JAIST Repository
https://dspace.jaist.ac.jp/

Title	Actor-based Protocol Composition Framework for Rapid Prototyping
Author(s)	Rattanaponglekha, Nuttapong
Citation	
Issue Date	2011-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/9926
Rights	
Description	Supervisor:Associate Professor Xavier Defago, 情報科学研究科, 修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Actor-based Protocol Composition Framework for Rapid Prototyping

By Nuttapong Rattanaponglekha

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Défago

September, 2011

Actor-based Protocol Composition Framework for Rapid Prototyping

By Nuttapong Rattanaponglekha (0910210)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Xavier Défago

and approved by
Associate Professor Xavier Défago
Associate Professor Toshiaki Aoki
Associate Professor Yasushi Inoguchi

August, 2011 (Submitted)

Copyright © 2011 by Nuttapong Rattanaponglekha

Abstract

Nowadays, the development of distributed systems and applications are larger and more complex. In one application, we have many protocols inside that have interactions together. Protocol developers not only consider their own protocol but they also have deep knowledge in other protocols. In each protocol, there are many hindrances, complexity that distract the developing application, prototyping the algorithm and evaluating the performance in each algorithm. The thesis has contribution in protocol composition framework. The protocol composition framework should provide facilities to make protocol programming and protocol composition easier. The idea of protocol composition framework is separated into 2 views, *composer* and *protocol programmer*. Composers no need to have deep knowledge of the composed protocol that includes details such as the cumulative state of the protocols to protect, or the handlers in which new threads are launched. On the other hand, protocol programmers only consider their own protocol logic, have no deep knowledge in other protocols. Protocol composition framework should provide facilities to make protocol programming and protocol composition easier.

Our protocol composition framework is based on actor model. By using actor model, we target to loose decoupling between protocols as much as possible and make algorithm code be more compact and expressive. We implement some protocols in agreement problems such as consensus algorithm, atomic broadcast algorithm. We use *boilerplate code* and *effective code* as criteria, compare the protocols on our framework with other framework and pseudocode. In the evaluation, we evaluate the expressiveness of our protocol in terms of line of code. We express the analysis of writing style compared with pseudocode and other framework. In terms of performance overhead, we evaluate the initialization time, the execution time and availability.

Acknowledgements

First of all, I would like to express my gratitude to my supervisor, Associate Professor Xavier Défago for all guidance and knowledge in both of academic term and non-academic term. Professor Xavier has taught me so many things, especially in how to organize the presentation that I had never mentioned it before.

I would also like to thank Associate Professor Toshiaki Aoki and Associate Professor Yasushi Inoguchi for taking time to serve on my dissertation committee.

My special thank to Career Development Program for Foreign Students in JAIST and Ministry of Education Culture Sports Science and Technology Japan, who provided me financial support and Japanese language study for life in Japan.

I want to thank all members in Défago-Lab for their kindness. I especially thank to Daiki Higashihara and Shintaro Hosoai for their kindness, guidance and friendships. They gave kindness and suggestions in both of research and life in Japan.

My thanks go to Thai people in JAIST for many supports, encouragement and warm friendship during hard time, especially in writing thesis.

Last, but not least, I wish to express my gratitude to my family and close friends, especially to my parents for their loving support throughout my life. Moreover, I would like to thank my niece (named as Yaya) for making me laugh by her cute face.

Contents

1 Introduction **9**

1.1 Motivation . . . 9

1.2 Objective . . . 10

1.3 Contribution . . . 10

1.4 Organization . . . 11

2 Distributed systems & Composition framework **12**

2.1 Characteristics . . . 12

2.2 System models . . . 13

2.2.1 Failure Models . . . 13

2.3 Agreement Problems . . . 14

2.3.1 Consensus . . . 14

2.3.2 Reliable broadcast . . . 15

2.3.3 Atomic broadcast . . . 15

2.4 Composition framework . . . 15

2.4.1 Model . . . 16

2.4.2 Neko protocol composition framework . . . 16

2.5 Related work . . . 19

2.5.1 Appia . . . 19

2.5.2 Cactus . . . 20

3 Benchmark and criteria **22**

3.1 Reliable broadcast . . . 22

3.1.1 Algorithm . . . 22

3.2 Consensus algorithm . . . 22

3.2.1 Chandra-Toueg consensus algorithm . . . 23

3.2.2 Mostéfaoui and Raynal consensus algorithm . . . 23

3.2.3 Paxos consensus algorithm . . . 24

3.3 Atomic broadcast . . . 25

3.3.1 Simple fixed sequencer algorithm . . . 25

3.3.2 Simple moving sequencer algorithm . . . 25

3.3.3 Simple privilege-based algorithm . . . 26

3.3.4 Simple communication history algorithm . . . 26

3.3.5 Using consensus to solve Atomic broadcast . . . 26

3.4 Criteria . . . 26
3.4.1 Effective code . . . 27
3.4.2 Boilerplate code . . . 27

4 Problem statement **28**
4.1 Problems . . . 28
4.2 Analysis . . . 31

5 Actor-based model **32**
5.1 Introduction . . . 32
5.2 Model . . . 32
5.2.1 Actor model in Scala language . . . 33
5.2.2 Why is actor model in Scala? . . . 33
5.3 Comparing with Java thread . . . 33
5.3.1 Design and Implementation view . . . 34

6 Yaya protocol composition framework: Design **39**
6.1 Introduction . . . 39
6.2 Architecture . . . 39
6.3 Sample protocol: Pingpong protocol . . . 41

7 Yaya protocol composition framework: User manual **44**
7.1 Overview . . . 44
7.2 Implementation view . . . 49
7.3 Getting started . . . 50
7.3.1 Hello world! . . . 50
7.3.2 Creating a configuration file . . . 50
7.3.3 Implementing protocols . . . 51
7.3.4 Starting execution . . . 53
7.4 More general implementation . . . 54

8 Yaya protocol composition framework: Usage **57**
8.1 Sample application . . . 57
8.2 Startup and configuration . . . 58

9 Comparative Analysis: Expressiveness **61**
9.1 Reliable broadcast . . . 61
9.1.1 Evaluation . . . 61
9.2 Consensus . . . 65
9.2.1 Chandra-Toueg consensus algorithm . . . 65
9.2.2 Mostéfaoui and Raynal consensus algorithm . . . 67
9.2.3 Paxos consensus algorithm . . . 69
9.2.4 Evaluation . . . 71
9.3 Atomic broadcast . . . 71
9.3.1 Evaluation . . . 74

9.4 Analysis . . . 74

10 Performance Analysis :Overhead **77**
10.1 Execution overhead & Availability performance . . . 78
10.1.1 Execution overhead time . . . 78
10.1.2 Availability performance . . . 79
10.2 Discussion . . . 80

11 Conclusion & Open Questions **81**
11.1 Conclusion . . . 81
11.2 Open Questions . . . 81

A Reliable broadcast **82**

B Consensus algorithm **83**
B.1 Chandra-Toueg consensus algorithm [11] . . . 83
B.2 Mostéfaoui and Raynal consensus algorithm [11] . . . 85
B.3 Paxos consensus algorithm [8] . . . 86

C Atomic broadcast algorithm **88**
C.1 Simple fixed sequencer algorithm [10] . . . 88
C.2 Simple moving sequencer algorithm [10] . . . 89
C.3 Simple privilege-based algorithm [10] . . . 90
C.4 Simple communication history algorithm [10] . . . 91
C.5 Using Consensus to solve Atomic broadcast [9] . . . 92

List of Figures

2.1 Original Architecture of Neko framework [7] 17
2.2 Present Architecture of Neko framework [7] 18

3.1 Example run of the Chandra-Toueg consensus algorithm 23
3.2 Example run of the Mostéfaoui and Raynal consensus algorithm 24
3.3 Example run of the Paxos consensus algorithm 25

5.1 Producer Consumer Application in Java Concurrency model 35
5.2 Producer Consumer Application in Scala actor model 37
5.3 Mailbox in buffer actor 38

6.1 Architecture of Yaya 40
6.2 Yaya message sending mechanism 41

7.1 Simulated model in Yaya protocol composition framework 44
7.2 Component module in a process 45
7.3 Communication within a process 45
7.4 Communication using message within a process 46
7.5 Structure of message in Yaya framework 46
7.6 YProcess class diagram 47
7.7 Class diagram of protocol in Yaya framework 47
7.8 Class diagram of message in Yaya framework 48
7.9 Difference between composers and protocol programmers 49
7.10 Systems model from *helloWorld* configuration file 50
7.11 Configuration file syntax 51
7.12 Protocol class syntax 52
7.13 Using of createMessage method 53
7.14 Using of sendTo method 53
7.15 Hello World program result 53
7.16 Composing steps for *composers* 54
7.17 Restriction for *composers* 55
7.18 Protocol implementing steps for *protocol programmers* 55
7.19 Restriction for *protocol programmers* 56

8.1 Architecture of a sample Yaya application: Atomic broadcast application . . 58

10.1 Application model for evaluation performance overhead 77
10.2 Execution overhead comparison between simulation in Yaya and Neko . . . 78
10.3 Availability comparison between simulation in Yaya and Neko 79

List of Tables

9.1 Comparative between implementation of Reliable broadcast on Neko and Yaya in terms of line of code . . . 65
9.2 Comparative between implementation of consensus algorithm on Neko and Yaya . . . 71
9.3 Summary of comparative between implementation of algorithm on Neko and Yaya . . . 75
9.4 Comparative between implementation of Atomic broadcast algorithm on Neko and Yaya . . . 76

Listings

2.1 Example of a Neko configuration file (for a simulation) 19
3.1 Example of boilerplate code 27
4.1 Each round messages are processed according to the message type 28
4.2 Estimate phase processing code 29
4.3 Failure detector handler code (A) 30
4.4 Failure detector handler code (B) 30
4.5 Chandra-Toueg consensus interface 31
5.1 Producer.java 34
5.2 Consumer.java 35
5.3 Producer.scala 36
5.4 Consumer.scala 36
5.5 UnboundedBuffer.scala 37
6.1 Pingpong code example implemented on Yaya 42
7.1 *helloWorld.config* configuration file for Hello world program (1) 50
7.2 *helloWorld.config* configuration file for Hello world program (2) 50
7.3 *helloWorld.config* configuration file for Hello world program (3) 51
7.4 Example of protocol class declaration 51
7.5 Example of overriding launch method 52
8.1 Configure file example 59
9.1 *Send* method in reliable broadcast on Neko 62
9.2 Pseudocode of *send* method in Reliable broadcast 62
9.3 deliver method in reliable broadcast on Neko 63
9.4 Pseudocode of *deliver* method in Reliable broadcast 63
9.5 contentMatch method in reliable broadcast on Yaya 64
9.6 Start executing Chandra-Toueg consensus algorithm on Yaya 65
9.7 phase 2 of Chandra-Toueg algorithm on Yaya 66
9.8 suspect content handling on Yaya 66
9.9 Pseudocode of Phase 3 in Chandra-Toueg consensus 67
9.10 Example code of Mostéfaoui and Raynal consensus algorithm on Neko (1) . 67
9.11 Example code of Mostéfaoui and Raynal consensus algorithm on Neko (2) . 68
9.12 Example code of Mostéfaoui and Raynal consensus algorithm on Yaya . . . 68
9.13 Example code of Paxos consensus algorithm on Neko 69
9.14 Example code of Paxos consensus algorithm on Yaya 70
9.15 Pseudocode of *Read* and *Write* content in Paxos consensus algorithm . . . 70

9.16 Example code of Atomic broadcast using consensus on Neko (1) 72
9.17 Example code of Atomic broadcast using consensus on Neko (2) 72
9.18 Example code of Atomic broadcast using consensus on Yaya 73
9.19 Pseudocode of Atomic broadcast using Consensus 74

Chapter 1

Introduction

1.1 Motivation

In distributed systems, the complexity [1] of systems and the expressiveness of protocol mechanism are hindrances to design and implementation of distributed systems. In distributed systems, there is no a central point of control with direct access to the states of all processes, therefore decisions often require an exchange of information and complex communication between the processes. The coordination of distributed processes requires complex message-based protocols for synchronization and communication. In the implementation of communication, there is a gap between application programs and the lower-level hardware and software infrastructure in order to coordinate how parts of applications are connected and how they interoperate. The application programmers need to know about the infrastructure of the systems. Moreover, in run-time, there is a tremendous number of messages that are passed in the systems. Programmers need to consider which message will be delivered and how is the responding of that message. This hindrance makes error-prone part of code, unexpected state and exception in the systems. These obstacles impede the engineering of large system.

We provide *middleware* of distributed systems. *Middleware* helps application programmers to concentrate on writing their application logic rather than the low-level of system mechanisms. Application programmers concentrate only on interface of middleware. *Middleware* can help to shield software developers from low-level, tedious, and error-prone platform details, such as socket-level network programming. *Middleware* also provide a consistent set of higher-level network-oriented abstractions. However, *middleware* is not flexible, we have no configuration of protocols stack . In terms of configurability, we face difficulties to implement a specify semantic property and construct services that are customized to the needs of the application.

For configurability [2], we provide *composition framework* for constructing protocols stack in the systems. *composition framework* requires that top-level and bottom-level interfaces of the protocols be identical for each layer, so they can be stacked on top of each other.

The purpose of a *composition framework* is to ease the development of custom protocol stacks [3] [4] [5]. In composition framework, the software is developed by different people, *composers* and *protocol programmers*. *Composers* construct their own protocols by composing the *microprotocols* [6] and making the interactions between them. *Protocol programmers* consider on their own protocol logic. However, there are many unnecessary components, complicated parts and distracted code from the main mechanisms. For example, in concurrency model restriction [6], concurrency issues, such as protecting the states of protocols against concurrent changes and avoiding deadlocks, cannot always be solved by the protocol programmers on their own: the composer must be involved. In order to do this, the composer needs to have deep knowledge of the composed protocols that includes details such as the cumulative state of the protocols to protect, or the handlers in which new threads are launched.

The complexity of a distributed system is a hindrance for designing and implementing application in distributed systems. We use a middleware to shield software developers from low-level, tedious, and error-prone platform details. However, we face difficulties to construct the protocols stack in systems and define the interactions between protocols. We provide composition framework to ease the development of custom protocol stack. In composition framework, we have some hindrances to express the protocol mechanism and define the interactions between the protocols.

1.2 Objective

In this project, we target on *rapid prototyping* by making coding syntax be easy to read, understand (proof, debugging) and also implement protocols. We consider how to make code be more compact and condense and also remove *boilerplate code* as much as possible (expressiveness, conciseness). We aim at reducing the *boilerplate code* to make a code better structured and more clear view between composers and protocol programmers. Composers no need to have deep knowledge of the composed microprotocols [6] that includes details such as the cumulative state of the microprotocols to protect, or the handlers in which new threads are launched. Protocol programmers consider only on their protocol logic and no need to provide special messages or special method for other protocol. Protocol programmers implement their own code that seem to resemble *pseudo-code* of their algorithm.

1.3 Contribution

We present a protocol composition framework based on an actor model. We leverage the use of the Scala language to allow for a significantly more compact and natural expression of communication protocols than with equivalent Java systems. We analyze our protocol composition framework compared with composition framework in Java concurrency model in terms of expressiveness. Moreover, we measure performances of our protocol

composition framework (e.g., initialization overhead, availability).

1.4 Organization

In chapter 2, we explain about the background and related work of this research and explain our benchmarks(algorithms in agreement problem) and criteria in Chapter 3. We address the problems of the research in chapter 4. In chapter 5, we describe the actor model which is used to implement our prototype of composition framework. In chapter 6 and 8, we propose our composition framework prototype, explaining architecture and usage of our framework. In chapter 9, we evaluate and analyze expressiveness of our framework comparing with Neko in terms of line of code and we then evaluate the performance overhead of our framework in chapter 10. Finally, in chapter 11, we conclude our research and describe about continuing research part.

Chapter 2

Distributed systems & Composition framework

Distributed systems consist of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

2.1 Characteristics

In distributed systems, there are multiple autonomous components which are not shared by all users and some resources in systems may not be accessible. Software runs in concurrent processes on different processors, so there are multiple point of control and multiple point of failure. Common characteristics are following:

Resource sharing One process can be able to use any hardware, software, data anywhere in the system. So, resource manager needs to control access, provide naming scheme and control concurrency.

Openness Openness is concerned with extensions and improvements of distributed systems. New components have to be integrated with existing components. Systems should easily interoperate.

Concurrency Components in distributed systems are executed in concurrent processes. Components need to access and update shared resource (e.g. variables, databases, device drivers). If concurrent updates are not coordinated, integrity of the system may be violated (e.g. lost updates, inconsistent data).

Scalability Distributed systems need scalability to accommodate more users and respond requests faster by not changing the components when scale of a system increases.

Fault Tolerance In distributed system, there are multiple point of failure, components (e.g. processes, channels) may fail. It is important to specify the assumptions on failures, including expected semantics of failures.

Transparency Distributed systems should be perceived by users and application programmers as a whole rather than as a collection of cooperating components.

2.2 System models

Distributed systems are modeled as a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$ that interact by exchanging messages over communication channels [11]. There exist a number of models that restrict the behavior of components (e.g. processes, devices, data, network) in distributed systems. Models have two uses:

Abstract models [11] Distributed algorithms are designed for a model of the environment. In other words, they are only guaranteed to work in a certain environment if the assumptions of the model hold in that environment. It is desirable that such models be (1) general, to allow algorithm designed for the model to work in a variety of environments, and (2) simple, to allow a simple expression for algorithms, and easy correctness proofs.

Descriptive models [11] Models can provide a description of existing environments. Their main use is analyzing the performance of distributed algorithms. Compared to abstract models, this kind of model is usually detailed and complex, to allow accurate performance estimates, though the level of detail and complexity depends on the goals of the performance study.

2.2.1 Failure Models

In distributed systems, the components, both processes and channels, may fail. It is important to determine how the correctly system can protect itself against failure. The type of fault models are following:

Crash model A process is said to crash if it executes its local algorithm correctly up to some moment, and then does not execute any step thereafter.

Byzantine behavior A process is said to Byzantine if it executes arbitrary steps that are not in accordance with its local algorithm. In particular, a Byzantine process may send messages with an arbitrary content.

2.3 Agreement Problems

Agreement problems are a fundamental class of problems in distributed systems. In distributed systems, there are possibilities that the failure will occur in processes or channels. Therefore, participating processes need to make a decision and must agree on some common decision with the same.
In this section, we give definitions for three agreement problems: consensus, reliable broadcast and atomic broadcast

2.3.1 Consensus

Consensus problem(defined in [11]) is a main problem in agreement problems that many problems in agreement can be reduced into consensus problem. Each process will propose its own value, and then each of them will receive the same decision value, which is one of the proposed values. Consensus algorithms are used to guarantee that a collection of processors carry out identical computations, agreeing on the results of some critical steps. It allows the processors to tolerate the failure of some processors.
In the consensus problem, each process starts with an initial value from a fixed set V, and must eventually reach a common and irrevocable decision from V. More formally, the consensus problem is specified as follows [11]:

Validity If a process decides v, then v was proposed by some process.

Agreement No two correct processes decide differently.

Integrity Every process decides at most once.

Termination All correct processes eventually decide.

The agreement condition of consensus may sound odd because it allows two processes to disagree even if one of them fails a very long time after deciding. Clearly, such disagreements are undesirable in many applications since they may lead the system to inconsistent states. This is why one introduces a strengthening of the agreement condition, called the *uniform agreement* condition, which precludes any disagreement even due to faulty processes:

Uniform agreement No two processes decide differently.

Besides consensus, a lot of group communication problems have a uniform and a non-uniform variant (e.g., reliable broadcast and atomic broadcast). The non-uniform variant allows incorrect processes to take actions (just before they crash) which might never be taken by correct processes. The application developer must consider the consequences of such actions when deciding which variant to use. The developer should keep in mind that ensuring uniformity often has a cost in terms of performance.

2.3.2 Reliable broadcast

Reliable broadcast(defined in [11]) is used to ensure that all processes will deliver a message broadcast previously, even though the message losses occur. More formally, reliable broadcast is defined by two primitives R-broadcast(m) and R-deliver(m) where m is some message. Reliable broadcast is specified as follows [11]:

Validity If a correct process R-broadcasts m then it eventually R-delivers m.

Integrity For any message m, every process R-delivers m at most once, and only if m was R-broadcast by some process.

