INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Adaptive Group Communication

Tiago José Pinto Taveira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente:	Prof. Doutor Alberto Manuel Ramos da Cunha
Orientador:	Prof. Doutor Luís Eduardo Teixeira Rodrigues
Vogal:	Prof. Doutor Vítor Manuel Alves Duarte

October 2010

Agradecimentos

First I would like to thank my advisor, Prof. Luís Rodrigues, for his endless support and patience throughout this dissertation work. His experience and guidance proved itself invaluable.

I would also like to say a special thanks to Liliana Rosa, who was always there to help. I am also grateful to all the other members of the Distributed Systems Group at INESC-ID, particularly João Leitão, Nuno Carvalho, and José Mocito.

Finally, my sincere thanks to my parents, my brother, and all my family, for the encouragement and words that gave me the motivation I needed.

This work was partially funded by the research grant PTDC/EIA/71752/2006 in the context of the Redico project and by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

Lisboa, October 2010

Tiago José Pinto Taveira

To my parents, José and Maria, and
my brother, Bruno.

Resumo

Comunicação em grupo corresponde a um conjunto de serviços de filiação, comunicação, e coordenação, que suportam o desenvolvimento de aplicações distribuídas baseadas em grupos de processos. Estes serviços são tipicamente providenciados por uma pilha de protocolos, que inclui camadas como detecção de falhas, difusão fiável, sincronia na vista, ordem total, entre outros. O desempenho destes protocolos é também altamente dependente das condições em que o sistema executa, tais como a latência da rede, taxa de erros da ligação, carga, etc. Assim, múltiplas implementações destas camadas foram propostas, cada uma possuindo um desempenho óptimo em determinadas condições. Este trabalho aborda as arquitecturas e mecanismos necessários para a construção de plataformas de composição de protocolos adaptativas, nomeadamente as que suportam comunicação em grupo, para que as mesmas possam lidar com um leque vasto de requisitos.

Abstract

Group communication denotes a set of membership, communication, and coordination services that support the development of distributed applications based on process groups. These services are typically provided by a protocol stack, which includes layers such as failure detectors, reliable multicast, view-synchrony, total order, among others. The performance of these protocols is highly dependent on the operational envelope, including network latency, link error rates, load profile, etc. Therefore, multiple implementations of each layer have been proposed, each excelling in a different scenario. This work addresses the architectures and mechanisms required to build adaptive protocol composition frameworks, namely those supporting group communication, such that they can cope with a wide range of varying requirements.

Palavras Chave
Keywords

Palavras Chave

Sistemas Adaptativos

Comunicação em Grupo

Plataformas de Composição de Protocolos

Keywords

Adaptive Systems

Group Communication

Protocol Composition Frameworks

Índice

1 Introduction **3**

1.1 Motivation . . . 3

1.2 Contributions . . . 5

1.3 Results . . . 5

1.4 Research History . . . 5

1.5 Structure of the Document . . . 6

2 Related Work **7**

2.1 Introduction . . . 7

2.2 Group Communication . . . 7

2.2.1 Membership . . . 8

2.2.2 Reliable Multicast . . . 10

2.2.3 Examples of Group Communication Systems . . . 12

2.2.4 The Appia Group Communication Stack . . . 12

2.3 Adaptive Group Communication Systems . . . 14

2.3.1 Configurable vs Adaptive Systems . . . 14

2.3.2 Adaptive Failure Detectors . . . 15

2.3.3 Adaptive Total Order . . . 15

2.3.4 Monolithic Solutions vs Modular Protocol Switching . . . 16

2.3.5 Protocol Composition Frameworks . . . 17

2.3.6 Requirements for Dynamic Adaptation 17

2.3.7 Cactus Framework and Switching Protocol 18

2.3.8 Other Switching Protocols 20

2.3.9 When to Adapt 21

2.4 *RAppia* 22

2.5 Discussion 24

3 RAppia 2.0 27

3.1 Introduction 27

3.2 Basic Appia Concepts 27

3.3 The RAppia Architecture 28

3.4 Previous Implementations of RAppia 29

3.4.1 RAppia 0.5 30

3.4.2 RAppia 1.0 31

3.5 RAppia 2.0 32

3.5.1 Overview 32

3.5.2 Adaptation Manager 33

3.5.2.1 Manager 33

3.5.2.2 Context Monitor 34

3.5.3 Reconfiguration Agent 34

3.5.3.1 Reconfigurator 35

3.5.3.2 Sensor 35

3.5.4 Interpreter 36

3.5.5 Event Scheduler 39

3.6 Refactoring Existing Protocols for Adaptation Support 40

3.6.1 Appia Group Communication Protocols 41
3.6.1.1 Sequencer-based Total Order Quiescence 43
3.6.1.2 Token-based Total Order Quiescence 44
3.7 Platform Example 44
3.7.1 Gathering Context Information 44
3.7.2 Policy Definition and Application 45
3.7.3 Switching Total Order Services 47

4 Evaluation **55**
4.1 Introduction 55
4.2 Adaptation Discussion 55
4.3 Experimental Settings 56
4.4 Adaptation Mechanisms Performance 57
4.4.1 Event Scheduling 57
4.4.2 Message Model 58
4.4.3 Adaptation Performance 59

