPILER:
157 // def Int xor3(Int a, Int b) =
158 //    abs(a%2 − b%2) + 2 ∗ xor3(a/2, b/2);
159
160 def Int xor4(Int a, Int b) =
161     if a == b then
162         0
163     else
164         // this doesn't work! parenthesis have no effect
165         (if a%2 == b%2 then 0 else 1) + 2 * xor4(a/2, b/2);
166
167 def Int xor(Int a, Int b) =
168     if a == b then
169         0
170     else
171         let (Int x) = if a%2 == b%2 then 0 else 1 in
172         x + 2 * xor(a/2, b/2);
173
174 // return all KBuckets further away than the closest neighbor.
175 //
176 def List<Table> distantNeighbors(Table table) =
```

```
177     case table {
178         Split(_, _, _) => Nil;
179         Split(left, right, n) => Cons(left, distantNeighbors(right));
180         _ => Nil;
181     };
182 
183 /*
184  * DEBUGGING
185  */
186 
187 def String contactsToString(List<Contact> l) =
188     case l {
189         Nil => "";
190         Cons(x, Nil) => toString(id(x));
191         Cons(x, tail) => toString(id(x)) +" "+ contactsToString(tail);
192     };
193 
194 def String tableToString(Table t) =
195     case t {
196         EmptyTable => "E";
197         KBucket(list, n) => "(kbucket "+ contactsToString(list) +")";
198         Split(r, l, n) => "(split "+ tableToString(r) +" "+
199                           tableToString(l) +")";
200     };
201 
202 def Unit pTable(Table t) = println(tableToString(t));
203 
204 // Defines methods corresponding to the different message types
205 interface Node {
206     Unit ping(Contact caller);
207     Unit pong(Contact caller);
208 
209     Unit store(Hash key, Value v, Contact caller);
210 
211     Unit findNode(ID node, Contact caller);
212     Unit findNodeReply(List<Contact> contacts, Contact responder);
213 
214     Unit findValue(Hash key, Contact caller);
215     Unit findValueReply(Value v, Contact responder);
216 }
217 
218 interface AdvancedNode extends Node {
219 
220     Unit join(Node node, ID id);
221     Unit init();
222 
223     Unit publish(Hash key, Value v);
224     Maybe<Value> find(Hash key);
```

```
225
226     Unit insert(Contact c);
227     Unit printContacts();
228 }
229
230 class Node(ID ownID, Int k, Int alpha, Int w) implements AdvancedNode {
231     Map<Hash, Value> storedValues = EmptyMap;
232     Table contacts = EmptyTable;
233     Bool busy = True;
234
235     // when awaiting replies:
236     Int pendingReplies = 0;
237     List<Contact> nodesFound = Nil; // these are the contacts to which we
238                                     // should replublish a value
239     Bool valueLookup = False;
240     Maybe<Value> valueFound = Nothing;
241
242     List<Contact> queue = Nil;
243     Set<Contact> tried = EmptySet; // visited; already tried contacts, in case
244                                    // of incoming duplicates
245
246     Hash target = -1;
247
248
249     // Insert node correctly into routing table
250     //
251     Unit insert(Contact c) {
252         contacts = insertContact(contacts, c, k, w, ownID);
253     }
254
255     List<Contact> searchContacts(Hash hash) {
256         return findContacts(contacts, hash, k, w, ownID);
257     }
258
259     Unit registerContacts(List<Contact> contacts) {
260         List<Contact> iterator = contacts;
261
262         while (iterator != Nil) {
263             Contact c = head(iterator);
264
265             if (node(c) != this && ~contains(this.tried, c)) {
266                 this.nodesFound =
267                     insertSorted(this.nodesFound, c, this.target);
268                 this.queue =
269                     insertSorted(this.queue, c, this.target);
270             }
271
272             iterator = tail(iterator);
```