Agreement If a correct process R-delivers m then all correct processes eventually R-deliver m.

Uniform Agreement If a process R-delivers m then all correct processes eventually R-deliver m.

2.3.3 Atomic broadcast

Atomic broadcast(defined in [11]) is an extension to reliable broadcast: beside ensuring that all processes receive the messages, it also ensures that processes receive the messages *in the same order*. Atomic broadcast is defined by two primitives A-broadcast(m) and A-deliver(m) where m is some message. Atomic broadcast is specified with the properties of reliable broadcast and the following property [11]:

Total Order For any two correct processes p and q, if p A-delivers a message m' after message m then q A-delivers m' only after A-delivered m.

Uniform atomic broadcast is specified with the properties of uniform reliable broadcast reliable broadcast and the following property:

Uniform Total Order For any two processes p and q, if p A-delivers a message m' after message m then q A-delivers m' only after A-delivered m.

2.4 Composition framework

The idea of composition framework is to encourage developers to partition complex protocols into simple protocols, each of which is implemented by a protocol layer.

In distributed systems, there exist a tremendous number of message passing, protocols and communications. Effort to bring structure to all development in distributed systems have been only partially successful. The lack of structure impedes the engineering of large,

complex distributed systems. We have a *protocol stack* [3] as middleware infrastructure that provides a service to simplify the development of distributed applications. However, the development of protocol stacks is not easy, customizing the protocol stack is also complex. The purpose of a *protocol composition framework* is to ease the development of custom protocol stacks.

2.4.1 Model

In this section, we classify protocol composition frameworks mechanisms into three models.

Composition model In composition model, we have to specify how protocol modules are arranged when they are composed. It can be hierarchical or cooperative. In hierarchical model, protocol modules are arranged in protocol stack. Each protocol module can only communicate with next layer. In cooperative model, we have no hierarchy, every protocol communicate with others like graph. Moreover, some complex models (e.g., a hybrid approach) are also possible.

Interaction model In interaction model, we have to define the way protocol modules may interact and exchange information. It may be event-driven or it may be message-passing.

Concurrency model In concurrency model, we have to define whether and how concurrency is allowed in the framework. If some concurrency is allowed, the model should also specify how to synchronise concurrent threads. If no concurrency is allowed, application programmers need to consider how to solve problems from outside composition.

2.4.2 Neko protocol composition framework

Neko framework [7] is one of protocol composition framework. Neko is a simple communication platform that allows developer to both simulate a distributed algorithm an execute it on a real network, using the *same implementation* for the algorithm. Neko is written in Java and is thus highly portable because it could be run on any machines that have virtual machine. It has been deliberately kept simple, extensible and easy to use.

Architecture

The architecture of Neko consists of two main parts: *application framework* and *networks*. At the level of application, every processes in system (numbered from 0 to $n - 1$, n is amount of processes in the system) communicate using a simple message passing interface: a sender process pushes its message onto network with the primitive *send*, and the network then pushes that message onto the receiving process with *deliver*. In each process, application layer is programmed as multi-layered programs.

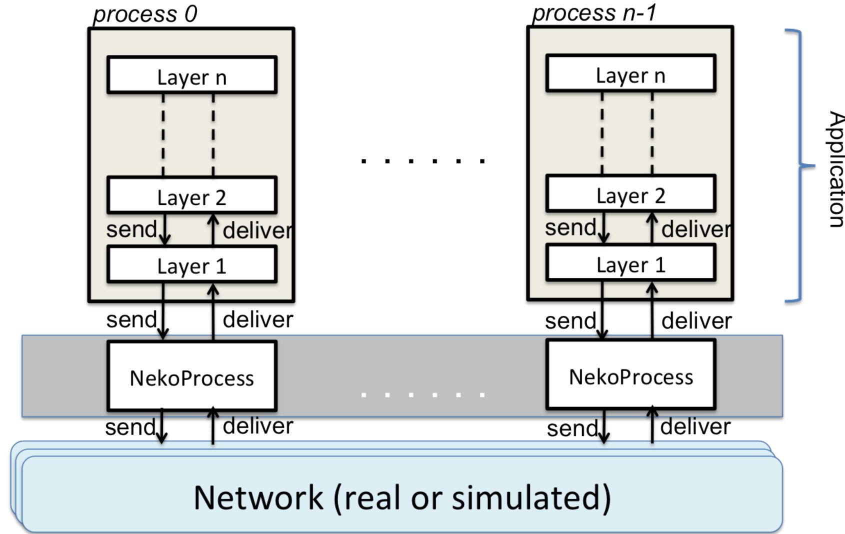

Figure 2.1: Original Architecture of Neko framework [7]

Application layers In Figure 2.1, we show the original architecture of Neko Framework. Neko applications are constructed as a *hierarchy of layers*. Messages are passed down from above protocol using *send*, and delivered messages are passed up from network using *deliver* method. Layers are either *active* or *passive*. In passive layers, there are *no* own thread, the below layer will call deliver method. In active layers, there are own thread, they actively pull messages from the below layer by using *receive*. The call to *receive* blocks until a message is available. Messages are delivered to message queue and then they get messages by *receive* orderly. However, in the present architecture as shown in Figure 2.2, Neko applications are constructed as a *graph of protocols*. Messages are passed down from the *sender thread* [6] using *send* until they reach destinations. Messages are passed up by *dispatcher* using *deliver* method. We can implement both of *active protocol* (protocols that have their own thread) and passive protocol (protocol that have no their own thread).

NekoProcess Each process of the distributed application has an associated object of type NekoProcess, placed between the layers of the application and the network. The NekoProcess takes several important roles:

1. It holds some information, e.g., the address of the process. All layers can access to process to get these information.

2. It implements some generally useful services, such as sent and received logging messages by the process.

NekoMessage All communication primitives (send, deliver and receive) transmit instances of NekoMessages. Every message is composed of a content part that consists of any Java object, and a header with the following information:

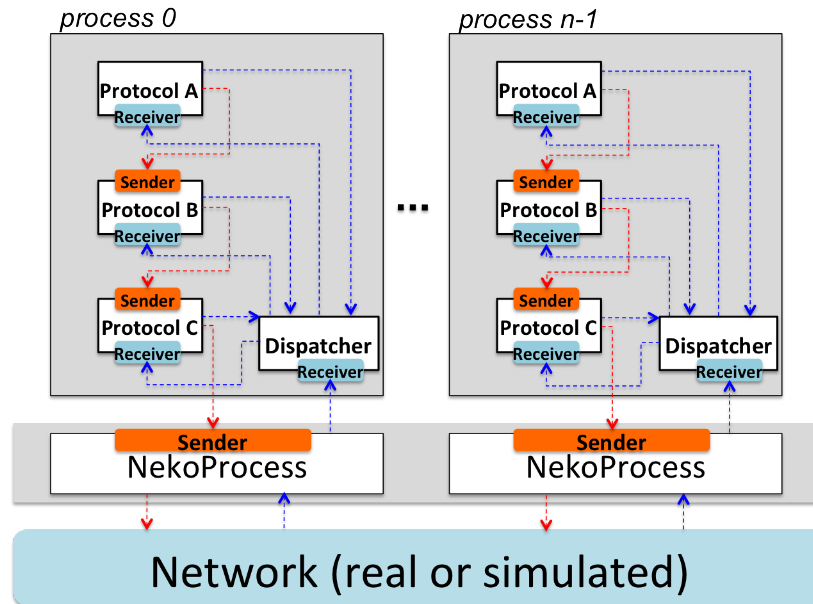

Figure 2.2: Present Architecture of Neko framework [7]

1. **Addressing (source, destinations)** The addressing information consists of sender address process and destination address processes. The numbering of processes starts from 0 to $n-1$ which n is number of processes in system.

2. **Network** When Neko manages several networks in parallel, each message carries the identification of the network that should be used for transmission.

3. **Message type** Each message has a user-defined type field (integer). It can be used to distinguish messages belonging to different protocols.

Startup and Configuration

Configuration In *Neko composition framework*, all aspects (e.g. number of processes, network type) are configured by a single file. The name of this file is passed as a parameter to Neko on startup. Neko ensures that each process of the application has the information in the configuration file when the application starts. An example of a Neko configuration file is shown in Listing 2.1

```
1  simulation = true
2  process.num = 2500
3  process.initializer = test.nProcessesOverhead.PingPongInitializer
4  network = lse.neko.networks.sim.RandomNetwork
5  RandomNetwork.lambda = 0.0
6  handlers = java.util.logging.FileHandler, java.util.logging.
       ConsoleHandler
7  java.util.logging.FileHandler.pattern = test/nProcessesOverhead/log.log
8  messages.level = FINE
```

Listing 2.1: Example of a Neko configuration file (for a simulation)

Bootstrapping In *Neko composition framework*, we have two types of bootstrapping, a simulation and a distributed execution. In this research, we concentrate only on a simulation. Only one Java Virtual Machine (JVM) is needed with the name of configuration file, and Neko will create and initialize all processes. the number of processes is specified by the entry process.num.

Initialization After we finished specifying a number of processes, we also need to specify the process initializer and the networks in the simulations. The process initializer is defined in configuration file by the entry process.initializer. The entry process.initializer defines a class of initializer that is used to initialize each process. The network is specified by the entry network.

Execution Once all the initialization is finished, all NekoThreads are started and the application begins executing. The application can access to the entries of the configuration file.

Shutdown Neko provides a shutdown function that any process can call and which results in shutting down all processes. Processes may implement more complex termination algorithms that end with calling the shutdown function. The termination algorithm executed by the shutdown function exchanges messages on the control network.

2.5 Related work

2.5.1 Appia

Appia [17], a protocol composition framework that offers a clean and elegant way for application to express interchannel constraints. This feature is obtained as an extension to functionality provided by current systems. Thus, *Appia* retains the flexible and modular design that has previously proven to be advantageous, allowing communication stacks to be composed and reconfigured in run-time.

Composition model [5], In Appia, a protocol module is defined as a pair *layer-session*. The layer defines three sets of events, namely, events *accepted* by the protocol module,

events *provided*, and events *required* for proper operation. The session contains a private state and the protocol code (structured as a set of event handlers).

Interaction model [5], The interaction model among protocol sessions is event-driven. Events are triggered by instantiating the event class, and providing three parameters: the channel, the *source* session (i.e., the session that is triggering the event), and a *direction* (either upwards or downwards). These three parameters define the route of the event (i.e., the sequence of sessions). If a protocol layer did not declare a given event class as "accepted", its companion session will not be put into the event's route. Thus, all occurrences of that event will bypass this session. A session forwards an event occurrence to the next session in the event's route by calling the event's method *go()*. Event occurrences convey data inside. Data sharing among sessions in not possible.

Concurrency model [5], All event occurrences in all channels in the stack are processed by only one single thread: the event scheduler thread. The main advantage of this single-threaded model is the absence of racing conditions inside sessions' code, i.e., protocol developers never have to worry about thread synchronization. All event occurrences are put into event scheduler's queue. In each step, the event scheduler pops the first event event occurrence e from the event queue, looks up the next session that handles e, and executes the corresponding event handler. If the session creates a new event occurrence $\acute{e}$ in the same direction as e, $\acute{e}$ is inserted in the event queue immediately after e.

2.5.2 Cactus

Cactus [18] is a system for constructing configurable network protocol and services where each service property or functional component is implemented as a separate software module called a *micro-protocol* [19]. A customized instance of a service is then created by choosing a collection of micro-protocols based on the properties to be enforced, and configuring then together with the Cactus runtime system to form a *composite protocol* that implements the service. A micro-protocol is structured as a collection of *event handlers* that are executed when a specified event occurs.

Composition model [5] Cactus defines a *two-level* composition model: coarse grain and fine grain. The coarse grain protocols, called *composite protocols*, are composed by defining a hierarchical graph. In the graph, several composite protocols may be placed at the same level. The fine grain protocols, called *micro-protocols*, are arranged in cooperative way (no hierarchy) and interacting via event triggering and data sharing. Micro-protocols cannot exist on their own in the hierarchy graph: they need to be within a composite protocol.

Interaction model [5]

1. **Composite Protocols** Composite protocols are linked by edges that define an acyclic oriented graph. Only composite protocols that are linked by an edge in

this graph can directly interact. Composite protocols are instantiated at run-time yielding *sessions*. Each composite protocol session maintains its own state and its own micro-protocols. A session can *open* a new session of a composite protocol to which it is linked. The communication between two sessions is by *message* passing. In Cactus, a message is a data structure consisting of a set of *attributes*. Attributes are tuples of the form (*tag*, *scope*, *value*). *Tags* are used to retrieve the attribute's *values*, which are the actual data. The *scope* restrains the attribute's visibility. A session may send a message up or down to the next session in the graph. When sending a message up, if there are several sessions that may receive the message, a *demux* function must be provided in order to decide which is the receiving session.

2. **Micro-Protocols** Micro-protocol execution is event-driven: micro-protocols are structured as a set of event handlers and contain private state. Event handlers' code, may modify this state and trigger other events.

Concurrency model [5] A composite protocol, i.e., all the micro-protocols it contains, are idle as long as no message arrives from above/below. Message arrivals start out the activity in a composite protocol. At that point, the framework triggers an occurrence of the predefined event called "*message has arrived*": all handlers of micro-protocols bound to this event will be executed one after the other. The handler code may read/modify the private state in the micro-protocol as well as the shared state in the composite protocol, and may trigger other events. There are two ways to trigger an event: *raising* and *invoking*. *Raising* is asynchronous (a thread is spawn or reused from a thread pool), while *invoking* is synchronous (function call semantics). In the *raising* scheme, the private state of a micro-protocol may be exposed to racing conditions. So, it is necessary to synchronise the access to this private state (as well as to the micro-protocols' shared state). Such a synchronisation is not enforced by Cactus; it is the responsibility of the protocol programmer to enforce synchronisation using standard mechanisms (mutex, monitors, semaphores).

Chapter 3

Benchmark and criteria

In this chapter, we describe about algorithms that are used as benchmark to evaluate our framework. We also define our criteria in terms of *boilerplate code*.

3.1 Reliable broadcast

Reliable broadcast is a communication primitive in distributed systems. A reliable broadcast is defined by the following properties:

1. **Validity** if a correct process sends a message, then correct process will eventually deliver that message.

2. **Agreement** if a correct process delivers a message, then all correct process eventually deliver that message.

3. **Integrity** every correct process delivers the same message at most once and only if that message has been sent by a process.

3.1.1 Algorithm

The message that is sent by reliable broadcast will receive at most once time. If process receive the message at first time, process will send that message to other destinations. The pseudocode is shown in appendix A.

3.2 Consensus algorithm

Consensus is agreement problem in distributed systems that encapsulates the task of group agreement. There are many algorithms for solving consensus problem.

3.2.1 Chandra-Toueg consensus algorithm

Chandra-Toueg [9] algorithm is based on a rotating coordinator. Each process executes rounds numbered 1, 2, etc. Throughout the rounds each process maintains an estimate of the decision value. Each round is split into four phases :

- **Phase 1** Every process sends its estimate with the updated round number to the current leader.
- **Phase 2** The coordinator waits for a majority of estimates, selects the most recent estimate and sends it to all processes.
- **Phase 3** Every process waits for the estimate from the coordinator. If process receives the estimate from the coordinator, it updates its own estimate and sends back an *ack* to the coordinator. If the failure detector of the process suspects the coordinator, the process sends back a *nack* to the coordinator.
- **Phase 4** The coordinator waits for a majority of *acks* or one *nack* to arrive. If the majority is reached, the coordinator reliably broadcasts the decision to others. When they received decision, they decide and deliver the decision to listener. If the coordinator receives a *nack*, it starts over with the next round, and the coordinator is changed to be another process.

Example run of the Chandra-Toueg consensus algorithm is shown in Figure 3.1. The pseudocode of Chandra-Toueg consensus algorithm is shown in appendix B.1

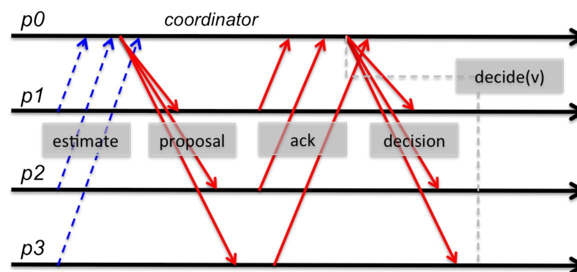

Figure 3.1: Example run of the Chandra-Toueg consensus algorithm

3.2.2 Mostéfaoui and Raynal consensus algorithm

Mostéfaoui and Raynal consensus algorithm [11] [12] is based on rotating coordinator same as Chandra-Toueg consensus algorithm. First, each process has its own estimate. Only the coordinator has estimateFromCoordinator same as its own estimate, estimateFromCoordinator in other process is *null*. The coordinator sends proposal to other process. If the coordinator is not suspected, the process that receives proposal from the coordinator replaces estimateFromCoordinator with the coordinator's estimate and then sends estimate

to others. When the number of received estimate from other process is reached, it sends the decision to others and decide. If the coordinator is suspected, the process will not update a new estimate from the coordinator. Each process will be reset and started next round.

Example run of the Mostéfaoui and Raynal consensus algorithm is shown in Figure 3.2. The pseudocode of Mostéfaoui and Raynal consensus algorithm is shown in appendix B.2

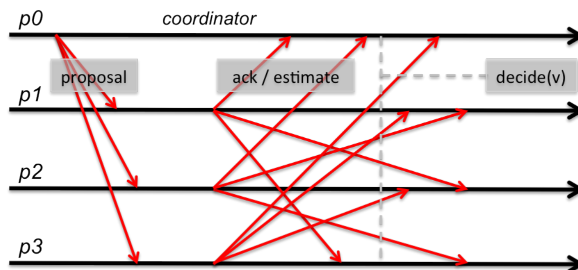

Figure 3.2: Example run of the Mostéfaoui and Raynal consensus algorithm

3.2.3 Paxos consensus algorithm

We implemented the most basic of the Paxos family on Neko and Yaya. Paxos consensus [8] algorithm is based on coordinator rotating. The protocol proceeds over several rounds, each round has two phases.

- **Read phase** The coordinator sends a *read* message with its own round number to all processes. Each process responds with an *ack* if it never accepted a *read* or a *write* with a round number higher that the round number in *read* message. If the process has already accepted a *read* or a *write* with a round number higher than the *read* message, it responds with a *nack* message. The coordinator waits for the majority of *ack* or one of *nack*. If the number of *ack* is reached, it moves on to the write phase. If it has received one *nack*, it increments its round number and starts over with the read phase.

- **Write phase** The coordinator sends a *write* message with its estimate and round number to all processes. Similarly to the *read* phase, the other processes check if they accepted a previous message with a higher round number. If not, they replace their own estimate with coordinator's estimate, save the round number and respond with *ack* message. If yes, they respond with a *nack* message. The coordinator waits for a majority of *ack* in which case it can broadcast the *decision* message. If the coordinator receives *nack*, it increments its round number and starts over with the read phase.

Example run of the Paxos consensus algorithm is shown in Figure 3.3. The pseudocode of Paxos consensus algorithm is shown in appendix B.3

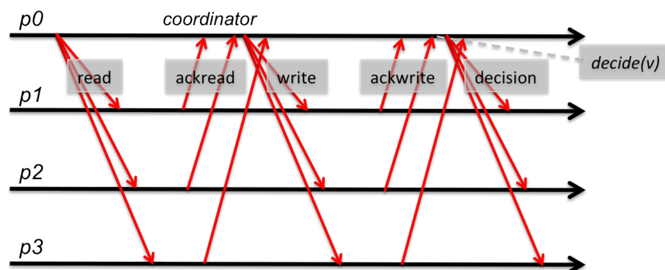

Figure 3.3: Example run of the Paxos consensus algorithm

3.3 Atomic broadcast

In distributed systems, atomic broadcast is a broadcast messaging protocol that ensures that messages are received reliably and in the same order by all participants. There are several algorithms in atomic broadcast.