5 Conclusions **63**
5.1 Conclusions 63
5.2 Future Work 64

Bibliography **68**

List of Figures

1.1 The latency of three total order algorithms for an increasing system load. 4

2.1 The structure of an Adaptive Component (AC) in *Cactus*. 18

2.2 Service replacement with state transfer in *RAppia*. 23

3.1 The implementation of an application that uses separate audio and video channels. 29

3.2 RAppia's System Model 29

3.3 The integration of the Reconfigurator and Sensor components into an existing stack. 35

3.4 The execution of an *activate reconfiguration* action. 38

3.5 The representation of a channel composition for two channels containing shared sessions. 40

3.6 The stacks used in the reconfiguration of total order services, with the shared sessions highlighted. 48

List of Tables

4.1 Time messages take to go through all layers implementing view-synchrony. 58

4.2 Message size in a view-synchronous stack for both Appia 4.1.0 and RAppia . . . 58

4.3 Size of adaptation events sent from the Adaptation Manager to each of the system nodes, for a total order protocol switch. 59

4.4 Size of messages requesting and sending context information, between the Context Monitor and the Sensor components. 59

4.5 Time spent for each adaptation event in a total order switch scenario, for 3 nodes. 60

4.6 Time spent for each adaptation event in a total order switch scenario, for 5 nodes. 60

4.7 Time spent in Appia for the adaptation process in a total order switch scenario, for 3 and 5 nodes. 61

Acronyms

AAM Adaptation-aware Algorithm Module

ACM Association for Computer Machinery

AC Adaptive Component

CAM Component Adaptor Module

DAC Distributed Adaptive Component

ECA Event-Condition-Action

FIFO First-in First-out

GCS Group Communication System

GC Group Communication

IEEE Institute of Electrical and Electronics Engineers, Inc.

JMX Java Management Extensions

LAN Local Area Network

QoS Quality of Service

RTT Round-trip Time

SETO Statistically Estimated Total Order

TCP Transmission Control Protocol

1 Introduction

Group communication denotes a set of membership, communication, and coordination services that support the development of distributed applications based on process groups (Birman 1993). It has been widely used to build multi-participant applications, such as collaborative applications (Rhee, Cheung, Hutto, Krantz, & Sunderam 1999), and fault-tolerant replicated services, for instance, database replication systems (Pedone, Guerraoui, & Schiper 1998).

The services provided by group communication include failure detection, membership, and reliable multicast communication with different ordering properties (including FIFO order, causal order, and total order) (Powell 1996). These services are typically provided by a protocol stack, where each service is implemented by one or multiple layers. Systems that use this kind of protocol stacks are Horus (van Renesse, Birman, & Maffeis 1996), Ensemble (Birman, Constable, Hayden, Hickey, Kreitz, Van Renesse, Rodeh, & Vogels 1999), Cactus (Chen, Hiltunen, & Schlichting 2001), Appia (Miranda, Pinto, & Rodrigues 2001), among others.

Since there is a large number of protocols implementing group communication services, each with its own advantages and disadvantages, it makes sense to provide programmers with the tools to better adapt the system to their specific needs. This work addresses the architectures and mechanisms required to build adaptive protocol composition frameworks, namely those supporting group communication, such that they can cope with a wide range of varying requirements.

1.1 Motivation

The performance of group communication protocols is highly dependent on the operational envelope, including network latency, link error rates, workload, etc. Therefore, multiple implementations of each service have been proposed. As an example, consider the case of total order multicast, also known as atomic multicast. A survey of existing alternatives to implement

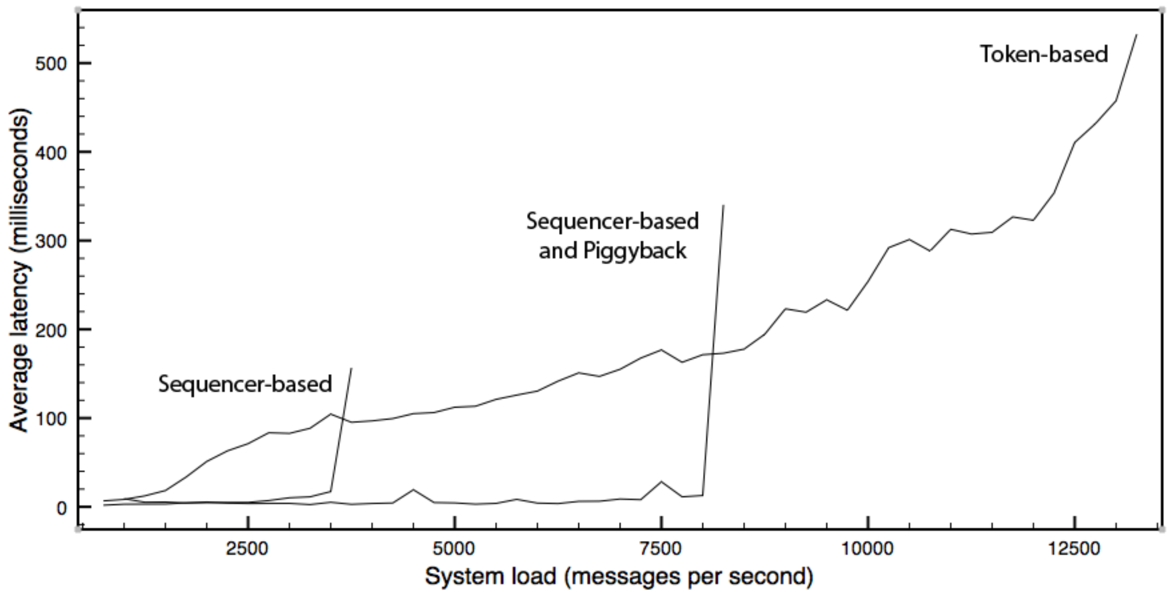

Figure 1.1: The latency of three total order algorithms for an increasing system load.