```
273         }
274     }
275
276     // RPC
277     // return k closest nodes to ID
278     Unit findNode(ID nodeID, Contact caller) {
279         List<Contact> l = this.searchContacts(nodeID);
280         Node n = node(caller);
281
282         n!findNodeReply(l, Contact(this.ownID,this));
283         this.insert(caller);
284     }
285
286     // Locate the k closest nodes to hash in the entire DHT
287     //
288     Either<Maybe<Value>, List<Contact>> nodeLookup(Hash hash, Bool findValue) {
289         this.target = hash;
290         this.valueLookup = findValue;
291         this.queue = Nil;
292         this.nodesFound = Nil;
293         this.tried = EmptySet;
294         this.pendingReplies = 0;
295
296         List<Contact> init = this.searchContacts(hash);
297         this.registerContacts(init); // !!!
298
299         Int nContacts = tSize(this.contacts);
300         Int kk = min(k, nContacts);
301         Int nRPCs = min(alpha, nContacts);
302
303         Int responses = 0;
304
305         while (responses < kk && this.valueFound == Nothing) {
306
307             Int i = 0;
308             while (i < nRPCs && this.queue != Nil) {
309                 Contact c = head(this.queue);
310                 Node node = node(c);
311
312                 this.tried = insertElement(this.tried, c);
313                 this.pendingReplies = pendingReplies + 1;
314
315                 if (findValue) {
316                     node!findValue(hash,Contact(this.ownID, this));
317
318                 } else {
319                     node!findNode(hash, Contact(this.ownID, this));
320                 }
```

```
321
322                 i = i + 1;
323                 this.queue = tail(this.queue);
324             }
325
326             await this.pendingReplies == 0 || this.valueFound != Nothing;
327
328             responses = responses + nRPCs - this.pendingReplies;
329
330             // Updating kk and nRPCs as if timeouts were possible
331             kk = min(this.k, kk + length(this.queue));
332
333             nRPCs = if this.pendingReplies == nRPCs then
334                         this.k       // whole round failed — try more!
335                     else
336                         this.alpha;
337         }
338
339         return if valueLookup then
340                     Left(this.valueFound)
341                 else
342                     Right(keep(this.nodesFound, this.k));
343     }
344
345     // RPC
346     //
347     Unit findNodeReply(List<Contact> found, Contact responder) {
348         this.registerContacts(found);
349         this.pendingReplies = this.pendingReplies - 1;
350
351         while (found != Nil) {
352             Contact c = head(found);
353
354             if (node(c) != this)
355                 this.insert(c);
356
357             found = tail(found);
358         }
359
360         // XXX should also insert responder,
361         //     but since we assume non-faulty communication and perfect nodes,
362         //     we know that the responder must already be in our routing table,
363         //     and it doesn't need to be updated, as no nodes will fall out
364     }
365
366     // Join the network using given bootstrap node
367     //
368     Unit join(Node node, ID id) {
```

```
369             this.insert(Contact(id,node));
370             this.nodeLookup(this.ownID, False);
371 
372             // bucket refresh, disabled because of complexity
373 
374             this.busy = False;
375         }
376 
377         // RPC
378         //
379         Unit findValue(Hash key, Contact caller) {
380             Node n = node(caller);
381             Contact signature = Contact(this.ownID, this);
382 
383             Maybe<Value> v = lookup(this.storedValues, key);
384 
385             if (v == Nothing) {
386                 List<Contact> l = this.searchContacts(key);
387                 n!findNodeReply(l, signature);
388 
389             } else {
390                 n!findValueReply(fromJust(v), signature);
391             }
392 
393             this.insert(caller);
394         }
395 
396         // RPC
397         //
398         Unit findValueReply(Value v, Contact responder) {
399             // double check that we are in fact looking for a value
400             // (terminates lookup)
401             if (this.valueLookup) {
402                 this.valueFound = Just(v);
403             }
404         }
405 
406         Unit publish(Hash key, Value value) {
407             await this.busy == False;
408 
409             Either<Maybe<Value>, List<Contact>> lookup = this.nodeLookup(key, False);
410             List<Contact> contacts = right(lookup);
411 
412             while (contacts != Nil) {
413                 Node n = node(head(contacts));
414 
415                 n!store(key, value, Contact(this.ownID, this));
416                 contacts = tail(contacts);
```