3.3.1 Simple fixed sequencer algorithm

In a fixed sequencer algorithm [10], one process is elected as the sequencer and is responsible for ordering messages. One specific process takes the role of a sequencer and builds the total order. To broadcast a message, sender sends message to the sequencer. Upon receiving message, the sequencer assigns it a sequence number and relays message with its sequence number to destinations. Destinations receive message according to sequence numbers. The pseudocode of Simple fixed sequencer algorithm is shown in appendix C.1

3.3.2 Simple moving sequencer algorithm

Moving sequencer algorithms [10] are based on the same principle as fixed sequencer algorithms, but allow the role of sequencer to be transferred between several processes. To broadcast a message, sender sends a message to the sequencers. Sequencers circulate a token message that carries a sequence number and a list of all messages to which a sequence number has already been attributed. Upon receipt of the token, that sequencer assigns a sequence number to all received, but yet unsequenced, messages. It sends the newly sequenced messages to the destinations, updates the token, and then passes the token to the next sequencer. The pseudocode of Simple moving sequencer algorithm is shown in appendix C.2

3.3.3 Simple privilege-based algorithm

Privilege-based algorithms [10] rely on the idea that senders can broadcast messages only when they are granted the privilege. The privilege to broadcast messages is granted to only one process at a time, but this privilege circulates from process to process, among the senders. Senders circulate a token message that carries a sequence number to be used when broadcasting the next message. When a process wants to broadcast a message, it must first wait until it receives the token. It then assigns a sequence number to each of its messages and sends them to all destinations. Destination processes deliver messages in increasing sequence numbers. The pseudocode of Simple privilege-based algorithm is shown in appendix C.3

3.3.4 Simple communication history algorithm

In communication history algorithms [10], as in privilege-based algorithms, the delivery order is determined by the senders. However, in contrast to privilege-based algorithms, processes can broadcast messages at any time, and total order is ensured by delaying the delivery of messages. The messages carry a (physical or logical) timestamp. The destinations observe the messages generated by the other processes and their timestamps. The pseudocode of Simple communication history algorithm is shown in appendix C.4

3.3.5 Using consensus to solve Atomic broadcast

Atomic broadcast [9] is fundamental problem in fault tolerant distributed computing. Informally, Atomic broadcast requires that all correct processes deliver the same messages in the same order. Formally, Atomic broadcast is a Reliable broadcast that satisfies:

- *Total order* If two correct processes p and q deliver two messages m and $\acute{m}$, then p delivers m before $\acute{m}$ if and only if q delivers m before $\acute{m}$.

The total order and agreement properties of Atomic broadcast ensure that all correct processes deliver the same *sequence* of messages.

In appendix C.5, we show how to transform any Consensus algorithm into an Atomic broadcast algorithm in terms of pseudocode. Atomic broadcast algorithm uses repeated executions of Consensus. Intuitively, the kth execution of Consensus is used to decide on the kth batch of messages to be atomically delivered.

3.4 Criteria

In this section, we give a definition of *effective code* and *boilerplate code* as our criteria to evaluate our protocol composition framework.

3.4.1 Effective code

Effective code is the term used to describe sections of code that are the main parts of algorithm. In our evaluation, effective code is logically equivalent compared with pseudocode of algorithm and we assume that the pseudocode of algorithm is the minimum section of effective code in that algorithm.

3.4.2 Boilerplate code

Boilerplate code is the term used to describe sections of code that have to be included in many places with little or no alteration. In our evaluation, boilerplate code is logically not equivalent compared with pseudocode. We illustrate *boilerplate code* as the following:

- Special contents declaration and content casting
- Interface than ordinary protocol that is provided by framework
- Condition checking in each state of protocol
- Getter and setter method, parameters, aspects.
- Redundant part in algorithm logic
- Synchronization handling

We show the examples of the boilerplate code in 3.1

```
1  \\ Special content declaration
2  public abstract static class ContentWithRound extends ExecutionID
3          implements Serializable {
4          public ContentWithRound(int number, int round){
5              super(number);
6              this.round = round;
7          }
8          public final int round;
9      }
10 \\ ConsensusInterface and FailureDetector are interfaces
11 public class ConsensusCTExecution extends ProtocolImpl implements
       ReceiverInterface, ConsensusInterface, FailureDetectorListener {
12     public void propose(Object obj){...}
13 ...}
14 \\ Content casting
15 processEstimate((EstimateContent) content);
16 \\ Synchronization handling
17 public void propose(Object o){
18   ...
19   synchronized(lock){
20     // create new message
21   }
22 }
```

Listing 3.1: Example of boilerplate code

Chapter 4

Problem statement

In this chapter, we address the problems in Neko composition framework in terms of expressiveness. We use Chandra-Toueg consensus algorithm as a benchmark. We compare some parts of Chandra-Toueg consensus protocol on Neko with pseudocode and state the complicate parts.

4.1 Problems

We show each case of the complicate parts in Chandra-Toueg consensus algorithm as following:

```
1  public class ConsensusCTExecution extends ProtocolImpl
2    implements ReceiverInterface, ConsensusInterface, FailureDetectorListener
       {
3    private void processMessageWithRound(NekoMessage m){
4      Object content = m.getContent();
5      switch(m.getType()){
6        case MessageTypeConst.CONS_ESTIMATE:
7          processEstimate((EstimateContent) content);
8          break;
9        case MessageTypeConst.CONS_PROPOSE:
10         processPropose((ProposeContent) content);
11         break;
12       case MessageTypeConst.CONS_ACK:
13         processAck((AckContent) content);
14         break;
15       case MessageTypeConst.CONS_ABORT:
16         processAbort((AbortContent) content);
17         break;
18       default:
19         throw new UnexpectedMessageException(m);
20     }
```

Listing 4.1: Each round messages are processed according to the message type

In Listing 4.1, we have to declare message type constant. In line 7,10,13 and 16, we have to cast content type according to message content type. In this case, we need to concern message type with content type. This should be correctly matching, if not it will occur an exception. This complicated can easily make an error in an implementation if we have a lot number of protocol. We show pseudocode that is the same part of Listing 4.1 as following:

```
1 Phase 1: all processes p send estimate_p to the current coordinator
2 Phase 2: coordinator gathers (n+1)/2 estimates and proposes new estimate
3 Phase 3: all processes wait for new estimate proposed by current
    coordinator
4 Phase 4: the current coordinator waits for replies: (n+1)/2 acks or 1 nack.
     If they indicate
```

```
1  private void processEstimate(EstimateContent content) {
2    if(process.getID() != coord) // Coordinator checking
3      throw new RuntimeException("Unexpected message");
4    if(phase != 2) { //ignore // Phase checking
5      return;
6    }
7    if(content.getLastUpdated() > lastUpdated) {
8      lastUpdated = content.getLastUpdated();
9      estimate = content.getEstimate();
10   }
11   numEstimate++;
12   if(numEstimate >= limit) {
13     phase = 4;
14     NekoMessage m = new NekoMessage(others, getId(),
15                     new ProposeContent(k, round, estimate),
16                     MessageTypeConst.CONS_PROPOSE);
17     sender.send(m);
18     processAck(null);
19   }
20 }
```

Listing 4.2: Estimate phase processing code

In Listing 4.2, it shows estimate phase in Chandra-Toueg. In this phase, only coordinator can executed this method, so we have to check if this process is coordinator (in line 2, 3). Moreover, even though the process is coordinator but if this process has already been changed to other phases, the content will be ignored. We check current phase in line 4-6.

```
1  private void incRound() {
2    ...
3    if(process.getID() != coord) {
4      if(isCoordSuspected()) { // Suspicion handling
5        int[] coordDest = { coord };
6        NekoMessage m = new NekoMessage(coordDest, getId(),
7                    new AckContent(k, round, false),
8                    MessageTypeConst.CONS_ACK);
9        sender.send(m);
10     }
11   ...
```

Listing 4.3: Failure detector handler code (A)

```
1  private void processSuspicion(int content) {
2    if(coord != process.getID() && content == coord) {
3      if(!isUsingAbortMessage || !isWaitingForSuspicionOrAbort){
4        // Suspicion handling
5        int[] coordDest = { coord };
6        NekoMessage m = new NekoMessage(coordDest, getId(),
7                    new AckContent(k, round, false),
8                    MessageTypeConst.CONS_ACK);
9        sender.send(m);
10     }
11     incRound();
12   }
13 }
```

Listing 4.4: Failure detector handler code (B)

In Listing 4.3 and Listing 4.4, we show failure detector part in Chandra-Toueg consensus algorithm. In Listing 4.3, this part is inside incRound method which is run in each round. The process detects failure by its own failure detector. In Listing 4.4, this part is handled for other processes's messages. However, the code in listing 4.3 at line 5-9 and in listing 4.4 at line 4-8 is *redundant part* in source code.

In Listing 4.5, we show consensus interface that provide *propose* method for starting execution. Consensus execution core (ConsensusCTExecution) is encapsulated in ConsensusCT that provide *propose* method to other protocols or applications. It is hindrance for composing protocols, we face difficulties to recognize which protocols should have interaction, which aspects need to be configured. Moreover, because of concurrency, we have to consider some part of code that must be a critical section. In Listing 4.5 line 13, we have a group to make an agreement. There is possibility that one thread is executing this part but another changes a group variable. This situation leads execution to inconsistency state. Programmers need careful consideration in this complexity.

```
1  public class ConsensusCT extends MultipleExecutions
2    implements ConsensusInterface { // Special Interface and special
       composition instance that wrap protocol logic inside.
3    ...
4    public ReceiverInterface createReceiver(int execID) {
5      ConsensusCTExecution c = createExecution(execID); // Protocol logic is
         encapsulated here
6      c.setSender(sender);
7      c.setDecisionListener(myDecisionListener);
8              failureDetectorMulticast.addListener(c);
9              return c;
10         }
11   public void propose(Object o){
12     NekoMessage m;
13     synchronized(lock){ // Synchronization part
14       k++;
15       m = NekoMessage(new int[] { process.getID() },
16                   getId(), new ConsensusValueWithGroup(k, o,
17                          (int []) group.clone()),
18                    EventTypeConst.CONS_START);
19     }
20     deliver(m);
21   }
22 }
```

Listing 4.5: Chandra-Toueg consensus interface

4.2 Analysis

According to Chandra-Toueg [9] consensus algorithm that is implemented on Neko, we emphasized that there are some hindrances for implementing algorithm on Neko. We conclude them and classify into a number of problems as following:

1. The code is too long and complex, we have to declare special messages, message type and concern about casting content.

2. Interaction among protocols, some situations we need special interface that wrap the core executing inside.

3. There are many boilerplate code, for examples, method for declaring data content structure, synchronization and method for iteration of algorithm flow.

4. It is capable of making an error. Concerning of concurrency model [6] is very important in implementing on Neko.

Chapter 5

Actor-based model

5.1 Introduction

Fundamental concept is adopted from the philosophy that *everything is an actor*. This is similar to the idea of object-oriented programming languages but differs in that object-oriented software is executed sequentially, while the Actor model is inherently concurrent. In response to a message that it receives, an actor can make local decisions according to message type, create more actors, send more messages, and determine how to respond to the next message received. Actors provide a programming model that gives stronger guarantees about concurrent code when compared with the traditional *shared-memory-based* abstraction. Actors are essentially well encapsulated active objects, which can only communicate by sending one another immutable messages asynchronously.

5.2 Model

The actor model [13] [14] emphasizes the communication occurring during computation. For examples, parameter passing between subroutines of a program, message transferred between computers in a geographically distributed network, and process synchronization in a multiprocessing computer. All these communications may be considered as *message passing*

The actor model [13] [14] is one of message passing models. In message passing models, the models differ in their conception of message passing. For some models, the mechanism of message passing is similar to a telephone network, so that message transmission is essentially instantaneous, but sometimes the line or the receiver is busy and messages cannot be sent. However, for the actor model, message passing mechanism is similar to mail service, so that messages may always be sent but there are some variable delays to their destinations.

5.2.1 Actor model in Scala language

In concurrency programming, Java language includes support for concurrency and although this support is sufficient, it turns out to be quite difficult to implement or get larger and more complex. Scala augments Java's native concurrency by adding *actors* [15] [16]. Actors provide a concurrency model that is easier to work with, therefore, help developer to avoid many of the difficulties of using Java's native.
An actor is a thread-like entity that has a mailbox for receiving messages. We use subclass `scala.actors.Actor` and implement the `act` method. The idea of actor in Scala language is *shared nothing*. We can communicate without using shared memory and locks by sending each other *messages*. Scala actor model provides the ! method for sending message asynchronously, like this: `ActorInstance ! message`. `ActorInstance` is actor instance and `message` is a sending message that can be any type.
An actor will receive messages by partial function *receive* or *react*. In case of *react*, a return type is *nothing*, when we call `start` on an actor, the `start` method will in some way arrange things such that some thread will eventually call `act` on that actor. If that `act` invokes `react`, the `react` method will look in mailbox for a message the passed partial function can handle. If it finds a message that can be handled, `react` will schedule the handling of that message and throw an exception. If it doesn't find one, it will place the actor in "cold storage"and will be waked up when it gets more messages in its mailbox, and throw an exception. In either case, `react` will complete abruptly with this exception, and so will `act`. The thread that invoked `act` will catch exception, forget about the actor, and move on to other duties. Because the *react* method does not need to return, the implementation does not need to preserve the call stack of the current thread. Instead, the library can reuse the current thread for the next actor that wakes up.

5.2.2 Why is actor model in Scala?

We choose actor model in Scala language cause of following reasons:
1. Scala language can be run on Java Virtual Machine (JVM), we can implement with other library that can be run on JVM.
2. Scala language supports functional and object-oriented programming.
3. Type inference in Scala makes syntax shorter and more expressive.
4. Message-based concurrency with pattern matching in Scala makes programming in concurrency more safe than shared-memory concurrency with locks.

5.3 Comparing with Java thread

In this section, we consider about actor model in Scala comparing with Java concurrency thread. We consider ease of implementing in both Java thread and Scala actor by using producer consumer application as a benchmark.

5.3.1 Design and Implementation view

Java concurrency model In Java concurrency model, objects and resources can be accessed by many separate threads; each thread has its own path of execution but can potentially access any object in the program. Programmer needs to ensure reading and writing access to objects by using *synchronized* between thread. Thread synchronization ensures that objects are executed by only one thread at a time and that threads are prevented from accessing partially updated objects during executing by another thread. We show producer-consumer application that is implemented by Java thread using shared memory and synchronized in the following:

```
1  public class Producer extends Thread {
2    private List<Integer> queue;
3    private int delay;
4    private int next = 0;
5    public Producer(List<Integer> queue,int delay){
6        this.queue = queue;
7        this.delay = delay;
8    }
9    public void run() {
10     while(true){
11       synchronized (queue) {
12         queue.add(next);
13         queue.notifyAll();
14       }
15       next++;
16       try {
17         sleep(delay);
18       } catch (InterruptedException e) {}
19     }
20   }
21 }
```

Listing 5.1: Producer.java

In Listing 5.1, all producers have *queue* as a shared-memory. When one producer needs to produce element, it observes the queue if it is not locked by another threads, the producer can move into critical section. In the critical section, producer adds new element into the queue and notifies all threads that are waiting for the queue.

In Listing 5.2, all consumers have *queue* as a shared-memory. When one consumer needs to consume element, it observes the queue if it is available, the consumer can move into critical section. In the critical section, consumer will check the queue if it is not empty, consumer will get a element from queue. If it is empty, consumer will wait for next element. Consumers will be notified by producer that finished adding element into the queue.

```
public class Consumer extends Thread {
  private List<Integer> queue;
  private int delay;
  public Consumer(List<Integer> queue, int delay){
    this.queue = queue;
    this.delay = delay;
  }
  public void run() {
    while(true){
      synchronized (queue) {
        if(queue.size()>0){
          Integer number = queue.remove(queue.size() - 1);
          System.out.println(number);
        }else{
          try {
            queue.wait();
          } catch (InterruptedException e) {}
        }
      }
      try {
        sleep(delay);
      } catch (InterruptedException e) {}
    }
  }
}
```

Listing 5.2: Consumer.java

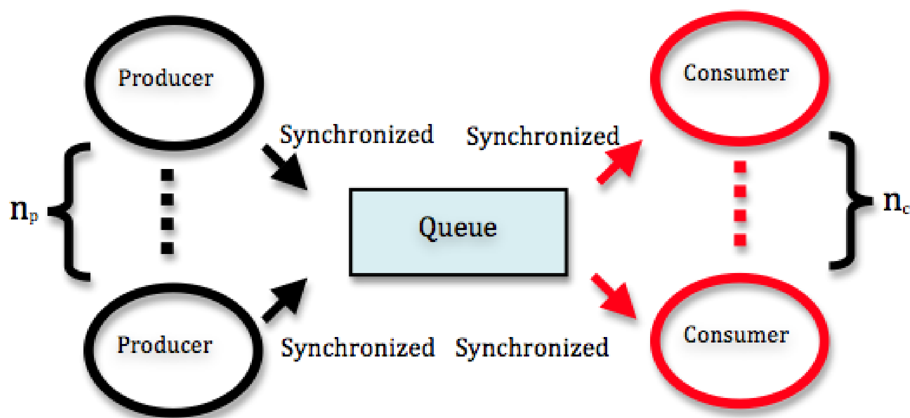

Figure 5.1: Producer Consumer Application in Java Concurrency model

In Figure 5.1, when either producer or consumer wants to do something with queue, it needs to lock the queue first. In implementation, we need to consider when one thread needs to lock a shared-memory and when the thread will release locking.

Scala actor model In Scala actor model, everything is an actor, when one actor needs to communicate with each others, it uses *message passing*. In each actor view, it receives messages from *mailbox*, filters the type of message and reacts the message. The relation between each actor is almost independent from each other. We can concentrate only in the mechanism of each other. We show producer-consumer application that is implemented by Scala actor model in the following:

```
1  class Producer(buffer: UnboundedBuffer, delay: Long, producerId: Int)
       extends Actor{
2    def act(){
3      var i = 0
4      loop{
5        buffer ! Put(i)
6        i += 1
7        Thread.sleep(delay)
8      }
9    }
10 }
```

Listing 5.3: Producer.scala

In Listing 5.3, producers send a message to buffer and wait for a while to send a new message again. Producers no need to consider synchronization, they only consider their own mechanism.

```
1  class Consumer(buffer: UnboundedBuffer, delay: Long, consumerId: Int)
       extends Actor{
2    private var i = 0
3    def act() {
4            loop{
5        Thread.sleep(delay)
6        buffer ! Get(this)
7        react {
8          case respond => {
9              println(consumerId+":"+respond);
10         }
11       }
12     }
13   }
14 }
```

Listing 5.4: Consumer.scala

In Listing 5.4, consumer send consuming request to buffer and wait for coming back element. Consumers consider only their own mechanism that sends the request and waits a respond back. They no need to synchronize the buffer before sending a message.

```scala
class UnboundedBuffer extends Actor{
  private var queue: List[Int] = Nil
  private val MAX_SIZE = 100
  def act {
    loop{
      react {
        case Get(consumer) if(queue.size != 0) => {
          consumer ! queue.head
          queue = queue.tail
        }
        case Put(value) if(queue.size < MAX_SIZE)=> {
          queue = queue :+ value
        }
      }
    }
  }
}
```

Listing 5.5: UnboundedBuffer.scala

In Listing 5.5, buffer is doing as an actor when message comes to the buffer it will be saved into mailbox. Buffer will take a message from mailbox and then check a message type. In the buffer, there are 2 types of message *Get(from)* and *Put(value)*. The buffer will receive *Get(from)* message from consumer if there are element in queue, it will send an element to consumer. If there are no element in queue, the consumer request will not be taken from mailbox until there is new element from producer. For *Put(value)* message, buffer will receive it from producers and put a new element into the queue. We need only consider in interesting messages, no need to handle *Get(from)* when there is no element in queue and *Put(value)* when the queue is full.

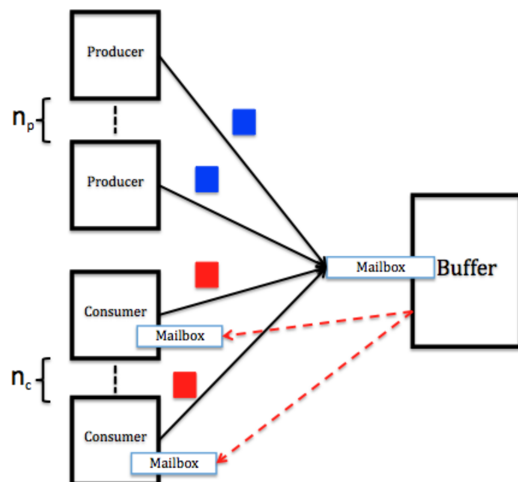

Figure 5.2: Producer Consumer Application in Scala actor model

In Figure 5.2, it shows that producers and consumers send messages to *mailbox* regardless of the buffer's status. The buffer only consider in each message type cases and then make a reaction according to message type.