this service identified about sixty protocols (Défago, Schiper, & Urbán 2004). None of these protocols outperforms the others in all scenarios. Instead, each implementation offers better results on a specific network setting and/or in face of a particular workload. Figure 1.1 presents three possible total order configurations (sequencer-based, sequencer-based and a piggybacking mechanism, and a token-based total order), that offer very different capabilities regarding their maximum load and corresponding latency. In this case, the token-based approach supports the highest load, at the expense of an increased latency.

Given that it is often very difficult, or even impossible, to estimate, when the system is deployed, what will be the workload and the operational conditions, it is quite hard to select offline the best protocol stack configuration. Furthermore, some aspects of the operational envelope are dynamic, and change during its operation. For instance, the network load varies significantly depending on the time of the day. This motivates the need to develop adaptive group communication services, that are able to change configuration parameters on the fly, or even replace a service implementation by a more suitable alternative, in response to observed changes in the execution context.

Despite all the advances in the area of dynamic adaptation of group communication services, the implementation of autonomic systems that use group communication remains a significant challenge. This work addresses the problem of building such adaptive group communication services.

1.2 Contributions

This work addresses the problem of adaptive group communication, More precisely, the thesis analyzes, designs, implements, and evaluates techniques to build an adaptive protocol composition framework, with support for the dynamic adaptation of group communication services. As a result, the thesis makes the following contributions:

- the definition of a set of components that aid in the runtime adaptation of group communication protocols;
- the specification of a generic procedure to perform protocol switching in the context of a group communication system.

1.3 Results

The results produced by this thesis can be enumerated as follows:

- A prototype of a new RAppia release, a framework supporting the dynamic adaptation of protocol stacks.
- An adaptable version of a group communication protocol suite.
- An experimental evaluation of the implemented system, compared to the previous, non-configurable composition framework, Appia.

1.4 Research History

This work was performed in the context of the Redico project. One of this project's main goals is to address the problem of building adaptive communication protocols. In particular, Redico focus on two research areas: the software tools to design, implement, deploy, and execute protocols that can be reconfigured in runtime; and the study of complex protocols, such as the ones involved in multi-participant systems, that dynamically adapt to changes in the operational envelope.

The initial objective of this work was to use an existing adaptive protocol composition framework, RAppia, in order to provide an adaptive version of the Appia group communication system. In particular, this work aimed at defining a set of policies that would select the most appropriate implementations for Appia's group communication stack, for example by modifying the total order or the failure detector protocols used. However, experiments with the previous RAppia prototype highlighted a number of limitations. This forced a shift in the emphasis of the work, as a new release of the RAppia framework had to be produced.

During my work, I benefited from the fruitful collaboration with the remaining members of the GSD team working on Redico, in particular from Liliana Rosa.

1.5 Structure of the Document

The rest of this document is organized as follows. Chapter 2 provides an introduction to the different technical areas related to this work. Chapter 3 introduces RAppia 2.0, a new version of the adaptive protocol composition framework, as well as the description of the work required to update the existing group communication protocol suite to operate with the new kernel. Chapter 4 presents the results of the experimental evaluation study. Finally, Chapter 5 concludes this document by summarizing its main points and providing directions for future work.

2 Related Work

2.1 Introduction

This chapter provides an overview of the related work relevant to the project. It starts by describing the fundamental concepts behind group communication, the services typically provided by group communication systems, and the main applications of this technology. Then, the need for adaptive communication systems is motivated. Subsequently, the architectures and runtime support that permit the construction these systems is addressed, with particular emphasis on the original version of *RAppia*, a protocol composition and execution framework designed specifically to support dynamic adaptation.

2.2 Group Communication

Group communication is a designation that is used to refer to a set of communication and coordination services that aim at supporting the development of distributed applications where a *group* of processes need to exchange information and coordinate to perform a common task. For example, a group can be a set of users communicating using a chat system or playing an online game (Rhee, Cheung, Hutto, Krantz, & Sunderam 1999). A group of processes can also be formed to replicate a given component for fault-tolerance: each group member is a replica, and all members process the same set of requests from clients (Pedone, Guerraoui, & Schiper 1998). Finally, processes may coordinate to distribute tasks among them (Khazan, Fekete, & Lynch 1998).

There are two primary services provided by a group communication system (Chockler, Keidar, & Vitenberg 2001): *membership* and *multicast communication.*

2.2.1 Membership

The purpose of the membership service is to provide to each participant up-to-date information about current members of the group (Hiltunen & Schlichting 1998). Such information is usually called a *group view*, or simply a view. The interface of this service allows processes to *join* a group and to voluntarily *leave* the group. Processes may also abandon a group involuntarily as a result of a fault.

Whenever a change in the group membership occurs, a new view is *delivered* to the application. In this case we state that a *view change* has occurred. Typically, each view has an unique identifier and these identifiers are assigned such that delivered views have monotonically increasing identifiers. When an application receives and processes a new view, it is common to say that the application *installs* that view.