```
417             }
418         }
419
420         // search the DHT for value accociated with key
421         Maybe<Value> find(Hash key) {
422             await busy == False;
423
424             this.nodeLookup(key, True);
425             // TODO
426             // re−publish value to k closest nodes that did ∗not∗ have the value!
427             return this.valueFound;
428         }
429
430         Unit refresh() {  }
431
432         // RPC
433         //
434         Unit ping(Contact caller) {}
435         Unit pong(Contact responder) {}
436
437         // RPC
438         //
439         Unit store(Hash key, Value value, Contact caller) {
440             this.storedValues = InsertAssoc(Pair(key,value), this.storedValues);
441             // TODO insert caller into contacts?
442         }
443
444         Unit init() {
445             this.busy = False;
446         }
447
448         Unit printContacts() {
449             await busy == False;
450
451             Unit p = println(toString(ownID));
452                  p = pTable(contacts);
453         }
454
455     }
456
457
458     // MAIN
459     {
460         // Test program initiating a large number of nodes
461
462         Int n = 1000;  // number f nodes
463         Int k = 6;
464         Int alpha = 3;
```

```
465      Int w = 20;
466
467      Int limit = 1000000;
468
469      AdvancedNode bootstrap = new Node(1337, k, alpha, w);
470      bootstrap!init(); //!!!
471
472      AdvancedNode m;
473      Int i = 0;
474
475      while (i < n) {
476          m = new Node(random(limit), k, alpha, w);
477          m!join(bootstrap,1337);
478          i = i + 1;
479      }
480
481      m!printContacts();
482  }
```

Appendix D

BitTorrent

D.1 Full implementation

```
module BitTorrent;
export *;

import println from ABS.Meta;

// data types
data State = Init | Leecher | Endgame | Seed;

data PeerInfo = Info(Bitfield bf, Bool choking, Bool interesting, Bool choked,
Bool interested);

type PeerSet = Set<Peer>;
type PieceSet = List<PeerSet>;

type Piece = List<Bool>;
type File = List<Piece>;
type Bitfield = List<Bool>;


// generic list functions
def Bool all(List<Bool> bools) =
    case bools {
        Nil => True;
        Cons(False, _) => False;
        Cons(True, rest) => all(rest);
    };

def Bool any(List<Bool> bools) =
    case bools {
        Nil => False;
        Cons(True, _) => True;
        Cons(False, rest) => any(rest);
```

```
33      };
34
35  def Int index<A>(List<A> l, A e) =
36      case l {
37          Nil => -1;
38          Cons(e, _) => 0;
39          Cons(x, rest) => let (Int i) = index(rest, e) in
40                           if i == -1 then -1 else i+1;
41      };
42
43  def List<A> replace<A>(List<A> l, Int index, A e) =
44      case l {
45          Nil => Nil;
46          Cons(x, rest) => if index == 0 then
47                               Cons(e, rest)
48                           else
49                               Cons(x, replace(rest, index-1, e));
50      };
51
52  def PieceSet initPieceSet(Int n) =
53      if n > 0 then
54          Cons(EmptySet, initPieceSet(n-1))
55      else
56          Nil;
57
58  def PieceSet addBitfield(PieceSet known, Peer p, Bitfield have) =
59      case known {
60      Nil => Nil;
61      Cons(s, rest) => let (Set<Peer> peers) =
62                       if head(have) then insertElement(s, p) else s in
63                       Cons(peers, addBitfield(rest, p, tail(have)));
64      };
65
66  // Add peer to list of peers who has the given piece
67  def PieceSet addPeer(PieceSet known, Peer p, Int piece) =
68      case known {
69          Cons(s, rest) => if piece < 1 then
70                               Cons(insertElement(s, p), rest)
71                           else
72                               Cons(s, addPeer(rest, p, piece-1));
73          Nil => Nil;
74      };
75
76  // rarest piece set (that the local peer is interested in)
77  def PieceSet rare(PieceSet ps, PieceSet rare, Int min, Bitfield have) =
78      case ps {
79          Nil => rare;
80          Cons(p, rest) => let (Int n) = size(p) in
```