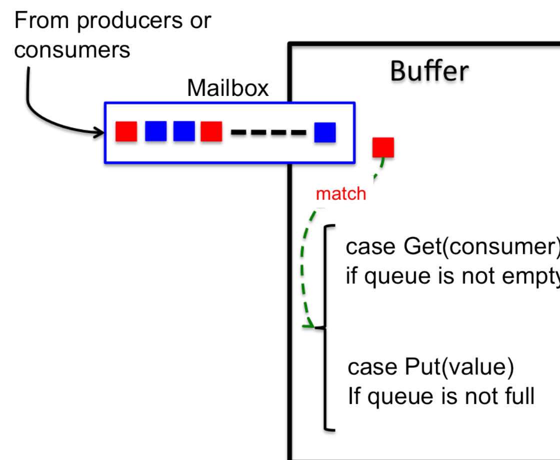

Figure 5.3: Mailbox in buffer actor

In Figure 5.3, we show that the buffer takes a one message at a time and find a corresponding case. After finished execution one message, the buffer will take a new message from mailbox again until mailbox is empty.

Chapter 6

Yaya protocol composition framework: Design

In this chapter, we present the overview of Yaya protocol composition framework. We first present the architecture of the framework. We then describe the idea of composition framework that is based on actor model. Finally, we give an example that is implemented on Yaya.

6.1 Introduction

We implement Yaya protocol composition framework based on the idea of Neko composition framework. In Neko composition framework, we have above protocol and below protocol. Above protocol sees below protocol as *sender interface*, send a message to below by using *send* method. On the received side, below protocol sees above protocol as *receiver interface*, a message is delivered to above protocol by using *deliver* method. In yaya protocol composition framework, using actor model, we have only one channel to receive the messages from other protocols. In more details, we explain in architecture section.

6.2 Architecture

As shown in Figure 6.1, the architecture of Yaya consists of two main parts: *application* and *network*

At the level application, a collection of processes (numbered from process 0 to process $n-1$) communicate using a message passing interface (YPublisher interface) that wraps actor's message passing method (! method(send *msg* to the actor asynchronously)) inside. A sender process pushes its message onto the network that is implemented YReceiver, and the network then pushes that message onto the receiving process with ! method.

Application layers Yaya protocols are constructed as a direction graph. One protocol only concern about its own interaction, where is the next direction of the message.

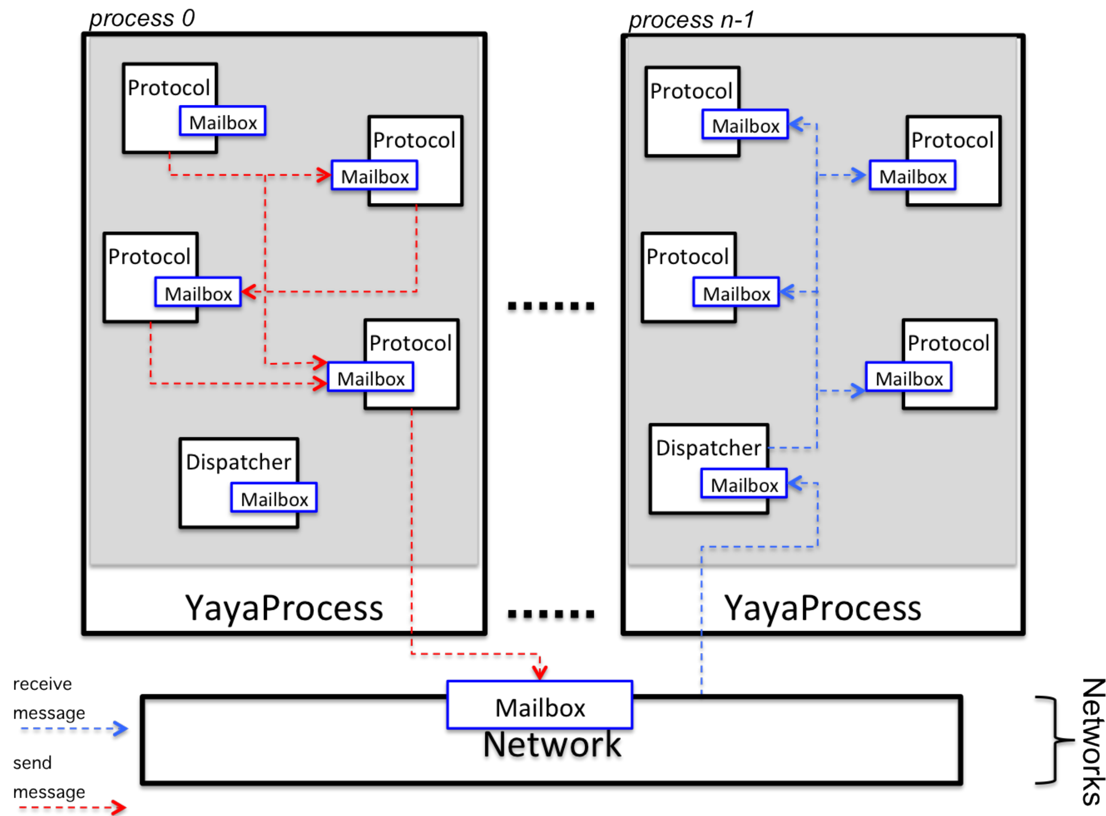

Figure 6.1: Architecture of Yaya

Messages to be sent are passed to next protocol defined by composer until they are sent onto network or there are no passing anymore depending on the protocol execution. Messages are passed by using send method (there are two parameters, message and next protocol id). Delivered messages from network are passed up to *dispatcher* and it then passes message to protocol depending on message type. Messages may be delivered in reverse direction of sender process.

YayaProtocol In each protocol in Yaya framework, we need to implement YProtocol interface for expressing meaning and carrying some variables(e.g. protocol id, number of process and the process that holds this protocol instance). We implement YReceiver interface in order to listen messages that come from other protocols or other processes. We implement YPublisher interface for sending messages to other protocols or other processes. Inside the protocol, protocol has to declare its own content type that interests in. If there is no interested messages have been sent or delivered to this protocol, it will ignore that messages. Therefore, protocol programmers divide message handling into a number of cases based on content type, protocol state, some conditions.

YayaProcess Each process of distributed application has an associated object of type

YayaProcess class, hold all protocols that are running on. It holds some process information, useful services(e.g. logging message). Moreover, process make an interaction that is defined by protocol composer before starting execution.

YayaMessage All communication in Yaya transmit instances of YayaMessage. A message is created by createMessage method inside instance of YProtocol. Messages are composed by two main parts: message header and content stack. Message header holds some information, such as sender process id and message destinations. Content stack contains contents from previous protocols illustrated in Figure 6.2. Instance object in content stack is instance of YayaContent.

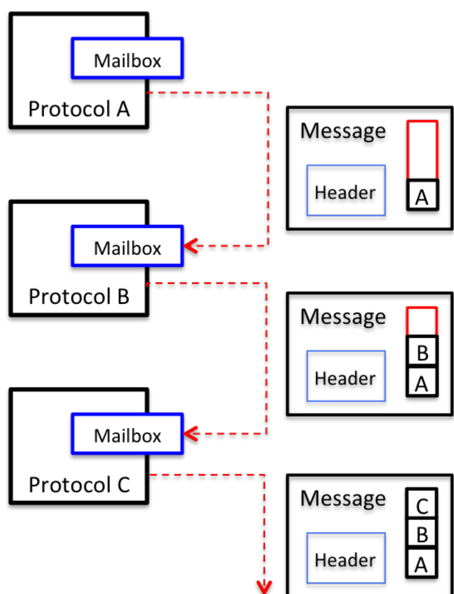

Figure 6.2: Yaya message sending mechanism

YayaContent In each YayaMessage instance, there is a stack of YayaContent which contains a protocol id and content part. Protocol id is from the protocol that put the YayaContent into stack. Content part consists of any Scala object. We can declare any content type that extends from Any in Scala language.

6.3 Sample protocol: Pingpong protocol

In this section, we illustrate how to implement protocol on Yaya by using an example. The given example protocol is *Pingpong protocol*, that is defined as following:

There are *two* types of message in the protocol. When process received message, process will execute that message according to message type. Message processing cases are explained as following:

1. **PING Message:** When process received PING message, the process will send PONG message back to sender process.

2. **PONG Message:** When process received PONG message back until a number of PONG message be same as number of received PING message processes, the process will send new PING message to other processes again.

From this description, we have *two* types of message and execution of each case. We implement this algorithm that is shown in Listing 6.1

```
1  case class PING;
2  case class PONG;
3  class PingPong(id: String, process: YProcess) extends YActiveProtocol(id,
     process){
4    ...
5    private val sender = "sender"; // sender instance will be defined by
       composer
6    private var receivedPong : Int = 0;
7    def contentMatch(msg: YMessage): PartialFunction[Any, Unit] = {
8      case PING() => { //received PING message
9        val dest = Array[Int]{msg.processID};
10       val newMsg = createMessage(dest).withContent(new PONG());
11       send(newMsg, sender);
12     }
13     case PONG() => { //received PONG message
14       if(receivedPong == nProcesses - 1){
15         receivedPong = 0;
16         val newMsg = createMessage(allButMe).withContent(new PING());
17         send(newMsg, sender);
18       }else{
19           receivedPong = receivedPong + 1;
20       }
21     }
22   }
23 }
```

Listing 6.1: Pingpong code example implemented on Yaya

In Listing 6.1, we need to define interested content type for the protocol (in line 1-2). In constructor of the protocol, we need to declare two parameters, protocol id and process for composer. Pingpong protocol needs one interaction (sender in line 5), so composer specifies interactions which one is sender in composition level. The most important part in protocol is contentMatch method. This method is used for processing a message that received from other protocols or other processes. In Listing 6.1, message handler are separated into two cases (line 9-22), PING content and PONG content.

1. In PING case, protocol creates a new message back which contains PONG content and send back by using send method (in line 10-11).

2. In PONG case, protocol checks a number of received PONG content. If a number of received PONG content is same as a number of received PING content process,

protocol will create message back that contains PING content and it then send to other processes by using send method.

Chapter 7

Yaya protocol composition framework: User manual

In this chapter, we explain about user manual of Yaya protocol composition framework. First, we describe about overview architecture and component models in Yaya framework. We then give an example of implementation for getting started. Finally, we explain in more general implementation.

7.1 Overview

Yaya is a software framework written in Scala language for constructing complex distributed algorithm from simple protocol. In Yaya framework, we have a model as shown in Figure 7.1.

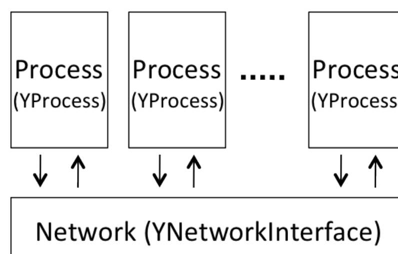

Figure 7.1: Simulated model in Yaya protocol composition framework

In Figure 7.1, we have a number of processes that are simulated in systems. Every processes are instance of YProcess class. Each process connect to network and exchange informations with other processes. Network in Yaya framework needs to implement YNetworkInterface interface. Every process has *id* to identify the process and we use this *id* to send information via the network.

In each process, we have *a number of protocols* that are running on systems and a one *dispatcher*. Every protocols have their own algorithm logic, event handling mechanism

and protocol states. We show the components module in process as Figure 7.2

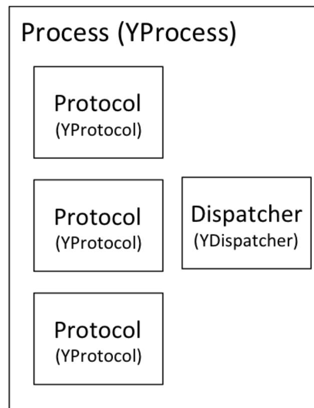

Figure 7.2: Component module in a process

In Figure 7.2, we have several protocols and one dispatcher in a process. Every protocol extends from YProtocol interface. Each protocol has its own *id* for communicating with related protocols or same protocol in other processes. Protocol uses *id* when send message to others in order to deliver message in receiving process. In receiving process, we use a dispatcher to dispatch a message to target protocol. Dispatcher uses protocol *id* to dispatch each message.

In communication within process, each protocol send a message to other protocol. Yaya framework provide two interfaces for communicating, YSender interface for sending message and YReceiver interface for receiving a message from other protocols or other processes. We show a communication within a process in Figure 7.3

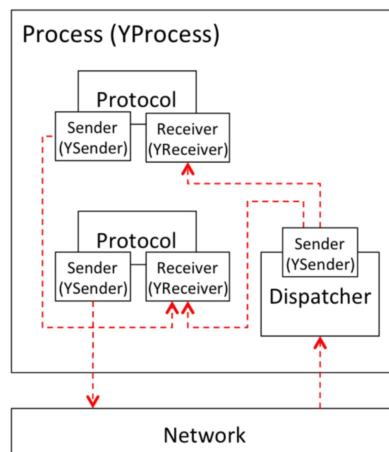

Figure 7.3: Communication within a process

In Figure 7.3, we have several protocol and one dispatcher that have interconnection between them. Each protocol implements *two* interfaces, YSender interface and YReceiver

interface. Protocol implements YSender interface for sending a message to other protocols. Protocol implements YReceiver interface for receiving a message from dispatcher or other protocols. Protocol has only one way for receiving a message, so programmers have to define content into cases. We show communication within a process using message in Figure 7.4 and structure of message in Yaya framework in Figure 7.5.

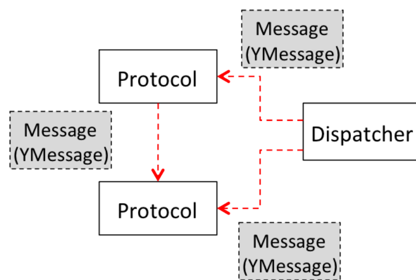

Figure 7.4: Communication using message within a process

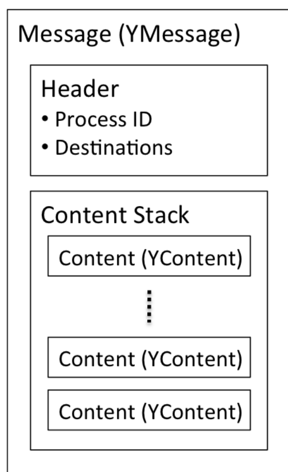

Figure 7.5: Structure of message in Yaya framework

In Figure 7.5, we show structure of message that is used to communicate between protocols and processes. Message in Yaya framework is instance of YMessage class. We separate message into two parts, header and content stack. Message's header contains process id that identifies the sender process and destinations that specify receiver processes. Content stack contains contents from protocols. In sender process, contents from protocols will be pushed into stack orderly. In receiver process, contents will be pulled from stack reversely.

In Yaya protocol composition framework, we have component models that are defined as following

1. **Process** In Yaya framework, process is instance of YProcess class. Each process holds process id, protocols, interactions between protocol. Process connects to networks and exchanges information with other processes by process id. We show YProcess class in Figure 7.6

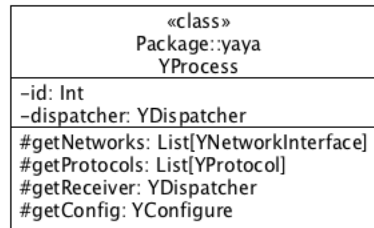

Figure 7.6: YProcess class diagram

2. **Protocol** Our protocols are implemented by using YProtocol interface. Most of our protocols extends YActiveProtocol that already implemented YSender interface and YReceiver interface. Every protocol has ID for interacting with the same protocol in other processes. Protocol receives message from other protocols or other processes by using YReceiver interface. When protocol receives message, it checks the message content. If content is matching with its interesting condition, protocol will execute that message but if not, the message will be kept in mailbox until it matches conditions. We show class diagram of protocol in Figure 7.7

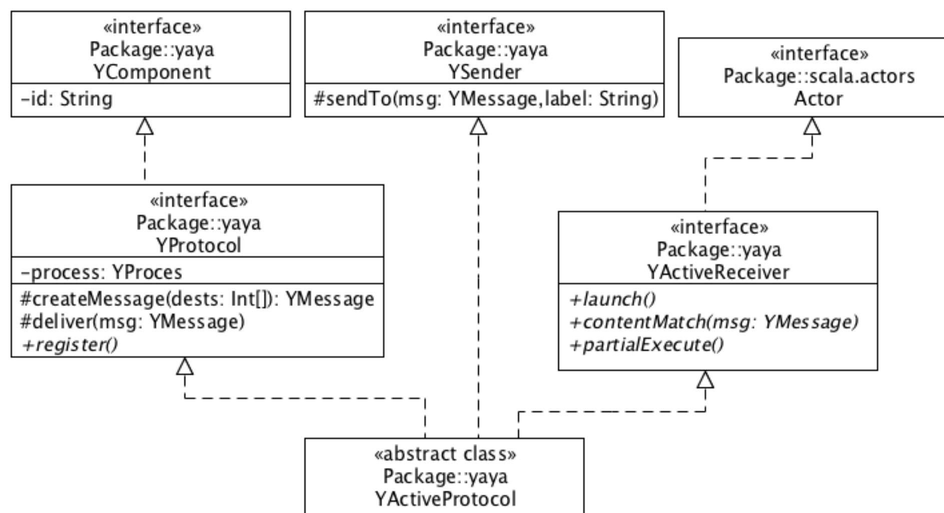

Figure 7.7: Class diagram of protocol in Yaya framework

3. **Message** Every message in Yaya framework is instance of YMessage class. We cannot create YMessage instance directly but we have to create a new message by using *createMessage* method in YProtocol class. Every message contains destinations, sender process id and stack of contents. We show class diagram of message in Figure 7.8

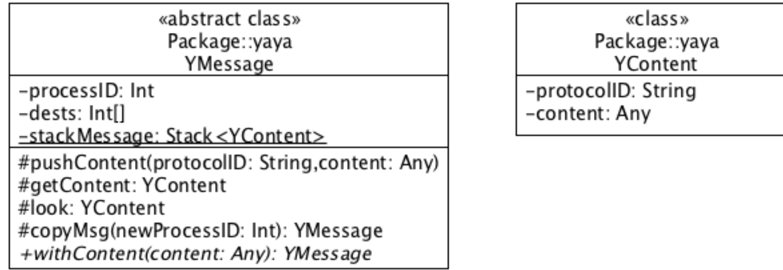

Figure 7.8: Class diagram of message in Yaya framework

4. **Protocol interaction** In the same process, one protocol connects with other protocols by using *sendTo* method in YSender interface. We have two parameters, *msg* for messsage and *label* for next protocol. We define the list of other protocols that are interested. When composing protocol, we will match that list of other protocols with protocol instances.

5. **Initialization** The initialization of our system consists of the following actios, in the order of occurrence:

   (a) The system is created and start to build processes according to a number of processes in configuration file.
   (b) The system creates network instances and makes connection between processes and networks.
   (c) The system initiates each process.
   (d) Each process creates protocols and registers the interactions between protocol.
   (e) Protocols in every processes will be launched

6. **Communication between processes** Protocols communicate with a protocol in another process by calling sendTo(msg, label) to send the message to next protocol in the sender process until the message is sent to network. The sender has to specify the following fields of YMessage:

   - the address that identifies the destination process or processes. The address is thus an array of integer process IDs.
   - the ID of destination protocol

- a content Object in the protocol.

In the receiving process, a dispatcher object forwards YMessage to the destination protocol. It does a lookup from protocol ID to protocol. Protocol checks content object if it is matching, protocol calls the corresponding execution.

7.2 Implementation view

In Yaya framework, we separate implementation into two views, *Composers* and *Protocol programmers*. Composers configure some aspects in the systems (number of processes, logging file) and make a composition between protocols. Composers make an implementation by creating configuration file. Each protocol in process is implemented by *Protocol programmers*. Protocol programmers consider their own algorithm logic, handling message from others. The difference between *two* roles is shown in Figure 7.9.