It may happen that concurrent changes to the group membership are detected at different instants, and even in different orders, at each process. However, to simplify the coordination at the application level, group communication services execute agreement protocols before delivering views, to ensure that all participants obtain a consistent perception of the system evolution (Cachin 2005).

It is possible to devise different membership services, that ensure different properties of the delivered views in face of events such as concurrent joins and leaves, faults or network partitions. For a network partition, which happens when a group of nodes can no longer communicate with the rest as a result of a broken link, it is important to distinguish *primary partition membership* and *partitionable membership* services.

With primary partition membership, one of the resulting partitions in the network is denoted as the primary partition, and only nodes that belong to it are allowed to deliver messages and views. In a partitionable membership service, no restrictions are imposed for message and view delivery, and concurrent views may exist. When a network partition is healed, these concurrent views may be *merged* in a single view. Systems that use a primary partition membership service include ISIS (Birman & Joseph 1987) and Phoenix (Malloth, Felber, Schiper, & Wilhelm 1995). Partitionable membership was first introduced as part of Transis (Dolev & Malki 1996), and is also present in Totem (Moser, Melliar-Smith, Agarwal, Budhia, & Lingley-Papadopoulos 1996), Horus (van Renesse, Birman, & Maffeis 1996), and Appia (Miranda, Pinto, & Rodrigues 2001), among others.

As previously mentioned, a view change can be triggered by a member unpredictably leaving the group due to a fault. This sort of event is detected by a component of the group communication system called a *failure detector*. In a synchronous environment, it is possible to implement a *perfect* failure detector, which is guaranteed to eventually detect a faulty process and never incorrectly detects a correct process as failed. On the other hand, in an asynchronous environment one can make no assumptions regarding the time it takes for a process to complete an execution step or a message transmission. Because of this, it is only possible to implement *unreliable* failure detectors, which can make mistakes. More precisely, a failure detector can be characterized by two properties (Chandra & Toueg 1996), *completeness* and *accuracy*:

- *Completeness.* There is a time after which every process that crashes is permanently suspected by some correct process (which corresponds to the actual ability to detect failures).
- *Accuracy.* There is a time after which some correct process is never suspected by any correct process (which defines the kind of mistakes that can be made).

Based only on these two properties, one can define failure detectors that provide different Qualities of Service (QoS). Namely, we can measure how fast can a fault be detected, and on what degree are we willing to trade speed for a correct detection. The work by Chen, Toueg and Aguilera (Chen, Toueg, & Aguilera 2002) provides an insight about this, including three main metrics for the QoS specification of a failure detector:

- *Detection Time (T_P)*, is the time that elapses from p's crash to the time when another process q starts suspecting p permanently.
- *Mistake recurrence time (T_{MR})*, represents the time between two consecutive mistakes.
- *Mistake duration (T_M)*, measures the time it takes the failure detector to correct a mistake.

The first, detection time, measures the speed of a failure detector, which relates to the completeness property stated above. Mistake recurrence time and mistake duration are both accuracy metrics.

2.2.2 Reliable Multicast

The other main service of a group communication system is the multicast service. This service allows to send messages to all group members, typically using the group identifier as a multicast address (i.e., the sender is not required to list each individual recipient explicitly).

Most group communication services support *reliable* multicast. Informally, the guarantees of reliable multicast are the following (Chandra & Toueg 1996): i) all correct processes deliver the same set of messages, ii) all messages multicast by correct processes are delivered, and iii) no spurious messages are ever delivered.

In a system where the group membership is dynamic, reliability needs to be defined in relation to a given group view. Therefore, a message is sent to all members of the last installed view and should be delivered to all correct members of that view. This semantics is known as *view-synchronous reliable multicast*, and is characterized more precisely by the following properties (Cachin 2005):

- *Same-view-delivery.* If a process P_i sends a message m in some view V and a process P_j delivers m in view V', then $V = V'$.
- *(Regular) View-synchronous delivery.* If two processes P_i and P_j both install a new view V in the same previous view V', then any message delivered by P_i was also delivered by P_j in V'.
- *Integrity.* Every process delivers at most one copy of message m, and only if m was previously multicast by the associated sender.

It is also important to establish the distinction between *regular* and *uniform* reliable multicast. Regular reliable multicast means that if a message is delivered to a correct process, then all correct processes deliver the message. Uniform reliable multicast states that if a process (even if faulty) delivers a message, then all correct processes deliver it too. More precisely:

- *(Uniform) View-synchronous delivery.* If a process P_j installs a new view V in view V', then any message delivered by a process P_i in V' was also delivered by P_j in V'.

Different ordering policies may be enforced on the message exchange among group members. The most relevant are FIFO (First-In First-Out), Causal Order, and Total Order (Chockler, Keidar, & Vitenberg 2001). FIFO ordering implies that messages are delivered in the same order they were sent. This kind of ordering is often used as a basic building block to other guarantees. Causal order is stronger than FIFO; it ensures that messages are delivered according to the *happened-before* notion first defined by Lamport (Lamport 1978). Finally, total order multicast, sometimes also referred to as atomic multicast, ensures that all the messages sent in the system are delivered by all processes in the same order.