```
81
82                          if head(have) || n > min then
83                              rare(rest, rare, min, tail(have))
84
85                          else if n == min then
86                              rare(rest, Cons(p, rare), n, tail(have))
87
88                          else
89                              rare(rest, list[p], n, tail(have));
90      };
91
92  // param n: how many blocks there are in a piece
93  def Bitfield have(List<Int> blocks, Int n) =
94      case blocks {
95          Nil => Nil;
96          Cons(i, rest) => Cons(i >= n, have(rest, n));
97      };
98
99  def Bool interest(Bitfield remote, File downloaded, Int blocks) =
100     case remote {
101         Nil => False;
102         Cons(have, rest) => if have && ~all(head(downloaded)) then
103                                 True
104                             else
105                                 interest(rest, tail(downloaded), blocks);
106     }; // XXX here proper shortcircuiting could have made the if/else obsolete!
107
108 def List<Int> priority(File f, Int i) =
109     case f {
110         Nil => Nil;
111         Cons(blocks, rest) => if any(blocks) && ~all(blocks) then
112                                 Cons(i, priority(rest, i+1))
113                             else
114                                 priority(rest, i+1);
115     };
116
117 // return index of all pieces found in ‘have‘ and not in ‘x‘:
118 def List<Int> want(Bitfield bf, Bitfield x, Int i) =
119     case bf {
120         Nil => Nil;
121         Cons(have, rest) => let (List<Int> rec) =
122                             want(rest, tail(x), i+1) in
123                             if have && ~head(x) then
124                                 Cons(i, rec)
125                             else
126                                 rec;
127     };
128
```

```
129 // XXX want is sorted
130 def List<Int> rarest(List<Int> want, PieceSet ps, List<Int> rare, Int min, Int
131     case want {
132         Nil => rare;
133         Cons(n, rest) => if i < n then
134                              rarest(want, tail(ps), rare, min, i+1)
135                          else
136                              let (Int s) = size(head(ps)) in
137                              if s < min || s == -1 then
138                                  rarest(rest, tail(ps), list[n], s, i+1)
139                              else if s == min then
140                                  rarest(rest, tail(ps), Cons(n, rare), min, i+1
141                              else
142                                  rarest(rest, tail(ps), rare, min, i+1);
143     };
144
145 def Maybe<Int> findStrict(List<Int> pieces, Bitfield bf) =
146     case pieces {
147         Nil => Nothing;
148         Cons(i, rest) => if head(bf) then
149                              Just(i)
150                          else
151                              findStrict(rest, tail(bf));
152     };
153
154 def Bool complete(File f) =
155     case f {
156         Nil => True;
157         Cons(blocks, rest) => if all(blocks) then
158                                   complete(rest)
159                               else
160                                   False;
161     };
162
163
164 interface Peer {
165     Unit handshake(Peer caller, Bool init);
166     Unit bitfield(Peer caller, Bool init, Bitfield have);
167     Unit interested(Peer caller);
168     Unit uninterested(Peer caller);
169     Unit choke(Peer caller);
170     Unit unchoke(Peer caller);
171     Unit request(Peer caller, Int blockID);
172     Unit piece(Peer caller, Int blockID);
173     Unit have(Peer caller, Int pieceNum);
174 }
175
176 class Peer(State state, List<Peer> init, Int pieces, Int blocks) implements Pee
```