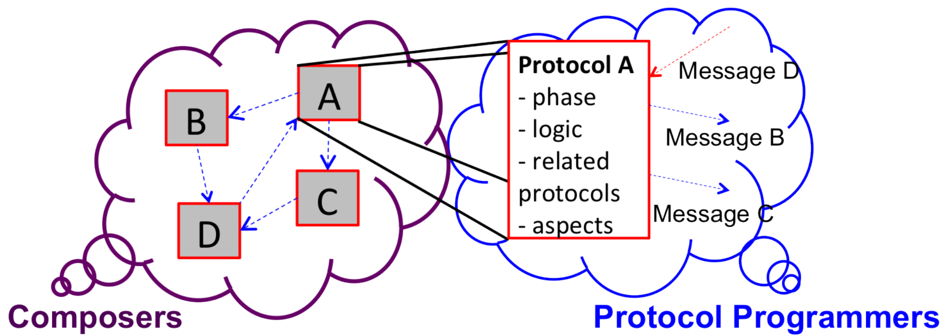

Figure 7.9: Difference between composers and protocol programmers

Composers In Yaya protocol composition framework, composers customize protocols inside the process by writing in configuration file. Composers need to define how protocols are composed, the communications between each protocol, one protocol connects to which protocol. In Yaya framework, the composition model is cooperative model that every protocols are arranged in graph, one protocol has connections with any protocols.

Protocol programmers In Yaya protocol composition framework, programmers consider only their own algorithm logic, messages between protocols and protocols that have interactions with their protocol. The interaction model in Yaya framework is on event-based. When protocol receives message, protocol will be reactive and consider the content in message matching with their own interesting content.

7.3 Getting started

In this section, we explain how to first create simulation on Yaya framework. As a first example, we use the standard Hello world program to demonstrate the use of Yaya framework.

7.3.1 Hello world!

In Hello world program, process 0 will send Hello message to all processes. When process receives Hello message, the process will print *HelloWorld* message.

7.3.2 Creating a configuration file

In configuration file, we need to define a number of processes in the system, protocols and networks. We show the configuration file for Hello world program in Listing 7.1, Listing 7.2 and Listing 7.3 and illustrate this configuration file in Figure 7.10.

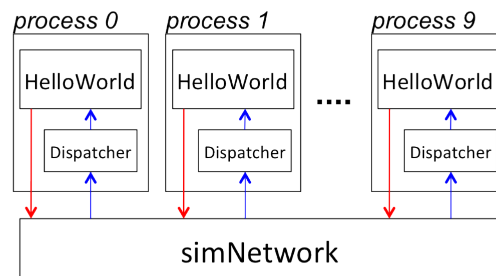

Figure 7.10: Systems model from *helloWorld* configuration file

For a syntax of configuration, we show the syntax of configuration file in Figure 7.11.

```
1 process.num = 10;
2 networks = [simNetwork];
3 protocols = [helloWorld];
```

Listing 7.1: *helloWorld.config* configuration file for Hello world program (1)

In Listing 7.1, the configuration file indicates that there are 10 processes connected with simNetwork network and each process has helloWorld protocol. We give definition of helloWorld protocol in Listing 7.2 and definition of simNetwork network in Listing 7.3.

```
1 helloWorld = {
2   classpath = yaya.sample.HelloWorld;
3   interaction = {
4     sender = simNetwork;
5   }
6 }
```

Listing 7.2: *helloWorld.config* configuration file for Hello world program (2)

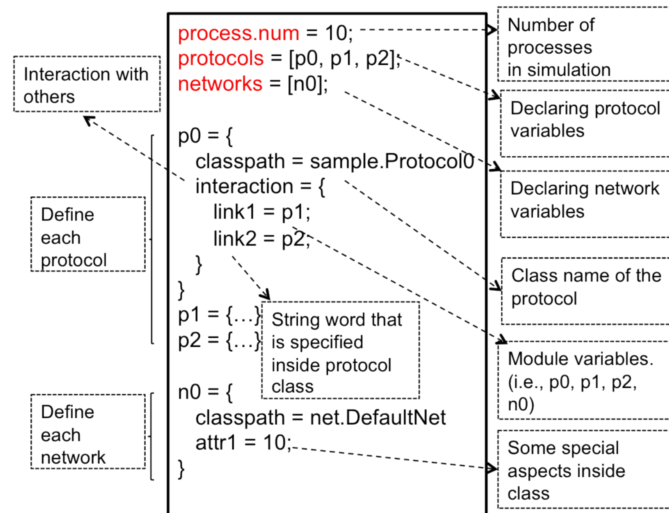

Figure 7.11: Configuration file syntax

In Listing 7.2, we define helloWorld protocol that is instance of yaya.sample.HelloWorld class. In yaya.sample.HelloWorld class, we have to declare sender interaction (sender attribute is defined in HelloWorld class). For this example, HelloWorld protocol send a message to other processes via simNetwork network.

```
1 simNetwork = {
2   classpath = yaya.network.BasicNetwork;
3   BasicNetwork.lambda = 1;
4 }
```

Listing 7.3: *helloWorld.config* configuration file for Hello world program (3)

In Listing 7.3, we define simNetwork network that is instance of yaya.network.BasicNetwork class. In yaya.network.BasicNetwork class, we need to define BasicNetwork.lambda attribute.

7.3.3 Implementing protocols

Then, we have to create yaya.sample.HelloWorld class, declare content type and implement some method in class. Most of protocols on Yaya framework extend YActiveProtocol class. We show protocol class syntax in Figure 7.12 and this example implementation in Listing 7.4.

```
1 case class Hello()
2 class HelloWorld(id: String, process: YProcess)
3     extends YActiveProtocol(id, process) {
4   def contentMatch(message: YMessage): PartialFunction[Any, Unit] = {
5     case Hello() => Console.println(process.id+" HelloWorld")
6   }
7 }
```

Listing 7.4: Example of protocol class declaration

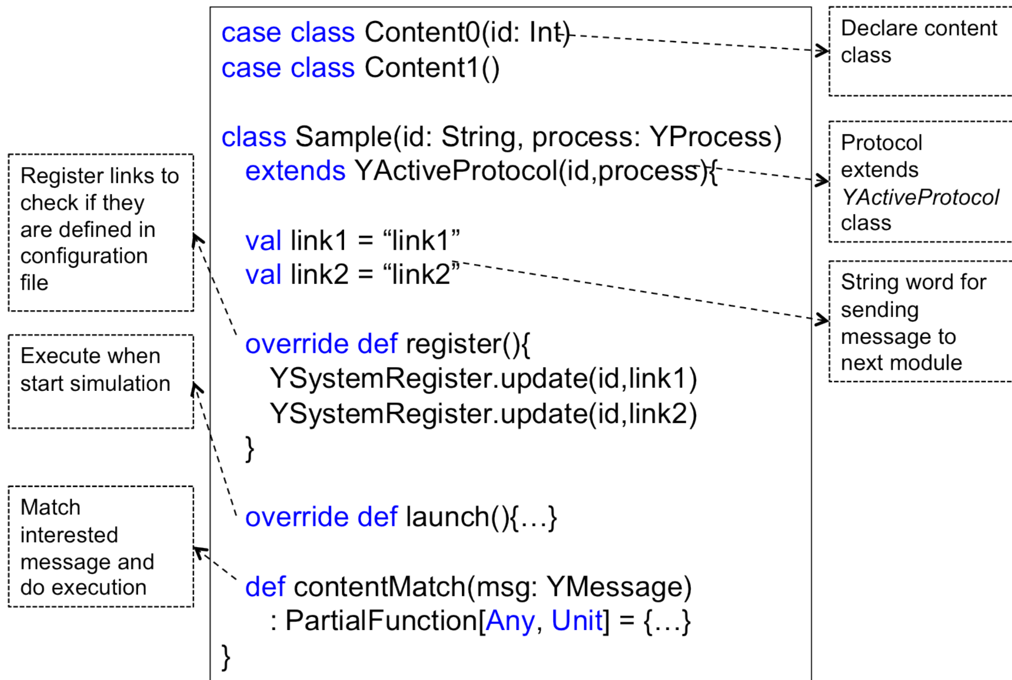

Figure 7.12: Protocol class syntax

In Listing 7.4, HelloWorld class extends YActiveProtocol class and specifies constructor that has id and process as parameters. Moreover, we need to implement contentMatch method for handling interested message. For content matching in contentMatch method, we have to define our interested messages for this protocol. For this example, when HelloWorld protocol receives Hello content, protocol will print “HelloWorld” message.

For process 0, it starts to send a “HelloWorld” message to all processes. When the simulation is started, we can start to send the message by overriding launch method. We show overriding launch method in Listing 7.5

```
1  case class Hello()
2  class HelloWorld(id: String, process: YProcess)
3      extends YActiveProtocol(id, process) {
4    val next = "sender" // next protocol is declared in configuration file
5    override def launch(){
6      if(process.id == 0){
7        val msg = createMessage(all).withContent(new Hello())
8        sendTo(msg, next);
9      }
10   }
11 }
```

Listing 7.5: Example of overriding launch method

In Listing 7.5, we override launch method for start sending message when simulation begin. Only process 0 sends message with Hello content to all process by using createMessage method. For createMessage method, we input an array of destinations as parameter and the first content in the message. We show how to create a message in Figure 7.13.

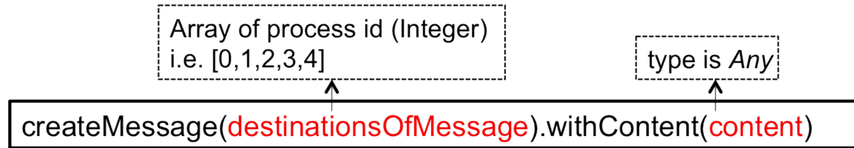

Figure 7.13: Using of createMessage method

Protocol can send created message to next protocol by using sendTo method. We show how to send a message in Figure 7.14.

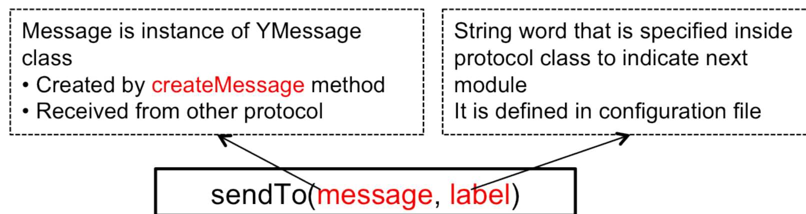

Figure 7.14: Using of sendTo method

7.3.4 Starting execution

In this example, we have *helloWorld.config* as configuration file for Hello World program. A simulation with configuration file *helloWorld.config* is launched the following way:

scala yaya.Main helloWorld.config

The result of this program is shown in Figure 7.15.

```
9 HelloWorld
8 HelloWorld
7 HelloWorld
5 HelloWorld
6 HelloWorld
4 HelloWorld
3 HelloWorld
2 HelloWorld
1 HelloWorld
0 HelloWorld
```

Figure 7.15: Hello World program result

7.4 More general implementation

In this section, we explain how to implement more complicated composition. We illustrate explanation by separating into *two* roles (*Composers* and *Protocol programmers*).

In more complicated composition, we have several protocols that are composed inside the systems. However, the interaction between protocols and the composing protocols have some restrictions. For these reasons, we explain the implementation by using steps in Figure 7.16 for *Composers* and Figure 7.18 for *Protocol programmers*.

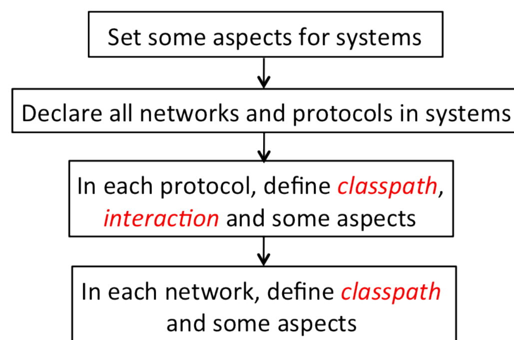

Figure 7.16: Composing steps for *composers*

In Figure 7.16, we have four steps for creating a configuration file. In defining each

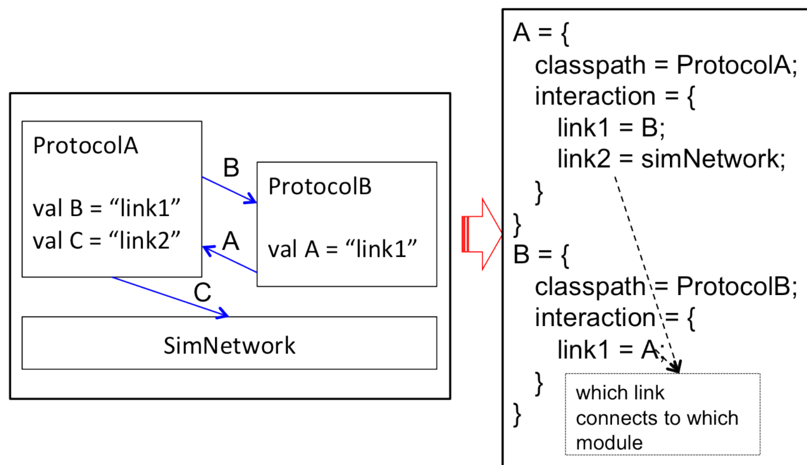

Figure 7.17: Restriction for *composers*

protocol (*classpath*, *interaction* and some aspects), for each protocol, composers have to know which links should connect to which protocols. We show this restriction in Figure 7.17.

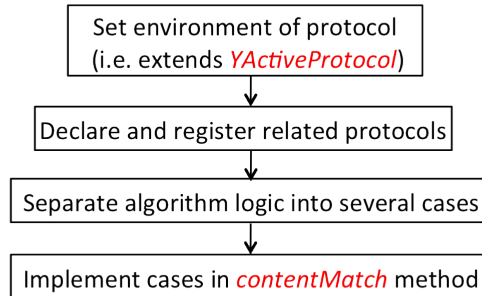

Figure 7.18: Protocol implementing steps for *protocol programmers*

For protocol programmers, we show the implementation steps in Figure 7.18. We have four steps for implementing protocol. In implementing cases in *contentMatch* method, some protocols are used by other protocols, so it has to provide the case for that protocols. On the other hand, when the protocol wants to notify some messages to other protocols, other protocols need to provide the case for notification. We show this restriction in Figure 7.19

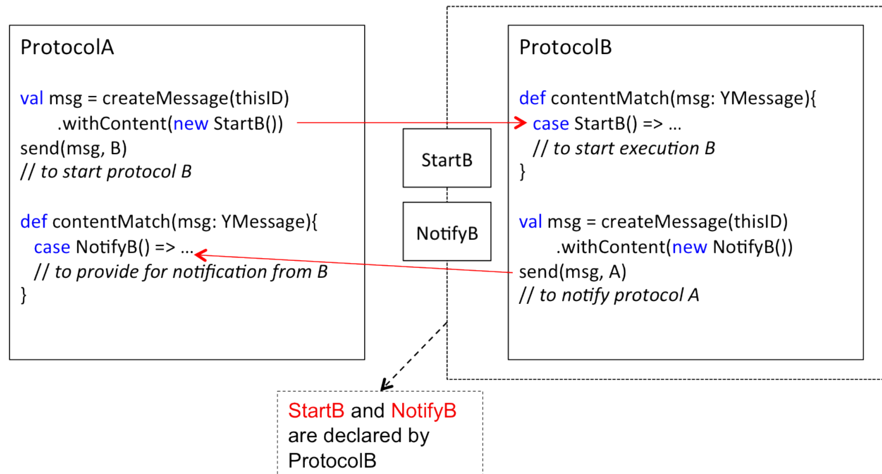

Figure 7.19: Restriction for *protocol programmers*

In Figure 7.19, we show the restriction between protocols. For ProtocolB class, protocol programmers have to declare two content types (StartB class and NotifyB) for other protocols. Other protocols that use ProtocolB have to know these content types.

Chapter 8

Yaya protocol composition framework: Usage

In this chapter, we illustrate the application on Yaya using an example. The example is explained in detail in order to describe how easily programmer can develop distributed applications with Yaya, how to compose protocols to be complex applications. First, we give an application using atomic broadcast as a sample application. We then show how to configure some aspects in application (e.g. number of processes, protocols). Last, we show how to start simulation execution.

8.1 Sample application

In a sample application, we show application using atomic broadcast. The architecture of atomic broadcast application is shown in Figure 8.1

Application In Figure 8.1, each process has an application component in top of application layer. Application in each process sends *string messages* to application in other processes by using atomic broadcast. All application in all processes will receive messages in same order.

Atomic Broadcast In Figure 8.1, each process has an atomic broadcast component that has next interactions (application, consensus and reliable broadcast). This atomic broadcast uses consensus and reliable broadcast in algorithm (composer specify which consensus algorithm and reliable broadcast algorithm will be used). When atomic broadcast received response, it will send the respond to listener (application).

Consensus In Figure 8.1, according to atomic broadcast specification, we have to define consensus algorithm. Consensus component has next interactions (reliable broadcast and usual network).

Reliable broadcast In Figure 8.1, reliable broadcast is used by consensus and atomic

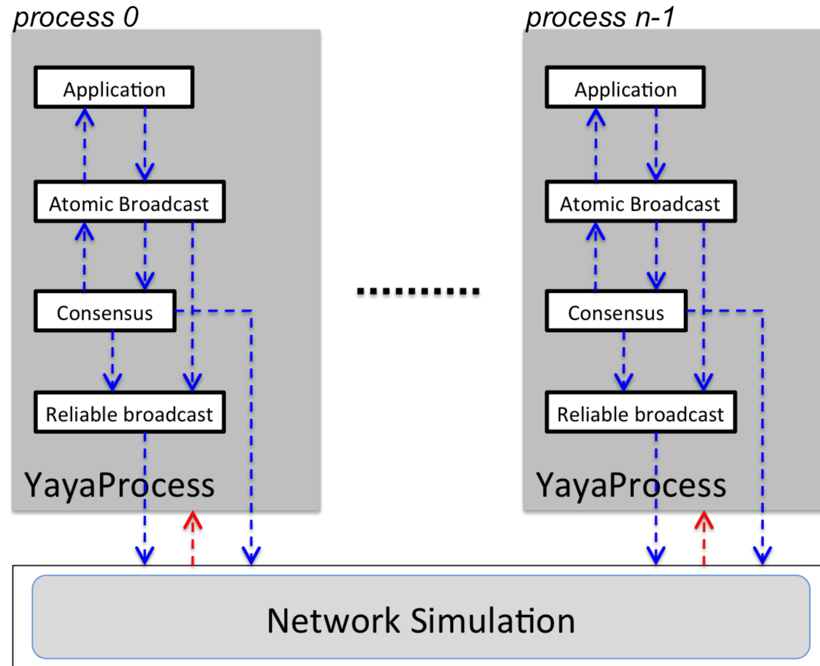

Figure 8.1: Architecture of a sample Yaya application: Atomic broadcast application

broadcast. Consensus algorithm and atomic broadcast no need to concern about inside of reliable broadcast algorithm. Reliable broadcast ensures only that every messages will be delivered to destinations.

For algorithm programmer view, every protocols only consider in its own algorithm, interested content type and next interactions. For composer view, composers consider only interactions of each protocol, no need to know detail of algorithm. We explain how to configure aspects in distributed application and compose protocols, in section 8.2.

8.2 Startup and configuration

In this section, we explain what support Yaya provide for initialization and configuring a distributed application by using a sample application in Figure 8.1.

Configuration In configuration file, we have to configure many aspects (number of processes) and protocols in a single file. The name of this file is passed to Yaya on startup. Each process has the information in the configuration file when the application starts. Basic configuration file contains:

1. **number of processes** We define number of processes that will be simulated in systems by defining `process.num` parameter.

2. **declaration of protocols** We declare each of protocol component name by defining protocols parameter.
3. **declaration of networks** We declare each of network component name by defining networks parameter.
4. **each protocol detail** We have to specify each protocol detail that is declared in protocols parameter. In basic detail, we have to specify *classpath* and *interaction*.
5. **each network detail** We have to specify each network detail that is declared in networks parameter. In basic detail, we have to specify *classpath*.
6. **logging description** We have to specify *logHandler* for taking log messages from a logger and exports them.