Given that this is an important service of any group communication system, we include here a more precise definition of this service. Total order is defined by two basic primitives, *TO-broadcast(m)* and *TO-deliver(m)*, where *m* is some message. The service has the following properties (Défago, Schiper, & Urbán 2004):

- *Validity.* If a correct process TO-broadcasts a message *m*, then it eventually TO-delivers *m*.
- *Uniform Agreement.* If a process TO-delivers a message *m*, then all correct processes eventually TO-deliver *m*.
- *Uniform Integrity.* For any message *m*, every process TO-delivers *m* at most once, and only if *m* was previously TO-broadcast by the sender of message *m*.
- *Uniform Total Order.* If processes *p* and *q* both TO-deliver messages *m* and *m'*, then *p* TO-delivers *m* before *m'*, if and only if *q* TO-delivers *m* before *m'*.

As with view-synchronous communication, it is possible to define non-uniform variants of the primitive, respectively:

- *Regular Agreement.* If a *correct* process TO-delivers a message *m*, then all correct processes eventually TO-deliver *m*.
- *Regular Total Order.* If two *correct* processes *p* and *q* both TO-deliver messages *m* and *m'*, then *p* TO-delivers *m* before *m'*, if and only if *q* TO-delivers *m* before *m'*.

In some settings, it is possible to implement the regular version of the service with more efficient algorithms, namely algorithms that exhibit lower latency (Défago, Schiper, & Urbán 2004). Given that not all applications require the more expensive uniform version, both variants have been implemented in existing systems (such as *Horus* or *Appia*).

2.2.3 Examples of Group Communication Systems

Group communication has been widely studied and many group communication systems have been implemented, including ISIS (Birman & Joseph 1987), Transis (Dolev & Malki 1996), Horus (van Renesse, Birman, & Maffeis 1996), Totem (Moser, Melliar-Smith, Agarwal, Budhia, & Lingley-Papadopoulos 1996), Spread (Amir & Stanton 1998), Ensemble (Cadot, Kuijlman, Langendoen, van Reeuwijk, & Sips 2001), Cactus (Hiltunen & Schlichting 2000), Phoenix (Malloth, Felber, Schiper, & Wilhelm 1995), JGroups[1], Appia (Miranda, Pinto, & Rodrigues 2001), among others.

The ISIS toolkit was the first to implement view synchronous communication. It was a monolithic implementation, that included most of the services listed above. The system had a commercial version used in several important deployments, such as in the NY stock exchange. Horus, and later Ensemble, were modular and improved versions of the ISIS system. Transis, Totem, and Spread added novel protocols, including protocols specialized for some network topologies, such as local area network or wide-area communication. Cactus is a highly modular and adaptive implementation of group communication. JGroups and Appia are open source group communication systems implemented in the Java language (and heavily inspired in the Horus/Ensemble systems).

2.2.4 The Appia Group Communication Stack

Appia is a Java-based protocol composition framework that offers a base protocol stack that implements view synchrony. This stack is composed by the following protocols (top to bottom):

- *VSync.* Ensures that a view change respects the *view-synchrony* property. It counts the number of delivered messages from each of the other group members, and when they match

[1]http://www.jgroups.org/

in all of the active members the view can be updated. The final view is announced by the view coordinator.

- *Leave.* Allows for a member to leave the group orderly by producing the *LeaveEvent.* This intention is announced to the view coordinator.

- *Stable.* This protocol is responsible for retransmitting messages that were received only by some members of the group. In order to do this, it determines which messages have not been received, and requests them to a member that possesses them (received messages are stored).

- *Heal.* Detects the existence of concurrent views in the same group, mainly by receiving messages that contain a different view than the one installed.

- *Inter.* Unifies the existing (concurrent) views from the same group, and executes a consensus algorithm to decide the new view. This process is done only by view coordinators.

- *Intra.* The *Intra* protocol acts as a view change manager, which is initiated either by a fault or by request. The change is handled by the view coordinator and in three main steps: *VSync* makes sure the change is correct, the new members in the view are determined by *Inter* and *Leave*, and the new view is delivered by the *View* event.

- *Suspect.* Implements a failure detector. A fault is announced with the *Fail* event which contains the current view's suspected peers.

- *Bottom.* Translates network to group identifications. Also filters events which do not belong to the current view.

Based on these protocols, other higher level properties and services can be provided. *Appia* includes three implementations of total order multicast: fixed sequencer (Défago, Schiper, & Urbán 2004), token based (part of the *privilege-based* class of algorithms (Défago, Schiper, & Urbán 2004)) and a statistically estimated total order (SETO) (Sousa, Pereira, Moura, & Oliveira 2002). The failure detector currently has two implementations, the φ accrual failure detector (Hayashibara, Defago, Yared, & Katayama 2004) and a heartbeat failure detector.

2.3 Adaptive Group Communication Systems

The performance of the protocols that implement group communication services, as many other protocols for distributed systems, are highly dependent of operational conditions such as observed load, network latency, available bandwidth, CPU and memory constraints, etc. It is therefore no surprise that many different protocols have been proposed for each of these services. Just to implement total order, about sixty protocols have been identified (Défago, Schiper, & Urbán 2004). Some perform better when all nodes transmit at the same pace, others perform better when the traffic is sporadic, some have been optimized for broadcast networks, others for networks with high latency, and so forth.

In the following sections we describe some key aspects in dealing with this kind of dynamic conditions.

2.3.1 Configurable vs Adaptive Systems

A group communication system is configurable if the most appropriate implementation may be selected at load time, when the service is instantiated. Examples of configurable group communication systems are Horus, Ensemble, Cactus, JGroups and Appia. A configurable group communication system allows the use of the protocol implementation that is most suitable for an expected set of operational conditions.