```
177     File downloaded = if state != Seed then
178                         let (Piece p) = copy(False, blocks) in
179                         copy(p, pieces)
180                       else Nil;
181     Bitfield have = copy(state == Seed, pieces);
182     PieceSet available = initPieceSet(pieces);
183
184     Map<Peer, PeerInfo> peers = EmptyMap;
185     Map<Peer, List<Int>> active = EmptyMap; // queue of requests pr remote peer
186
187     Unit run() {
188         while (init != Nil) {
189             head(init)!handshake(this, True);
190             init = tail(init);
191             // TODO other name for 'init'?
192         }
193     }
194
195     Unit handshake(Peer caller, Bool initial) {
196         PeerInfo info = Info(copy(False,pieces), True, False, True, False);
197         peers = put(peers, caller, info);
198
199         if (initial)
200             caller!handshake(this, False);
201         else
202             caller!bitfield(this, True, have);
203     }
204
205     Unit bitfield(Peer caller, Bool initial, Bitfield bf) {
206         if (state != Seed) {
207             PeerInfo info = lookupUnsafe(peers, caller);
208             Bool interesting = interesting(info);
209
210             if (interest(bf, this.downloaded, blocks)) {
211                 interesting = True;
212                 caller!interested(this);
213             }
214
215             PeerInfo update = case info {
216                 Info(_, x, _, y, z) => Info(bf, x, interesting, y, z);
217             };
218
219             peers = put(peers, caller, update);
220             available = addBitfield(available, caller, bf);
221
222         }
223
224         if (initial)
```

```
225             caller!bitfield(this, False, have);
226     }
227
228     Unit interested(Peer caller) {
229         // we must've been introduced if you're interested in me
230         PeerInfo info = lookupUnsafe(peers, caller);
231         Bool choked = choked(info);
232
233         if (size(keys(active)) < 4) {
234             choked = False;
235             caller!unchoke(this);
236
237         } // TODO else????
238
239         PeerInfo update = case info {
240             Info(bf, x, y, _, _) => Info(bf, x, y, choked, True);
241         };
242
243         peers = put(peers, caller, update);
244     }
245
246     Unit uninterested(Peer caller) {
247         PeerInfo info = lookupUnsafe(peers, caller);
248
249         // TODO update active set of peers
250
251         PeerInfo update = case info {
252             Info(bf, x, y, z, _) => Info(bf, x, y, z, False);
253         };
254
255         peers = put(peers, caller, update);
256     }
257
258     Unit choke(Peer caller) {
259         PeerInfo info = lookupUnsafe(peers, caller);
260
261         // TODO update active set of peers
262
263         PeerInfo update = case info {
264             Info(bf, x, y, _, z) => Info(bf, x, y, True, z);
265         };
266
267         peers = put(peers, caller, update);
268         // TODO
269         // when choking, take away all request by that peer!
270     }
271
272     Unit unchoke(Peer caller) {
```

```
273         // TODO egen branch for Endgame mode
274         PeerInfo info = lookupUnsafe(peers, caller);
275         PeerInfo update = case info {
276             Info(bf, _, x, y, z) => Info(bf, False, x, y, z);
277         };
278 
279         peers = put(peers, caller, update);
280 
281         this.getFrom(caller);
282     }
283 
284     // find something to request from peer.
285     // assume we are allowed to (not choked).
286     Unit getFrom(Peer peer) {
287         PeerInfo info = lookupUnsafe(peers, peer);
288         Bool interesting = interesting(info);
289 
290         List<Int> priority = priority(downloaded, 0);
291         Maybe<Int> piece = findStrict(priority, bf(info));
292 
293         if (piece == Nothing) {
294             List<Int> want = want(bf(info), have, 0);
295 
296             if (state == Leecher) {
297                 want = rarest(want, available, Nil, -1, 0);
298             }
299 
300             if (want != Nil) {
301                 Int rand = random(length(want));
302                 piece = Just(nth(want, rand));
303             }
304         }
305 
306         if (piece != Nothing) {
307             interesting = True;
308 
309             if (~choking(info)) {
310                 Int i = fromJust(piece);
311                 Int b = index(nth(downloaded, i), False);
312                 Int block = i*blocks + b;
313 
314                 peer!request(this, block);
315                 // should also keep track of outgoing requests
316 
317             } else {
318                 peer!interested(this);
319             }
320 
```