In Listing 8.1, we show a configuration file in a sample application in section 8.1

```
1  process.num = 5;
2  networks = [simNetwork];
3  protocols = [test ,atomic ,consensus ,rbcast];
4  test = {
5    classpath = yaya.atomic_broadcast.test.TestAtomicBroadcast;
6    interaction = {
7      sender = simNetwork;
8      atomic = atomic;
9  } }
10 atomic = {
11   classpath = yaya.atomic_broadcast.AtomicBroadcast;
12   interaction = {
13     rbcast = rbcast;
14     listener = test;
15     consensus = consensus;
16 } }
17 consensus = {
18   classpath = yaya.consensus.ConsensusCT;
19   interaction = {
20     listener = atomic;
21     diffusion = rbcast;
22     sender = simNetwork;
23 } }
24 rbcast = {
25   classpath = yaya.rbcast.RBroadcast;
26   interaction = {
27     sender = simNetwork;
28 } }
29 simNetwork = {
30   classpath = yaya.network.BasicNetwork;
31   BasicNetwork.lambda = 1;
32 }
33 log = { logHandler = src/yaya/atomic_broadcast/test/logAtomic; all :: all;
     }
```

Listing 8.1: Configure file example

In configuration file example, we have 5 processes simulated in systems. This simulation is run on simNetwork network. Each process 4 protocols are named as test, atomic, consensus and rbcast. Every protocol and network component is defined in Listing 8.1 (line 4-32).

Initialization The names of the classes implementing protocols and networks are given by classpath inside component. Each process gets configuration file and it then creates instances of protocol and network following a configuration file. When simulation is initialized, the system creates processes according to number of processes. Each process then constructs the hierarchy of protocols and interactions between protocols by accessing configuration file.

Chapter 9

Comparative Analysis: Expressiveness

In this chapter, we analyze algorithms that are implemented on Yaya comparing with implemented on Neko. We compare algorithm in two ways implementation by using line of code as an evaluation. We show how implementation of algorithms on Yaya can reduce complexity and boilerplate code in Neko. Our algorithm benchmarks that are used to evaluated our framework is shown in following:

- **Reliable broadcast** We show the implement of reliable broadcast that is provided for the other algorithm to use this protocol such as consensus algorithm.
- **Consensus algorithm** We show the fundamental problem in distributed system. In consensus problem, there are several algorithms as following:
  - **Chandra-Toueg consensus algorithm**
  - **Mostéfaoui and Raynal consensus algorithm**
  - **Paxos consensus algorithm**
- **Atomic broadcast** We show more composition view of our protocol by using atomic broadcast that is solved by consensus. We also show the simple algorithm in atomic broadcast.

9.1 Reliable broadcast

9.1.1 Evaluation

On Neko
In reliable broadcast implemented on Neko, we separate message handling into two methods (*send* and *deliver* method).

```
public void send(NekoMessage m) {
  GUID id = new GUID(process);
  int source = process.getID();
  int[] dests = m.getDestinations();
  Arrays.sort(dests);
  boolean sendingToSelf = 0 <= Arrays.binarySearch(dests, source);
  if (sendingToSelf) {
    synchronized (already) {
      IntHolder numReceiveLeft = new IntHolder(dests.length);
      already.put(id, numReceiveLeft);
    }
    receiver.deliver(m);
  }
  ContentRB content = new ContentRB(id, source, m.getProtocolId(),
                           m.getContent(), m.getType());
  NekoMessage newM = new NekoMessage(source, dests, getId(),
                              content, MessageTypeConst.RBCAST_MESSAGE);
  sender.send(newM);
}
```

Listing 9.1: *Send* method in reliable broadcast on Neko

```
proc send(m)
  newId = nextId(id,process) // get next id of message
  already = already + {m,newId,0} // add number of receives
  send (m,newId) to destinations
```

Listing 9.2: Pseudocode of *send* method in Reliable broadcast

In Listing 9.1, we show the implementation of *send* method in reliable broadcast on Neko. In *send* method, reliable broadcast provides method for listening message from other protocols. When there is message from others, protocol creates id for that message and sends to next sender. This part is same as the part of pseudocode in Listing 9.2.

In Listing 9.3, we show the implementation of *deliver* method in reliable broadcast on Neko. In *deliver* method, reliable broadcast provides it for receiving delivered message. It checks the message if it has already received that message, it will do nothing. If it is the first time that received the message, it will resend that message again to other destinations. This part is same as the part of pseudocode in Listing 9.4. The full pseudocode of reliable broadcast on Neko is shown in appendix A.

```
1  public void deliver(NekoMessage m) {
2    if (m.getType() == MessageTypeConst.RBCAST_MESSAGE) {
3      ContentRB content = (ContentRB) m.getContent();
4      GUID id = content.getId();
5      boolean reSend;
6      synchronized (already) {
7        IntHolder numReceiveLeft = (IntHolder) already.get(id);
8        if (numReceiveLeft == null) {
9          reSend = true;
10         numReceiveLeft = new IntHolder(m.getDestinations().length);
11         already.put(id, numReceiveLeft); // mark(1)
12       } else {
13         reSend = false;
14         numReceiveLeft.value--; // mark(2)
15         if (numReceiveLeft.value <= 0) {
16           already.remove(id);
17     } } }
18       if (reSend) {
19       NekoMessage newM = new NekoMessage(content.getSource(), m.
           getDestinations(), content.getProtocolId(), content.getContent(),
           content.getType()); // mark(3)
20       receiver.deliver(newM); //mark(3)
21       NekoMessage newM2 = new NekoMessage(m.getDestinations(), getId(),
           content, m.getType()); // mark(4)
22       sender.send(newM2); // mark(4)
23 } } }
```

Listing 9.3: deliver method in reliable broadcast on Neko

```
1 when receive(m,id)
2   if !(id isin already)
3     already = already + {m, newId, 1} // mark(1)
4     deliver(m) // mark(3)
5     send(m, id) to destinations // mark(4)
6   else
7     already = already update (m, id, receivedNumber + 1) // mark(2)
```

Listing 9.4: Pseudocode of *deliver* method in Reliable broadcast

On Yaya

We have to implement *contentMatch* method to handle message that come to this protocol. Depend on content type, message is processed and sent to next sender.

```
1  def contentMatch(msg: YMessage): PartialFunction[Any, Unit] = {
2    case bContent : BroadcastMsg if(!already.contains(bContent.id)) => {
3      already += (bContent.id -> msg.dests.length);
4      val broadcastMsg = msg.copyMsg(processID);
5      broadcastMsg.pushContent(id, bContent);
6      sendTo(broadcastMsg,SENDER);
7      dispatch(msg);
8    }
9    case bContent : BroadcastMsg => {
10     val numReceiveLeft = already.apply(bContent.id);
11     if(numReceiveLeft <= 0) already -= bContent.id;
12     else already += (bContent.id -> (numReceiveLeft - 1));
13   }
14   case content : Any => {
15     msg.pushContent(msg.getContent.protocolId, content);
16     val guid = GuID.newID(processID);
17     val dests = msg.dests;
18     val sendingToSelf = dests.contains(process.getId);
19     val broadcastMsg = msg.cloneMsg;
20     broadcastMsg.pushContent(id,new BroadcastMsg(guid));
21     sendTo(broadcastMsg,SENDER);
22     if(sendingToSelf){
23       val numberReceiveLeft = dests.length;
24       already += (guid -> numberReceiveLeft);
25       dispatch(msg);
26 }}}}
```

Listing 9.5: contentMatch method in reliable broadcast on Yaya

In Listing 9.5, there are three cases of content type. In *first* case, the content is BroadcastMsg(inform others that the message is used for reliable broadcast) and the protocol have not already received the message yet. The protocol creates a copy one and sends to other destinations. Then, it delivers the message to receiver according to content type in message. In *second* case, the content is BroadcastMsg but the protocol already received the message. The protocol reduces left number of received message. In *third* case, the content is Any. The protocol creates new id for the message, puts BroadcastMsg content in the message and then sends to other destinations. The part of first case and second case is the *send* method in pseudocode shown in Listing 9.2. The part of third case is the *deliver* method in pseudocode shown in Listing 9.4.

From both of reliable broadcast on Neko and Yaya, we show the complexity in terms of line of code in Table 9.1

Reliable broadcast	Boilerplate code	Effective code	Total
On Neko	28 (42.42%)	38 (57.58%)	66
Compared with pseudocode (14 lines)		2.71 times	4.71 times
On Yaya	16 (40%)	24 (60%)	40
Compared with pseudocode (14 lines)		1.71 times	2.86 times
code reduced	42.86%	36.84%	39.39%

Table 9.1: Comparative between implementation of Reliable broadcast on Neko and Yaya in terms of line of code

9.2 Consensus

Consensus is agreement problem in distributed systems that encapsulates the task of group agreement. There are many algorithms for solving consensus problem. In this section, we illustrates implementation on Neko and Yaya by using Chandra-Toueg consensus algorithm, Mostéfaoui and Raynal consensus algorithm and Paxos consensus algorithm. We then analyze the differences between Neko and Yaya in terms of line of code and implementation.

9.2.1 Chandra-Toueg consensus algorithm

On Neko

In chapter 4, we show examples of complexity in implementation of Chandra-Toueg consensus algorithm.

On Yaya

In implementation of Chandra-Toueg algorithm on Yaya, we separate necessary contents into several classes(EstimateContent, ProposeContent, AckContent, DecideContent and AbortContent). Each phase in Chandra-Toueg algorithm interests in different contents. Content handler is separated into several cases as following:

```
1  case StartWithGroup(EXN, value, group) if isInState(Phase.READY) &&
       isCoordinator(group) && !(RND > 0) => {
2    setGroup(group);
3    phase = Phase.DECIDING;
4    this.estimate = new EstimateValue(value, lastUpdated);
5    val msg = createMessage(others).withContent(new ProposeContent(EXN, round
         , this.estimate));
6    sendTo(msg,SENDER);
7  }
8  case StartWithGroup(EXN, value, group) if isInState(Phase.READY) => {
9    setGroup(group);
10   this.estimate = new EstimateValue(value, lastUpdated);
11 }
```

Listing 9.6: Start executing Chandra-Toueg consensus algorithm on Yaya

In Listing 9.6, other protocols send StartWithGroup content to this protocol, we split into 2 cases (coordinator and not coordinator). In Neko, this part need to be encapsulated by ConsensusInterface(special interface that is not provided by framework). In Yaya, we can implement this inside without special interface.

```
1 case EstimateContent(EXN, RND, estimate) if isCoordinator && isInState(
      Phase.ESTIMATING) => {
2   if(estimate.lastUpdated > this.lastUpdated){
3     this.lastUpdated = estimate.lastUpdated;
4     this.estimate = estimate;
5   }
6   numbEstimate = numbEstimate + 1;
7   if(numbEstimate > limit){
8     phase = Phase.DECIDING;
9     val msg = createMessage(others).withContent(new ProposeContent(EXN,
          round, this.estimate));
10    sendTo(msg,SENDER);
11 }}
```

Listing 9.7: phase 2 of Chandra-Toueg algorithm on Yaya

In Listing 9.7, this case is provided for listening estimate from other processes. Only coordinator collects estimate until the number of estimate is reached. There is condition statement for filtering that makes code be shorter and more expressiveness.

```
1 case ProposeContent(EXN, RND, estimate) if !isCoordinator && isInState(
      Phase.READY) => {
2   val coordDest = Array[Int]{getCoordinator(RND)};
3   this.lastUpdated = RND;
4   if(!coordIsSuspect){
5     this.estimate = estimate;
6     val msg = createMessage(coordDest).withContent(new AckContent(EXN,RND,
          true));
7     sendTo(msg,SENDER);
8   }else{
9     val msg = createMessage(coordDest).withContent(new AckContent(EXN,RND,
          false));
10    sendTo(msg,SENDER);
11  }
12 }
13 case Suspect(processNumber: Int) => {
14   suspectList += (processNumber -> true);
15 }
```

Listing 9.8: suspect content handling on Yaya

In Listing 9.8, there are 2 cases, ProposeContent and Suspect. In ProposeContent case, the process receives this content and if the coordinator is not suspected by the process, the process will send *ack* back to the coordinator. If the coordinator is suspected, the process will send *nack* back to the coordinator. In Suspect case, the process receives this content from failure detector in same process or other processes. Only one case, we can handle suspect from inside failure detector and other processes's failure detector. We have no

redundant part in source code. This part of code is same as **Phase 3** of pseudocode in Listing 9.9

```
1 Phase 3: {all processes wait for new estimate proposed by current
     coordinator}
2   wait until[received(c, r, estimate) from c or c isin query failure
       detector]
3   if[received(c, r, estimate) from c] then
4     estimate_p = estimate_c // set process's estimate as coordinator's
         estimate
5     ts = r
6     send(p, r, ack) to coordinator
7   else // p suspect that coordinator crashed
8     send(p, r, nack) to coordinator
```

Listing 9.9: Pseudocode of Phase 3 in Chandra-Toueg consensus

9.2.2 Mostéfaoui and Raynal consensus algorithm

On Neko

In this example, implementation of Mostéfaoui and Raynal consensus algorithm uses *active* layer on Neko. In *active* layer, protocol has its own thread to run execution. We implement *run* method and use *receive* method to get a received message. We illustrate the example in more details as following:

```
1  public void run() {
2    NekoMessage m = null;
3    while (true) {
4      m = receive();
5      if (m.getType() == EventTypeConst.CONS_START) break;
6      if (m.getType() == EventTypeConst.CONS_SUSPICION) continue;
7      postpone(m);
8    }
9    ...
10 }
```

Listing 9.10: Example code of Mostéfaoui and Raynal consensus algorithm on Neko (1)

In Listing 9.10 line 5-6, we need to check message type before start execution. It is hindrance for understanding the procedure of algorithm. When it receives messages that are not interesting, they will be pushed into queue list. This situation can occur any place during algorithm execution. In this situation, implementation in Neko creates a lot of *redundant part* on protocol.

```
1 if (!failureDetector.isSuspected(c)) {
2   while (true) {
3     m = receiveInRound(r);
4     if (decided) return;
5     if (m.getType() == EventTypeConst.CONS_SUSPICION || failureDetector.
        isSuspected(c)) {
6       if (m.getType() != EventTypeConst.CONS_SUSPICION)
7         postpone(m);
8       break;
9     }
```

Listing 9.11: Example code of Mostéfaoui and Raynal consensus algorithm on Neko (2)

In Listing 9.11, this example shows the handling failure situation in protocol. The failure situation can occur anytime, so programmers have to consider when it needs to check failure situation. This programming style It is cause of *ambiguous* code in protocol.

On Yaya

We split message handling into several cases. Implementation of Mostéfaoui and Raynal consensus algorithm on Yaya reduces *redundant* part and makes code be more expressive. The example of Mostéfaoui and Raynal consensus algorithm on Yaya is shown in following:

```
1 def contentMatch(msg: YMessage): PartialFunction[Any, Unit] = {
2   val EXN = execNumb; //this execution number
3   val RND = round; //this round number
4   {
5     case StartWithGroup(EXN, value, group) => ...
6     case ConsensusEstimate(EXN,RND, value, source) if phase==Phase.READY && !
        coordIsSuspect && source == getCoordinator(round) => ...
7     case ConsensusEstimate(EXN,RND, value, source) if phase==Phase.READY && !
        coordIsSuspect => ...
8     case ConsensusEstimate(EXN,RND, value, source) if phase==Phase.READY &&
        coordIsSuspect => ...
9     case Decision(exn, value) if phase == Phase.READY && !coordIsSuspect =>
        ...
10    case Suspect(processNumber: Int) if phase==Phase.READY => ...
11    case content: ConsensusEstimate => ...
12 } }
```

Listing 9.12: Example code of Mostéfaoui and Raynal consensus algorithm on Yaya

In Listing 9.12, we split into several cases depending on conditions and content type. This implementation style reduces *redundant* part and *ambiguous* part. It eases to control each case and situation.

9.2.3 Paxos consensus algorithm

On Neko

Implementation of Paxos consensus algorithm on Neko is implemented by passive layer. We have to implement *deliver* method to handle received message. We illustrate the example in more details as following:

```
1  public void deliver(NekoMessage m) {
2    ConsensusValue localDecisionMessage = null;
3    synchronized (this) {
4      switch (m.getType()) {
5      case EventTypeConst.CONS_START:
6        ConsensusValueWithGroup content1 = (ConsensusValueWithGroup) m.
             getContent();
7        processStart(content1);
8        break;
9      case CONS_READ:
10     case CONS_WRITE:
11     case CONS_ACKREAD:
12     case CONS_NACKREAD:
13     case CONS_ACKWRITE:
14     case CONS_NACKWRITE:
15       ContentWithRound content2 = (ContentWithRound) m.getContent();
16       processMessageWithRound(m);
17       break;
18     case MessageTypeConst.CONS_DECISION:
19       DecisionContent content5 = (DecisionContent) m.getContent();
20       processDecision(content5);
21       break;
22     default:
```

Listing 9.13: Example code of Paxos consensus algorithm on Neko

In Listing 9.13, we declare message type and content. We need to create special message type and content to be consistency. Inside each case, we have to check conditions such as a number of *read*, *write* and *round*.

On Yaya

Similarly to implementation of Chandra-Toueg consensus algorithm and Mostéfaoui and Raynal consensus algorithm, we split message handling into several cases. This implementation style makes code be more expressive and easy to trace the flowing of algorithm. The example is shown in Listing 9.14

```
1  def contentMatch(msg: YMessage): PartialFunction[Any, Unit] = {
2    val EXN = execNumb;
3    val RND = roundNumb;
4    {
5      case StartWithGroup(EXN, value, group) if isInState(Status.IDLE) => ...
6      case PrepareContent(EXN, round) if (read > round) || (write > round) =>
         ...
7      case PrepareContent(EXN, round) if !((read > round) || (write > round))
         => ...
8      case AckReadContent(EXN, RND, estimate, lastWrite) if nbNack == 0 =>
         ...
9      case WriteContent(EXN, round, estimate) if (read > round || write >
         round) => ...
10     case WriteContent(EXN, round, estimate) if !(read > round || write >
         round) => ...
11     case AckWriteContent(EXN,RND) => ...
12     case DecisionContent(EXN,estimate) => ...
13     case NackContent(EXN, RND) =>
14   }
```

Listing 9.14: Example code of Paxos consensus algorithm on Yaya

In Listing 9.14, the algorithm is written in each case. One phase in algorithm can be separated into several cases according to conditions inside each phase. This point helps programmers to trace the algorithm flowing without looking inside the case. We show this situation comparing with pseudocode in Listing 9.15

```
1  when receive(m) from p_j {In process i}
2    if m = (r_j, read) then //{In Yaya: PrepareContent}
3      if write_i > r_j or read_i > r_j then
4        send(r_j, nackread) to p_j
5          // In Yaya: PrepareContent if (read > round) || (write > round)
6      else {no received a write/read with a higher round}
7        read_i = r_j
8        send(r_j, estimate_i, ackread) to p_j
9          // In Yaya: PrepareContent if !((read > round) || (write > round))
10   if m = (r_j, estimate_j, write) //{In Yaya: WriteContent}
11     if write_i > r_j or read_i > r_j then
12       send(r_j, nackwrite) to p_j
13         // In Yaya: WriteContent if (read > round) || (write > round)
14     else {no received a write/read with a higher round}
15       write_i = r_j
16       estimate_i = estimate_j
17       send(r_j, ackwrite) to p_j
18         // In Yaya: WriteContent if !((read > round) || (write > round))
```

Listing 9.15: Pseudocode of *Read* and *Write* content in Paxos consensus algorithm

9.2.4 Evaluation

From both of consensus on Neko and Yaya, we show the complexity in terms of line of code in Table 9.2

Chandra-Toueg algorithm	Boilerplate code	Effective code	Total
On Neko	160 (42.44%)	217 (57.56%)	377
compared with pseudocode (39 lines)		5.56 times	9.67 times
On Yaya	69 (40.59%)	101 (59.01%)	170
compared with pseudocode (39 lines)		2.59 times	4.36 times
code reduced	56.88%	53.46%	54.91%

Mostéfaoui and Raynal algorithm	Boilerplate code	Effective code	Total
On Neko	74 (41.34%)	105 (58.66%)	179
compared with pseudocode (30 lines)		3.5 times	5.97 times
On Yaya	63 (43.15%)	83 (56.85%)	146
compared with pseudocode (30 lines)		2.77 times	4.87 times
code reduced	14.86%	20.95%	18.44%

Paxos algorithm	Boilerplate code	Effective code	Total
On Neko	148 (43.40%)	193 (56.60%)	341
compared with pseudocode (45 lines)		4.29 times	7.58 times
On Yaya	67 (44.67%)	83 (55.33%)	150
compared with pseudocode (45 lines)		1.84 times	3.33 times
code reduced	54.73%	56.99%	56.01%

Table 9.2: Comparative between implementation of consensus algorithm on Neko and Yaya

9.3 Atomic broadcast

In distributed systems, atomic broadcast is a broadcast messaging protocol that ensures that messages are received reliably and in the same order by all participants. There are several algorithms in atomic broadcast. In this section, we show implementation of algorithms on Neko and Yaya and we then analyze the differences between both of implementations.