However, in many scenarios it is impossible to accurately estimate in advance the operational conditions. Furthermore, in many settings, the operational conditions change significantly with time. For instance, in many networks, the observed latency depends on the time of the day. Also, the level of risk in a network may change (for instance, if unusual behaviour is detected), which may require the use of protocols with stricter security features. Therefore, it is interesting to support the runtime reconfiguration of protocols. Also, if runtime reconfiguration is supported, existing protocols can be upgraded to new, better versions without requiring the application to stop (Liu, van Renesse, Bickford, Kreitz, & Constable 2001).

In the next paragraphs, we give two examples that illustrate the importance of supporting runtime adaptation of group communication services.

2.3.2 Adaptive Failure Detectors

Failure detectors are a fundamental building block of any group communication system. Typically, failure detection relies on some form of periodic *heartbeat* mechanisms, i.e., nodes need to periodically exchange information with each other to mutually check that they have not crashed. If no heartbeat is received after some defined *timeout* value, the node is considered to have failed. There are two aspects of the failure detection that may require adaptation: the value of the timeout, that needs to be adjusted to the observed network latency, and the communication pattern, i.e., how nodes exchange heartbeats.

Regarding the timeout values, this is a configuration parameter that has been recognized to need adaptation, even for point-to-point communication. The TCP protocol (Jacobson 1988) embodies an algorithm to adjust the timeout value in runtime, and that work has been extended to the multi-point scenario by various other works (Hayashibara, Defago, Yared, & Katayama 2004; Bertier, Marin, & Sens 2002).

Regarding the pattern of communication, a simple pattern is to use all-to-all communication. For instance, each node periodically sends a heartbeat to every other node in the system. This requires the exchange of n^2 control messages in each detection cycle. Other, more effective, communication patterns include: hierarchic failure detectors, in which processes are grouped in monitoring subnets in an attempt to reduce message explosion (Felber, Défago, Guerraoui, & Oser 1999) and gossip-style failure detection (Renesse, Minsky, & Hayden 1998). However, there is often a tradeoff between the amount of communication and the latency of the failure detection. Therefore, the most appropriate communication pattern must be chosen in function of the size of the group and of the desired responsiveness of the failure detector. A summary on scalable failure detection services can be found in (Hayashibara, Cherif, & Katayama 2002).

2.3.3 Adaptive Total Order

One of the simplest ways of implementing total order is to elect a single sequencer, that is in charge of assigning a sequence number to each message transmitted in the group; then messages are delivered in the order specified by the sequencer. This algorithm is very effective when the following conditions hold: the network latency is small (given that messages sent by a node other than the sequencer are delayed by at least a roundtrip time) and the system load is low enough not to overload the sequencer. When these conditions are not met, it may be preferable to use other strategies to enforce total order. We will return to this topic later in the thesis.

2.3.4 Monolithic Solutions vs Modular Protocol Switching

As we have seen, there are several scenarios where one may want to change in runtime the behavior of a protocol. There are two main ways to achieve this goal.

One is to have a monolithic implementation of a protocol that implements all behaviors. In this case, the protocol must be able to adjust itself, commuting from one behavior to another. There are many examples of such protocols. For instance, the φ accrual failure detector (Hayashibara, Defago, Yared, & Katayama 2004) (which has the particularity of using as output a *suspicion level*), and an adaptable heartbeat failure detector (Bertier, Marin, & Sens 2002). Both algorithms adjust timeout values in runtime in an attempt to provide a better service. For total order, Rodrigues et al. (Rodrigues, Fonseca, & Veríssimo 1996) and Chockler et al. (Chockler, Huleihel, & Dolev 1998) also developed an adaptive protocol which is able to change the delivery order of messages in response to changes in transmission rates. However, using just one protocol to handle several conditions has some drawbacks. To start with, this approach is clearly not scalable, as the number of possible behaviors grows the protocol implementation becomes simply too large and complex. Also, it is very hard to add new behaviors without fully understanding the original code, and may require its restructuring.

The alternative to the monolithic approach is to have several individual implementations ready, and a *switching protocol* to switch among them. The modularity gained and the easy addition of new protocols is an important advantage of this approach. However, it requires switching protocols to be implemented. Both generic switching protocols (i.e., switching protocols that work for many services (Liu, van Renesse, Bickford, Kreitz, & Constable 2001)) and specialized switching protocols (for instance, a switching protocol that only works for switching among total order protocols (Mocito & Rodrigues 2006)) have been implemented. A cost analysis of these alternatives can be found in (Fonseca, Rosa, & Rodrigues 2009).

2.3.5 Protocol Composition Frameworks

A protocol composition and execution framework is a software package that supports the composition and execution of communication protocols. In terms of protocol design, the framework provides the tools that allow the application designer to compose stacks of protocols according to the application needs. In runtime, the framework supports the exchange of data and control information between layers and provides a number of auxiliary services such as timer management or memory management for message buffers. Several frameworks of this kind have been proposed, including the influential x-kernel (Hutchinson & Peterson 1991) (which inspired much of the subsequent work on this subject), Horus (van Renesse, Birman, & Maffeis 1996), Ensemble (Cadot, Kuijlman, Langendoen, van Reeuwijk, & Sips 2001), Cactus (Hiltunen, Schlichting, Ugarte, & Wong 2000), and Appia (Miranda, Pinto, & Rodrigues 2001). Typically, in these systems, protocols communicate by the exchange of events.