```
321         } else {
322             interesting = False;
323             peer!uninterested(this);
324         }
325 
326         PeerInfo update = case info {
327             Info(bf, x, _, y, z) => Info(bf, x, interesting, y, z);
328         };
329         peers = put(peers, peer, update);
330     }
331 
332     Unit request(Peer caller, Int blockID) {
333 
334         if (~choked(lookupUnsafe(peers, caller))) {
335             List<Int> q = lookupDefault(active, caller, Nil);
336 
337             active = put(active, caller, appendright(q, blockID));
338 
339             await choked(lookupUnsafe(peers, caller))
340                || let (List<Int> l) = lookupDefault(active, caller, Nil) in
341                   if isEmpty(l) then True else head(l) == blockID;
342 
343             q = lookupDefault(active, caller, Nil);
344 
345             if (~isEmpty(q)) {
346                 caller!piece(this, blockID); // TODO await this!
347                 active = put(active, caller, tail(q));
348             }
349         }
350     }
351 
352     Unit piece(Peer caller, Int blockID) {
353         Int pi = blockID / blocks;
354         Int bi = blockID % blocks;
355         Piece blocks = nth(downloaded, pi);
356 
357         blocks = replace(blocks, bi, True);
358         downloaded = replace(downloaded, pi, blocks);
359 
360         // if that was the last block, announce have to all
361         if (all(blocks)) {
362             have = replace(have, pi, True);
363 
364             Set<Peer> ps = keys(peers);
365 
366             while (hasNext(ps)) {
367                 Peer peer = take(ps);
368                 peer!have(this, pi);
```

```
369
370                 ps = fst(next(ps));
371             }
372         }
373
374         if (complete(downloaded)) {
375             state = Seed;
376         }
377
378         // if endgame, cancel any other request for the same block
379         // else if not choking, request more!
380         if (state == Init || state == Leecher) {
381             if (~choking(lookupUnsafe(peers, caller)))
382                 this.getFrom(caller);
383         }
384     }
385
386     Unit have(Peer caller, Int pieceNum) {
387         if (state != Seed) {
388             // 'have' is broadcast to everyone,
389             // may arrive before handshake is done:
390             await lookup(peers, caller) != Nothing;
391
392             available = addPeer(available, caller, pieceNum);
393
394             PeerInfo info = lookupUnsafe(peers, caller);
395             Bool intr = interesting(info);
396             Bitfield bf = replace(bf(info), pieceNum, True);
397
398             if (~intr) {
399                 if (interest(bf, downloaded, blocks)) {
400                     intr = True;
401
402                     if (choking(info)) {
403                         caller!interested(this);
404                     } else {
405                         this.getFrom(caller);
406                     }
407                 }
408             }
409
410             PeerInfo update = case info {
411                 Info(_, x, _, y, z) => Info(bf, x, intr, y, z);
412             };
413
414             peers = put(peers, caller, update);
415         }
416
```

```
417         }
418
419     }
```

Appendix E

Additional code listings

E.1 Alternative XOR function

```
// Returns b raised to the nth power
//
def Int pow(Int b, Int n) =
    if n < 0 then
        0
    else case n {
        0 => 1;
        _ => b * pow(b, n-1);
    };

// Extract the value of the nth (1−indexed) bit of x
// (return value is 0 or a power of 2).
//
def Int nthBitValue(Int x, Int n) =
    x % pow(2, n) - x % pow(2, n-1);

def Int xorLoop(Int a, Int b, Int n) =
    if n == 0 then
        0
    else
        let (Int bA) = nthBitValue(a, n) in
        let (Int bB) = nthBitValue(b, n) in
        let (Int bX) = if bA == bB then 0
                       else bA in
        bX + xorLoop(a, b, n-1);

def Int xor(Int a, Int b) =
    xorLoop(a, b, suffix(a,b));
```