On Neko In the example of using consensus to solve Atomic broadcast, we use Chandra-Toueg consensus algorithm to solve. In more details of implementing on Neko is shown as following:

```
1  public class ChandraTouegClient
2  extends ProtocolImpl
3  implements SenderInterface {
4  public void send(NekoMessage m) {
5          GUID guid = new GUID(process);
6          NekoMessage m1 = new NekoMessage(...);
7          rbcast.send(m1);
8      }
9  }
10
11 public class ChandraToueg
12 extends ChandraTouegClient
13 implements DecisionListener, ReceiverInterface {
14   public void deliver(NekoMessage m) {
15   ...
16   }
17   public synchronized void notifyDecision(int k, Object decision) {
18   ...
19 }    }
```

Listing 9.16: Example code of Atomic broadcast using consensus on Neko (1)

In Listing 9.16, we show the specifications in implementation of Atomic broadcast on Neko. In this example, we use Chandra-Toueg algorithm to solve Atomic broadcast. We have to create special interface for broadcasting message and decision listener. It is hindrance for compatibility of modifying interactions between protocols.

```
1  public class ChandraToueg extends ChandraTouegClient implements
       DecisionListener, ReceiverInterface{
2    ...
3    private void aDeliver() { ... }
4    public void deliver(NekoMessage m) {
5      synchronized (this) {
6        switch (m.getType()) {
7          case MessageTypeConst.AB_START:
8            ...
9            updateQueueSize(); // mark1
10           aDeliver(); // mark1
11           break;
12   } } }
13   public synchronized void notifyDecision(int k, Object decision) {
14     ...
15     updateQueueSize(); // mark2
16     aDeliver(); // mark2
17   }
18 }
```

Listing 9.17: Example code of Atomic broadcast using consensus on Neko (2)

In Listing 9.17, deliver method and notifyDecision method (in *mark1* and *mark2*) have aDeliver function call in their own method for checking undelivered messages and delivering messages. If there are undelivered messages, it creates proposal and sends to Consensus. We have to implement the same mechanism in both of two methods.

On Yaya In the example of Atomic broadcast on Yaya, the Atomic protocol interests in its own contents, starting consensus content and decided consensus content. We show the sample implementation in following:

```
1  case class UndeliveredMessage(val msgId: GuID, val content: Any)
2  class AtomicBroadcast(id: String, process: YProcess) extends
       YActiveProtocol(id, process){
3    ...
4    def contentMatch(msg: YMessage): PartialFunction[Any,Unit] = {
5      case content: UndeliveredMessage => {...} // mark(1) // protocol
           collects undelivered message
6      case A_Broadcast(content) => {...} // mark(2) // from above protocol or
            application that use this atomic broadcast. Protocol send
           UndeliveredMessage to destinations.
7      case Decide(exn, value) => {...} // mark(3) // wait for Decide from
           consensus
8        if(k == nextExecNumber){ // mark(4) // until k is increased in Decide
             case, it can not start consensus.
9          val undelivered = getDifference(un_delivered, a_delivered);
10         if(undelivered != Nil){
11           nextExecNumber += 1;
12           val consensusMsg = createMessage(all).withContent(new
                 StartWithGroup(k, undelivered, all));
13           sendTo(consensusMsg, consensus); // end of mark(4)
14   } } }
15 }
```

Listing 9.18: Example code of Atomic broadcast using consensus on Yaya

In Listing 9.18, the protocol considers its own contents (R_Message, A_Broadcast), starting consensus content (StartWithGroup) and decided consensus content (Decide). A_Broadcast content is provided for listening broadcast request from other protocols or applications. The consideration of interactions between protocol is external contents (StartWithGroup and Decide) that are declared by other protocols. We illustrate each case of this algorithm comparing with pseudocode in Listing 9.19.

```
1  To execute A–broadcast(m);                          // begin // mark(2) in Yaya
2    R–broadcast(m)                                    // end   // mark(2) in Yaya
3  A–deliver(−) occurs as follows:
4    when R–deliver(m)                                 // begin // mark(1) in Yaya
5      R_delivered = R_delivered U {m}                 // end   // mark(1) in Yaya
6    when R_delivered − A_delivered != Nil             // begin // mark(4) in Yaya
7      k = k + 1
8      A_undelivered = R_delivered − A_delivered {get undelivered messages}
9      propose(k, A_undelivered)                       // end   // mark(4) in Yaya
10     wait until decide(k, msgSet_k)                  // begin // mark(3) in Yaya
11     A_deliver = msgSet_k − A_delivered
12     automically deliver all messages in A_deliver in some deterministic
           order
13     A_delivered = A_delivered U A_deliver           // end   // mark(3) in Yaya
```

Listing 9.19: Pseudocode of Atomic broadcast using Consensus

9.3.1 Evaluation

From both of Atomic broadcast on Neko and Yaya, we show the complexity in terms of line of code in Table 9.4

9.4 Analysis

The evaluation of comparative between implementation algorithms on Neko and Yaya in Table 9.1 (Reliable broadcast algorithm), Table 9.2 (Consensus algorithm) and Table 9.4 (Atomic broadcast algorithm), we show a number lines of code in both of boilerplate code and effective code. Implementation on Yaya reduce lines of code in implementation (contents declaration, synchronization handler, message handler, interactions between protocol). We also show the summary of comparative between implementation of algorithm on Neko and Yaya in Table 9.3.1.

In part of **boilerplate code**, there are many parts of code that we need to provide for other protocols; for examples, declaration of contents type, special message type, methods for called by others and synchronization. The many boilerplate codes detract programmer from the core of algorithm flowing. Implementation on Yaya, excepting Mostéfaoui and Raynal algorithm, we can reduce the part of boilerplate code around **54.67%** lines of code from the implementation on Neko. We observe that most of algorithms need to declare contents type, special message type and a lot of methods provided for other protocols. In case of protocol that has a few of contents type, special message type, we reduce around **15%** line of code from implementation on Neko.

In part of **effective code**, we need to implement each case of content types, each step in algorithm. Implementation of algorithms on Yaya makes code more compact, separates the message handler into each case depending on conditions. Implementation on Yaya

Algorithm	pseudocode	Neko	Yaya
Reliable broadcast	14	66 (4.71 times)	40 (2.86 times)
Chandra-Toueg Consensus	39	377 (9.67 times)	170 (4.36 times)
Mostéfaoui and Raynal Consensus	30	179 (5.97 times)	146 (4.87 times)
Paxos Consensus	45	341 (7.58 times)	150 (3.33 times)
Simple fixed sequencer	21	81 (3.86 times)	51 (2.43 times)
Simple moving sequencer	25	172 (6.88 times)	80 (3.2 times)
Simple privilege-based	23	105 (4.57 times)	53 (2.30 times)
Simple communication history	20	95 (4.75 times)	55 (2.75 times)
Atomic broadcast using consensus	18	133 (7.39 times)	64 (3.56 times)

Table 9.3: Summary of comparative between implementation of algorithm on Neko and Yaya

reduces the part of effective code around **35.74%** lines of code from implementation on Neko. Moreover, when we want to add some aspects into algorithm such as optimization parameter, on Neko, we need to implement in many parts of code. If we did not separate each case clearly, we will face difficulties to add the aspect. On Yaya, we can separate each case easily and make each case be more expressive.

Simple fixed sequencer	Boilerplate code	Effective code	Total
On Neko	43 (53.09%)	38 (46.91%)	81
compared with pseudocode (21 lines)		1.81 times	3.86 times
On Yaya	20 (39.22%)	31 (60.78%)	51
compared with pseudocode (21 lines)		1.48 times	2.43 times
code reduced	53.49%	18.42%	37.04%

Simple moving sequencer	Boilerplate code	Effective code	Total
On Neko	93 (54.07%)	79 (45.93%)	172
compared with pseudocode (25 lines)		3.16 times	6.88 times
On Yaya	39 (48.75%)	41 (51.25%)	80
compared with pseudocode (25 lines)		1.64 times	3.2 times
code reduced	58.06%	48.10%	53.49%

Simple privilege-based	Boilerplate code	Effective code	Total
On Neko	75 (71.43%)	30 (28.57%)	105
compared with pseudocode (23 lines)		1.30 times	4.57 times
On Yaya	30 (56.60%)	23 (43.40%)	53
compared with pseudocode (23 lines)		1 times	2.30 times
code reduced	60%	23.33%	49.52%

Simple communication history	Boilerplate code	Effective code	Total
On Neko	63 (66.32%)	32 (33.68%)	95
compared with pseudocode (20 lines)		1.6 times	4.75 times
On Yaya	29 (52.73%)	26 (47.27%)	55
compared with pseudocode (20 lines)		1.3 times	2.75 times
code reduced	53.97%	18.75%	42.11%

Atomic broadcast using consensus	Boilerplate code	Effective code	Total
On Neko	75 (56.39%)	58 (43.61%)	133
compared with pseudocode (18 lines)		3.22 times	7.39 times
On Yaya	32 (50%)	32 (50%)	64
compared with pseudocode (18 lines)		1.78 times	3.56 times
code reduced	57.33%	44.83%	51.88%

Table 9.4: Comparative between implementation of Atomic broadcast algorithm on Neko and Yaya

Chapter 10

Performance Analysis :Overhead

In this chapter, we evaluate the performance of Yaya in terms of execution time and availability. We compare the result of performance with Neko. The evaluation environment is defined as following:

Evaluation environment: We use *Intel(R) Core(TM)2 Duo CPU P8600 2.40GHz*, memory 2 GB. Operating system is *Ubuntu 11.04(natty)*. We start Java Virtual machine that has memory(heap space) 1024 M and thread stack size is 256K.

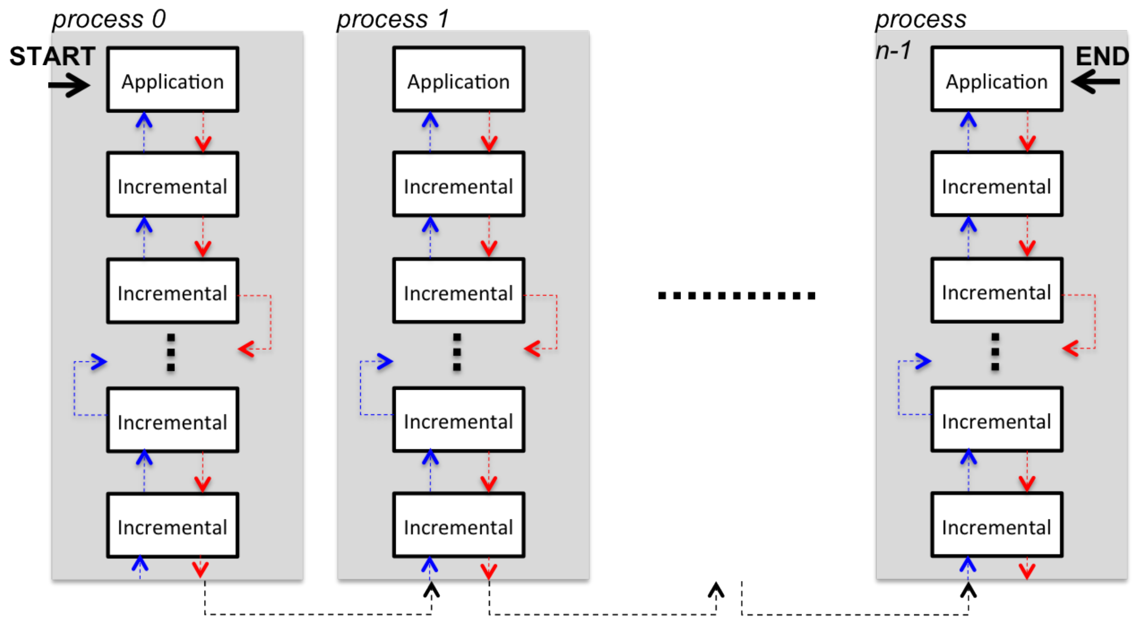

Figure 10.1: Application model for evaluation performance overhead

10.1 Execution overhead & Availability performance

In this section, we show execution overhead and availability performance by giving an example of application that is run on both Yaya and Neko.

10.1.1 Execution overhead time

In each process, we put 9 Incremental Protocol and 1 Application in the process. Application in process 0 starts to send a message to next Incremental Protocol and it then forwards to next Incremental Protocol in the same process. When finished passing message inside the process, the message is passed to next process. In received process, the message is delivered in reverse order and finally the message is delivered to Application. Application starts to send a new message again and pass down to the next process. The message is passed around all processes until the last process gets the message. The application model is shown in Figure 10.1

We run the simulation by increasing the number of processes. The result of execution overhead comparison between simulation in Neko and Yaya is shown in Figure 10.2 In

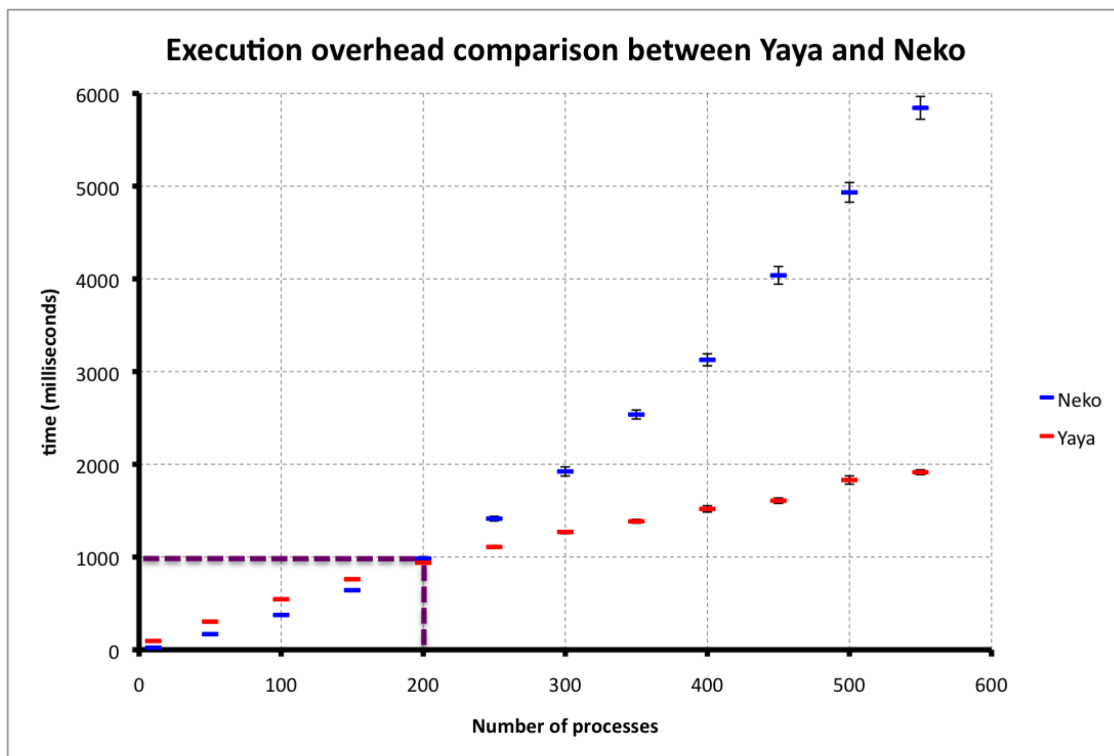

Figure 10.2: Execution overhead comparison between simulation in Yaya and Neko

Figure 10.2, we show execution time to finish this application depending on number of processes. When there are processes less than 200 processes approximately, the execution

time in Neko is less than Yaya, but when there are processes more than 200 processes, the execution time in Neko is greater than Yaya. In Neko, we can simulate around 560 processes (5600 active protocols) in the systems. In Yaya, we can create more than that. We explain the availability in next section.

10.1.2 Availability performance

In this evaluation, we use the same sample in section 10.1.1. The availability performance comparison between simulation in Neko and Yaya is shown in Figure 10.3 In Figure 10.3,

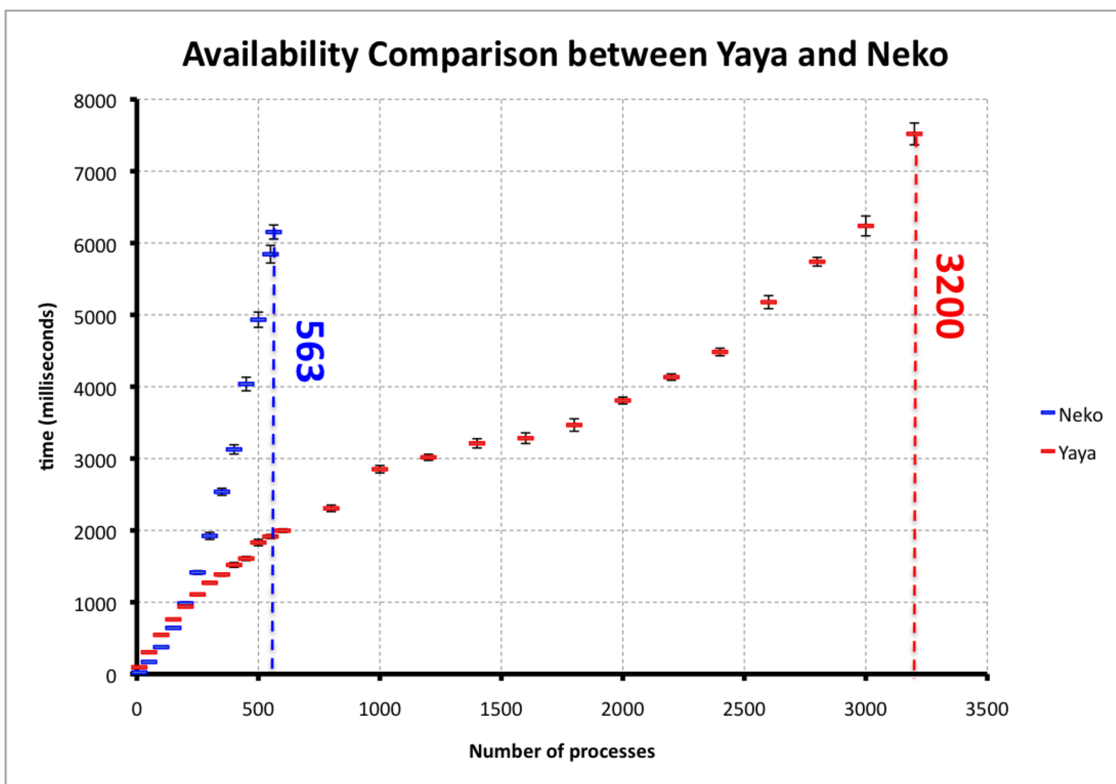

Figure 10.3: Availability comparison between simulation in Yaya and Neko

we show the number of processes that can be run simulation on Yaya and Neko. In Neko, we can simulate about 560 processes (each process has 10 active protocols, the total of active protocols is 5600 approximately). In Yaya, we can simulate about 3200 processes (each process has 10 active protocols, the total is 32000 active protocols approximately). However, simulation that has processes more than 2800 processes is getting slow until 3200 processes that we can not simulate anymore.

10.2 Discussion

In this chapter, we evaluate the performance overhead in terms of initialization time, execution time and availability of Yaya framework comparing with Neko framework. The result of initialization shows that simulation in Yaya has much overhead in initialization than Neko around 10 times. In initialization, all processes look for their own configuration protocols, interactions and some aspects in each protocol. Everything is in one configuration file. The result of execution time shows that execution time on Yaya is almost same as execution time on Neko. If there is much memory (heap memory space) for simulation, the execution time on Neko is less than Yaya. On the other hand, when we simulate a lot of protocols in the simulation, execution time on Yaya is less than Neko. Moreover, the result of availability performance in both Neko and Yaya shows how many maximum active protocol can be simulated in the simulation. We can simulate around 5600 active protocols in Neko but we simulate around 32000 active protocols in Yaya.