2.3.6 Requirements for Dynamic Adaptation

Most of the early protocol composition frameworks were concerned with providing the right tools to simplify the construction of configurable protocol stacks, i.e., by reducing the coupling among different protocols, such that the protocols could be configured in different ways, and the most appropriate stack could be used for each application scenario. Most recently, systems are also concerned with providing support for dynamic adaptation. We reproduce here a set of requirements, extracted from (Rosa, Rodrigues, & Lopes 2007a), that protocol composition frameworks need to satisfy to provide support for dynamic adaptation:

Requirement 1 the composition framework should support a programming model that makes easier for sources of context information to make it easily accessible.

Requirement 2 the composition framework should provide the mechanisms to support the capture of context information, both continuously or on-demand, as well as a mechanism to handle notifications generated by context sources.

Requirement 3 the composition framework should include, or be augmented with, services that are able to analyze the context information and report relevant changes.

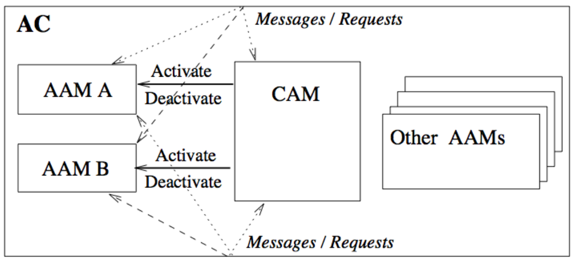

Figure 2.1: The structure of an Adaptive Component (AC) in *Cactus*.

Requirement 4 the composition framework has to provide support for dynamic reconfiguration, including mechanisms to perform parameter configuration, and mechanisms to perform the addition, removal, and exchange of services to a given composition.

Requirement 5 the composition framework should provide, either embedded in its kernel or as a set of additional services, a comprehensive set of mechanisms to support the coordination among nodes, to transfer service state information between services, and to enforce a quiescent state of a service.

Requirement 6 the composition framework should provide mechanisms to reason or obtain information on the system.

None of the frameworks we have cited previously fully satisfies this set of requirements (Rosa, Rodrigues, & Lopes 2007a), although all of them with the exception of *x-kernel* provide some mechanisms that favour dynamic adaptation. *Ensemble*, for example, uses virtual synchrony to support the installation of a new protocol configuration when a new view is installed (Birman, Constable, Hayden, Hickey, Kreitz, Van Renesse, Rodeh, & Vogels 1999). In the following paragraphs we provide some detail on the operation of *Cactus*, as an example of the architecture and services of this sort of frameworks.

2.3.7 Cactus Framework and Switching Protocol

The *Cactus* architecture consists of a number of system layers, which can be adaptive or non-adaptive (Chen, Hiltunen, & Schlichting 2001). The adaptive layers are constructed as a collection of *adaptive components* (ACs) which have both a *component adaptor module* (CAM) and *alternative adaptation-aware algorithm modules* (AAMs) (see Figure 2.1). Each AAM provides a different implementation of the component, while the CAM controls which implementation is better suited at a given moment. Cactus defines a *distributed adaptive component* (DAC) as a collection of adaptive components cooperating across different processes, which can provide message ordering or total order multicast, for example. Special care was taken to maximize the independence of the modules within an AC, so that new AAMs could be added without changes to existing code and that a CAM could control any AAM (this is achieved by each module specifying the set of operations it provides and details about adaptation steps).

Adaptation is performed in three phases: (i) change detection, (ii) agreement, and (iii) adaptive action. The first phase aims at detecting a change in the execution environment and asserting whether a modification to the current configuration is appropriate. The change detection can be done either in the CAM or the AAM modules. When a change is detected, *fitness functions* are used to determine the best implementation for the new scenario (one for each AAM). These functions map values reflecting the system state (such as network latency) to the suitability of the AAM, allowing the CAM to select the most appropriate implementations. The second phase consists of an agreement step between the various ACs (across different processes), using the system state perceived in each one of them to decide the actual global state. Finally, the last phase of the adaptation process consists of orchestrating the change of one AAM to another, while maintaining the correct behaviour of the service that the DAC implements.

Within the composition framework, ACs translate into *composite protocols* that contain both AAM and CAM *micro-protocols*. These micro-protocols are collections of *event handlers*, which are procedure-like segments of code that get executed when the corresponding event occurs. The way these handlers are set (issuing a *bind()* operation) favours the AAM transition, since new implementation modules are able to register the events they are interested in during runtime.

The *adaptive action* phase of *Cactus*, in which the actual protocol change happens, is an important topic by itself. One of the most important issues is that the nodes involved in the adaptation need to be coordinated, so that the provided service is maintained, and communication is not interrupted in a disruptive manner. Also, the adaptation should be as quick as possible, in order to minimize the time the service is not provided. Other issues of concern are, for example, to ensure that messages in transit from the previous configuration should still be processed, even if the destination is on a transition phase, and that no messages are ever dropped as a result of the adaptation.