Chapter 11

Conclusion & Open Questions

11.1 Conclusion

In our research, we address the expressiveness in protocol composition framework by using actor model for rapid prototyping. We contribute the prototype of protocol composition framework implemented by actor model in Scala language. We evaluate the expressiveness of protocols (e.g. Chandra-Toueg consensus, Paxos consensus, Atomic broadcast using consensus) that are implemented on our framework in terms of line of code and analyze each part comparing with Neko framework. In implementing algorithm on framework, we separate code into 2 parts boilerplate code and effective code. We illustrate the ratio between that two parts by protocol examples and we then show the percentage of reduced code in our framework from Neko framework. Finally, we evaluate performance overhead in our framework comparing with Neko in terms of initialization time, execution time and availability. We examine the increasing initialization time, execution time and how many protocols that can be simulated in our framework.

11.2 Open Questions

As part of our continuing research, in our framework, there are hindrances in interactions between protocols. The important hindrances is how to standardize the content type between protocols. Protocol that has interactions with others needs to know which content type is interested. Protocol programers need to ensure that declaration of their interested contents are standard and other protocol can use this content to communicate with their protocols. We emphasize the framework that can ease to compose protocols without understanding the inside mechanism of each protocol.

Appendix A

Reliable broadcast

In this section, we show pseudocode of reliable broadcast algorithm.

```
Initialization:
  already ← φ                    {delivered messages list (message, id, receivedNumber)}
  id ← 0
proc send(m)
   newId ← nextId(id, process)                              {get next id of message}
   already = already + {m, newId, 0}
   send (m, newId) to destinations.
when receive (m, id)
  if id ∉ already                                       {first time receive message m}
     already = already + {m, newId, 1}
     deliver (m)                                       {deliver message m to listener}
     send (m, id) to destinations fi               {send message m to others again}
  else                                          {already received message m before}
    already = already update (m, id, receivedNumber + 1)
```

Appendix B

Consensus algorithm

B.1 Chandra-Toueg consensus algorithm [11]

In this section, we show pseudocode of Chandra-Toueg consensus algorithm.

proc propose(v_p) $\equiv$
  $estimate_p \leftarrow v_p$ {$estimate_p$ is p's estimate of the decision value}
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$ {r_p is p's current round number}
  $ts_p \leftarrow 0$ {ts_p is the last round in which p updated $estimate_p$}
  while $state_p = undecided$ do {rotate through coordinators until decision reached}
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ {c_p is the current coordinator}
    Phase 1: {all processes p send $estimate_p$ to the current coordinator}
     if $r_p > 1$ **then**
      $send(p, r_p, estimate_p, ts_p)$ *to* c_p fi
    Phase 2: {coordinator gathers $\lceil \frac{n+1}{2} \rceil$ estimates and proposes new estimate}
     if $p = c_p$ **then**
      if $r_p > 1$ **then**
       wait until[for $\lceil \frac{n+1}{2} \rceil$ processes q : received$(q, r_p, estimate_q, ts_q)$ from q]
       $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \parallel p$ received$(q, r_p, estimate_q, ts_q)$ from $q\}$
       $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
       $estimate_p \leftarrow$ select one $estimate_q \neq \perp$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$ fi
      send$(p, r_p, estimate_p)$ to all fi
    Phase 3: {all processes wait for new estimate proposed by current coordinator}
     wait until[received$(c_p, r_p, estimate_{c_p})$ from c_p or $c_p \in D_p$]{query failure detector D_p}
     if [received$(c_p, r_p, estimate_{c_p})$ from c_p] **then** {p received $estimate_{c_p}$ from c_p}
      $estimate_p \leftarrow estimate_{c_p}$
      $ts_p \leftarrow r_p$
      send(p, r_p, ack) *to* c_p fi
     else {p suspects that c_p crashed}
     send$(p, r_p, nack)$ *to* c_p

Phase 4:{the current coordinator waits for replies: $\lceil \frac{n+1}{2} \rceil$ acks or 1 nack. If they indicate that $\lceil \frac{n+1}{2} \rceil$ processes adopted its estimate, the coordinator R-broadcasts a decide message}

if $p = c_p$ **then**

wait until [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack) or for 1 process q : $(q, r_p, nack)$]

if [for $\lceil \frac{n+1}{2} \rceil$ processes q : received (q, r_p, ack)] **then**

$R - broadcast\ (p, estimate_p, decide)$ fi fi od.

{if p $R - delivers$ a decide message, p decides accordingly}

when $R - deliver\ (q, estimate_q, decide)$

if $state_p = undecided$ then

$decide(estimate_q)$

$state_p \leftarrow decided$

B.2 Mostéfaoui and Raynal consensus algorithm [11]

In this section, we show pseudocode of Mostéfaoui and Raynal consensus algorithm.

```
proc propose(v_p) ≡
  estimate_p ← v_p                      {estimate_p is p's estimate of the decision value}
  state_p ← undecided
  r_p ← 0                               {r_p is p's current round number}
  while state_p = undecided do          {rotate through coordinators until decision reached}
      r_p ← r_p + 1
      c_p ← (r_p mod n) + 1             {c_p is the current coordinator}
      est_from_c_p ←⊥                   {est_from_c_p is the coordinator's estimate or invalid(⊥)}
      Phase 1:{coordinator proposes new estimate; other processes wait for this new estimate}
       if p = c_p then
         est_from_c_p ← estimate_p fi
       else
         wait until[received(c_p, r_p, est_from_c_{c_p}) from c_p or c_p ∈ D_p]{query failure detector D_p}
         if [received(c_p, r_p, est_from_c_{c_p}) from c_p] then{p received est_from_c_{c_p} from c_p}
           est_from_c_p ← est_from_c_{c_p} fi
       send (p, r_p, est_from_c_p) to all
      Phase 2:{each process waits for ⌈(n+1)/2⌉ replies. If they indicate that ⌈(n+1)/2⌉ processes
                 adopted the proposal, the process R − broadcasts a decide message}
       wait until [for ⌈(n+1)/2⌉ processes q : received (q, r_p, est_from_c_q)]
       rec_p ← (q, r_p, est_from_c_q) ‖ p received (q, r_p, est_from_c_q) from q
       if rec_p = {v} then
         estimate_p ← v
         R − broadcast(p, estimate_p, decide) fi{R − broadcast without the initial send to all}
       else if rec_p = {v, ⊥} then
             estimate_p ← v fi od.
                              {if p R − delivers a decide message, p decides accordingly}
when R − deliver(q, estimate_q, decide)
 if state_p = undecided then
   decide(estimate_q)
   state_p ← decided
```

B.3 Paxos consensus algorithm [8]

```
Initialization                                                {variable initialization}
  estimate_{p_i} ←⊥
  state_{p_i} ← undecided
  r_{p_i} ← i
  read_{p_i} ← 0
  write_{p_i} ← 0
proc propose(v_{p_i})
   estimate_{p_i} ← v_p                  {estimate_{p_i} is p_i's estimate of the decision value}
   while state_p = undecided do
         if p_i ∈ D_{p_i} then
           phaseOneOk = true
           Phase 1:                                                        {READ phase}
           if r_{p_i} > 1 then                  {send READ iff we are not on the first round}
             send(r_{p_i}, read) to all
             wait until [received (r_{p_i}, estimate_{p_j}) or (r_{p_i}, nackread) from ⌈(n+1)/2⌉ processes]
             if  received only ackread then                                       {only ack}
               estimate_{p_i} ← select estimate_{p_j} with highest write_{p_j} among all ackread received fi
             else
               phaseOneOk = false fi
           if phaseOneOk then
             Phase 2:                                                      {WRITE phase}
               send(r_{p_i}, estimate_{p_i}, write) to all
               wait until [received (r_{p_i}, ackwrite) or (r_{p_i}, nackwrite) from ⌈(n+1)/2⌉ processes]
             if  received only ackwrite then                                      {only ack}
               R − broadcast(p_i, estimate_{p_i}, decide) fi fi
           r_{p_i} ← r_{p_i} + n fi od.                                    {increment round}
                                                        {process messages sent by the leader}
when receive(m) from p
  if m = (r_{p_j}, read) then                                      {process a read message}
    if write_{p_i} > r_{p_j} or read_{p_i} > r_{p_j} then{p_i has received a write/read with a higher round than p_j}
      send(r_{p_j}, nackread) to p_j fi
    else            {p_i has not received a write/read from a process with a higher round}
      read_{p_i} ← r_{p_j}
      send(r_{p_j}, estimate_{p_i}, ackread) to p_j fi
  if m = (r_{p_j}, estimate_{p_j}, write) then                    {process a write message}
    if write_{p_i} > r_{p_j} or read_{p_i} > r_{p_j} then{p_i has received a write/read with a higher round than p_j}
      send(r_{p_j}, nackwrite) to p_j fi
    else            {p_i has not received a write/read from a process with a higher round}
      write_{p_i} ← r_{p_j} ;  estimate_{p_i} ← estimate_{p_j}
      send(r_{p_j}, ackwrite) to p_j fi
```

{if p_i $R-delivers$ a decide message, p_i decides accordingly}

when $R-deliver(p_j, estimate_{p_j}, decide)$
  if $state_{p_i} = undecided$ then
    $decide(estimate_{p_j})$
    $state_{p_i} \leftarrow decided$ fi

Appendix C

Atomic broadcast algorithm

C.1 Simple fixed sequencer algorithm [10]

Code of sequencer :
Initialization:
$seqnum \leftarrow 0$ $\{$last seq. number attributed to a message$\}$
proc $TO-broadcast(m) \equiv$ $\{$To $TO-broadcast$ a message $m\}$
$increment(seqnum)$
send $(m, seqnum)$ to all
deliver (m).
when receive (m)
$TO-broadcast(m)$
Code of all processes except sequencer :
Initialization:
$lastdelivered_p \leftarrow 0$ $\{$sequence number of the last delivered message$\}$
$received_p \leftarrow \phi$ $\{$set of received yet undelivered messages$\}$
proc $TO-broadcast(m) \equiv$ $\{$To $TO-broadcast$ a message $m\}$
send(m) to sequencer.
when receive $(m, seq(m))$
$received_p \leftarrow received_p \cup \{(m, seq(m))\}$
while $\exists \acute{m}, seq$ s.t. $(\acute{m}, seq) \in received_p \wedge seq = lastdelivered_p + 1$ do
deliver $(\acute{m})$
increment $(lastdelivered_p)$
$received_p \leftarrow received_p$ $\{(\acute{m}, seq)\}$ od

C.2 Simple moving sequencer algorithm [10]

Initialization:
  $recvQ_p \leftarrow \epsilon$ {sequence of received messages (receive queue)}
  $seqQ_p \leftarrow \epsilon$ {sequence of messages with a seq. number}
  $lastdelivered_p \leftarrow 0$ {sequence number of the last delivered message}
  $toknext_p \leftarrow p + 1(\text{mod } n)$ {identity of the next process along the logical ring}
  if $p = p_1$ then
    send $(\perp, 0, 1)$ to p_1 fi {format: (message, seq.number, next token holder)}
proc $TO - broadcast(m)$ {To $TO - broadcast$ a message m}
  send (m) to all
  $recvQ_p \leftarrow recvQ_p \triangleright m.$
when receive (m)
  $recvQ_p \leftarrow recvQ_p \triangleright m$
when receive $(m, seqnum, tokenholder)$
  if $m \neq \perp$ then {Receive new sequence number}
    $seqQ_p \leftarrow seqQ_p \triangleright (m, seqnum)$ fi
  if $p = tokenholder$ then {Circulate token, if appropriate}
    wait until $(recvQ_p \; seqQ_p) \neq \epsilon$
    $msg \leftarrow$ select first msg in $recvQ_p$ s.t. $(msg, -) \notin seqQ_p$
    send $(msg, seqnum + 1, toknext_p)$ to all
    $seqQ_p \leftarrow seqQ_p \triangleright (msg, seqnum + 1)$ fi
  while $\exists \acute{m}$ s.t. $(\acute{m}, lastdelivered_p + 1) \in seqQ_p \wedge \acute{m} \in recvQ_p$ do {Deliver messages that can be}
      $seqQ_p \leftarrow seqQ_p - \{(\acute{m}, -)\}$
      $recvQ_p \leftarrow recvQ_p - \{\acute{m}\}$
      deliver $(\acute{m})$
      increment $(lastdelivered_p)$

C.3 Simple privilege-based algorithm [10]

```
Initialization:
  sendQ_p ← ε                                   {sequence of messages to send(send queue)}
  recvQ_p ← ε                                   {sequence of received messages(receive queue)}
  lastdelivered_p ← 0                           {sequence number of the last delivered message}
  toknext_p ← p + 1(mod n)                      {identity of the next process along the logical ring}
  if p = p_1 then                               {virtual message to initiate the token rotation}
    send (⊥, 0, 1) to p_1 fi                    {format: (message,seq.number,next token holder)}
proc TO − broadcast(m)                          {To TO − broadcast a message m}
   if m ≠⊥ then                                 {Receive new messages}
     recvQ_p ← recvQ_p ▷ (m, seqnum) fi.
when receive (m, seqnum, tokenholder)
  if m ≠⊥ then                                  {Receive new messages}
    recvQ_p ← recvQ_p ▷ (m, seqnum) fi
  if p = tokenholder then                       {Circulate token, if appropriate}
    if sendQ_p ≠ ε then                         {Send pending messages, if any}
      msg ← head.sendQ_p
      sendQ_p ← tail.sendQ_p
      send (msg, seqnum + 1, toknext_p) to all
      recvQ_p ← recvQ_p ▷ (msg, seqnum + 1) fi
    else
      send (⊥, seqnum, toknext_p) to toknext_p fi
while ∃ḿ s.t. (ḿ, lastdelivered_p + 1) ∈ recvQ_p do     {Deliver messages that can be}
      recvQ_p ← recvQ_p − {ḿ}
      deliver (ḿ)
      increment (lastdelivered_p) od
```

C.4 Simple communication history algorithm [10]

Initialization:

$received_p \leftarrow \phi$ {Messages received by process p}

$delivered_p \leftarrow \phi$ {Messages delivered by process p}

$deliverable_p \leftarrow \phi$ {Messages ready to be delivered by process p}

$LC_p[p_1, ..., p_n] \leftarrow \{0, ..., 0\}$ {$LC_p[q]$: logical clock of process q as seen by process p}

proc $TO - broadcast(m)$ {To $TO - broadcast$ a message m}

$LC_p[p] \leftarrow LC_p[p] + 1$

send $(m, LC_p[p])$ to all

when no message sent for $\triangle_{live}$ time units

$LC_p[p] \leftarrow LC_p[p] + 1$

send $(\perp, LC_p[p])$ to all

when receive $(m, ts(m))$

$LC_p[p] \leftarrow$ max $(ts(m), LC_p[p]) + 1$ {Update logical clock}

$LC_p[sender(m)] \leftarrow ts(m)$

$received_p \leftarrow received_p \cup \{m\}$

$deliverable_p \leftarrow \phi$

for each message $\acute{m}$ in $received_p \setminus delivered_p$ **do**

if $ts(\acute{m}) < \min_{q \in x(t)} LC_p[q]$ **then**

$deliverable_p \leftarrow deliverable_p \cup \{\acute{m}\}$ fi

deliver all messages in $deliverable_p$, according to the total order

$delivered_p \leftarrow delivered_p \cup deliverable_p$

C.5 Using Consensus to solve Atomic broadcast [9]

Every process p executes the following:
Initialization: $\{\text{variable initialization}\}$
$R_delivered \leftarrow \phi$
$A_delivered \leftarrow \phi$
$k \leftarrow 0$
To execute $A - broadcast(m)$;
$R - broadcast(m)$
$A - deliver(-)$ occurs as follows:
when $R - deliver(m)$
$R_delivered \leftarrow R_delivered \cup \{m\}$
when $R_delivered - A_delivered \neq \phi$
$k \leftarrow k + 1$
$A_undelivered \leftarrow R_delivered - A_delivered$ $\{\text{get undelivered messages list}\}$
$propose(k, A_undelivered)$
wait until $decide(k, msgSet^k)$
$A_deliver^k \leftarrow msgSet^k - A_delivered$
atomically deliver all messages in $A_deliver^k$ in some deterministic order
$A_delivered \leftarrow A_delivered \cup A_deliver^k$

Bibliography

[1] Anand Ranganathan and Roy H. Campbell, "What is the Complexity of a Distributed System?," Technical Report UIUCDCS-R-2005-2568, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois, USA, April 2003.

[2] N. Bhatti and R. Schlichting, "A System for Constructing Configurable High-Level Protocols." *Proc. ACM SIG COMM '95*, 138-150, 1995

[3] R. Van Renesse, K.P. Birman, R. Friedman, M. Hayden, and D.A. Karr, "A framework for protocol composition in Horus." *In Proceedings of the 14th Symposium on the Principles of Distributed Computing ACM* (Ottawa, Ont., Aug. 1995), pp. 80-89.

[4] Bünzil, Daniel C., Mena, Sergio, Nestmann, Uwe, "Protocol Composition Frameworks, A Header Driven Model," *In Proceedings of the 4th International Symposium on Network Computing and Applications* (IEEE NCA05), 2005.

[5] S. Mena, X. Cuvellier, C. Grégoire, A. Schiper, "Appia vs. Cactus: Comparing Protocol Composition Frameworks," *In Proc. of SRDS,* 2003.

[6] P. Urbán, S. Mena, X. Défago and T. Katayama, "Concurrency in Microprotocol Frameworks," *JAIST,* IS-RR-2006-004, 2006.

[7] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithm," *In Proc. of the 15th Intl Conf. on Information Networking (ICOIN-15),* Beppu City, Japan, Feb. 2001.

[8] N. Hayashibara, P. Urbán, A. Schiper, and T. Katayama, "Performance comparison between the Paxos and Chandra-Toueg consensus algorithms," *In Proc. Int'l Arab Conf. on Information Technology (ACIT 2002),* pages 526-533, Doha, Qatar, December 2002.

[9] T.D. Chandra, S. Toueg, "Unreliable failure detectors for reliable distributal systems. *J. ACM 43, 2*(Mar.), 225-267.

[10] X. Défago, A. Schiper, P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey." *ACM Comput. Surv. 36,* 4, 372-421, 2004

[11] P. Urbán, “Evaluating the performance of distributed agreement algorithms: tools, methodology and case studies.” PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2003.

[12] A. Mostéfaoui and M. Raynal, “Solving consensus using Chandra-Toueg's unreliable failure detectors: a general quorum-based approach.” *In Proceedings of the 13th International Symposium on Distributed Computing (DISC'99),* pages 49-63. LNCS, Springer-Verlag, September 1999.

[13] C. Hewitt, P. Bishop and R. Steiger, “A universal modular actor formalism for artificial Intelligence,” *Int. Joint Conf. Artificial Intelligence* 1973, Stanford University, Stanford (August 1973) 235-245.

[14] G. Agha, “Actors: A model of Concurrent Computation in Distributed Systems,” The MIT Press, Cambridge, Massachusetts, 1986.

[15] P. Haller and M. Odersky, “Actors That Unify Threads and Events,” *In 9th International Conference on Coordination Models and Languages,* volume 4467 of Lecture Notes in Computer Science. Springer, 2007.

[16] P. Haller and M. Odersky, “Event-Based Programming without Inversion of Control,” *In Proc. Joint Modular Languages Conference*, Springer LNCS, 2006.

[17] H. Miranda, A. Pinto, and L. Rodrigues, “Appia, a flexible protocol kernel supporting multiple coordinated channels.” *In Proceedings of The 21st Int'l Conf. on Distributed Computing Systems (ICDCS-21),* pages 707-710, Phoenix, Arizona, USA, Apr.16-19 2001. IEEE Computer Society.

[18] M. A. Hiltunen and R. D. Schlichting, “The Cactus approach to building configurable middleware services.” *In Proc. Work- shop on Dependable System Middleware and Group Communication (DSMGC 2000),* Nurnberg, Germany, Oct. 2000.

[19] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. “Real-time dependable channels: Customizing QoS attributes for distributed systems.” *IEEE Transactions on Parallel and Distributed Systems, 10(6):600?612,* Jun 1999.