To solve these issues Cactus proposes a multi-step *graceful adaptation protocol.* It consists of a *preparation* step, an *outgoing switchover* step, and an *incoming switchover* step. The first consists of preparing an adaptive component to receive either messages in the new protocol, or adaptation-related messages in the old one. Outgoing switchover switches the processing of outgoing messages to the new protocol. Finally, incoming switchover switches the delivery of incoming messages to the new protocol. Therefore, the old protocol stops processing outgoing messages in the second step, and stops processing and delivering incoming messages in the third. It is important to note that all components should complete the preparation step before the outgoing switchover, since they need to be prepared to interpret messages under the new protocol.

2.3.8 Other Switching Protocols

Another generic switching protocol has been described in (Liu, van Renesse, Bickford, Kreitz, & Constable 2001). It assumes there is a *manager* process which initiates the switch to the new protocol. To start the transition, this manager sends a PREPARE message to the other members. Upon reception of the PREPARE message, a member returns an OK message that includes the number of messages it has sent in the old protocol. From this moment on, new messages will be sent using the new protocol, and when received, they will be buffered instead of delivered. The manager then multicasts a SWITCH message informing all other nodes how many messages they should still receive in the old protocol. When all in transit messages have been received and delivered, members are free to switch to the new protocol entirely. The switching protocol does not deliver messages sent using the new configuration until all messages sent in the old configuration are delivered. Due to this, there may be a significant delay in the service during the transition.

Two adaptations of this generic protocol have also been proposed: one for switching causal ordering implementations and another for FIFO implementations (Fonseca, Rosa, & Rodrigues 2009). For causal ordering, the new protocol is also used for new messages as soon as the transition starts. However, it is important to guarantee that the causal relation of messages is still maintained during the switch. To do this, a vector clock (Raynal, Schiper, & Toueg 1991) is added to new messages. This allows that messages in the old protocol are delivered without restrictions, and messages in the new protocol are delivered according to their vector clock. Note that the transition is initiated when a message to do so is received, or when a message under the new protocol is received. This transition ends as soon as all members are using the new protocol. For FIFO ordering the key is to preserve the order between the last message of the old protocol, and the first of the new one. The proposed implementation consists in marking the last message sent under the old protocol when the transition is initiated. Recipients just buffer all messages of the new protocol until the marked message is received. At that point, all new messages can be delivered.

There is also a proposed solution to switch between total order multicast implementations (Mocito & Rodrigues 2006). This protocol has the advantage that the message flow is not stopped during the transition and works as follows. First, a control message is sent to all processes indicating the transition. When this message is received, the node starts broadcasting messages using both total order protocols, and the first message sent using the new protocol is marked. When nodes begin receiving messages in both protocols, messages from the old one are normally delivered, and from the new one are buffered in order. As soon as a marked message is received from all members, a “sanity” procedure takes place. First, all messages received under the new protocol that have not yet been delivered by the old one are delivered in order. After that, received messages under the old protocol are simply discarded, and all messages start being delivered under the new protocol.

2.3.9 When to Adapt

Besides the issue of *how* to switch implementations, discussed in the previous sections, there is also the problem of *when* it is time to do so, and *who* initiates it.

An adaptation process is usually triggered by a change detected in the environment at a given host (as happens in *Cactus*, for example). Although this host could simply decide by itself that a different implementation of the service would benefit the system, the detected change might be incorrect considering the state perceived at other members. So, using an agreement process is usually a better option. For this agreement to happen, a system-wide policy regarding which kinds of adaptations make sense should be in place at each member. These policies could be as simple as the deterministic fitness functions of *Cactus*, or more complex decision processes.

As opposed to a distributed approach, one can use a centralized alternative, consisting of having the reconfiguration process handled by an *adaptation manager* (Rosa, Rodrigues, & Lopes 2007b). This adaptation manager holds both a system-wide adaptation *policy* that should be applied, and a set of *reconfiguration strategies.* In order to decide if a reconfiguration is needed at a given time, the adaptation manager collects context information from the nodes present in the system, and proceeds to evaluate the policy. If a reconfiguration is required, an appropriate reconfiguration strategy to achieve it is selected, dictating how nodes should coordinate. This coordination is then achieved by having the adaptation manager issue commands to the nodes.

2.4 *RAppia*

RAppia (Rosa, Rodrigues, & Lopes 2007a) is a concrete example of a framework that provides both extensive configurability of protocols and services, and mechanisms to execute reconfiguration (in fact, its aim was exactly to address the adaptation requirements previously stated). It is an enhanced version of the *Appia* protocol composition framework. In *RAppia*, services are composed in a stack, aiming to provide a given Quality of Service (QoS). Usually, services that provide basic functionality sit at the bottom of the stack (such as point-to-point reliability), and higher level abstractions at the top (such as publish-subscribe or distributed databases). An instance of one of these compositions is called a service *channel*, and each layer of that service channel corresponds to a service *session*, which maintains its own state. These sessions interact through an event-driven model (similar to *Cactus*). Thus, a service implementation corresponds to a collection of event handlers. Events can be generated by services or come from other processes through the network. They also have two fundamental attributes: a *channel* and a *direction.* The first corresponds to the channel the event will flow in, and the second the direction taken in the stack, either up or down.

The *RAppia* adaptation support is built on three different aspects (Rosa, Rodrigues, & Lopes 2007a): the service programming model, adaptation-friendly services, and kernel mechanisms. The programming model employed, based on the exchange of events, allows for a group of default events that favour adaptability to be defined. These include events to access context information produced by services (*ContextQuery*, *ContextAnswer*, and *ContextNotification*), to place services in a quiescent/normal state (*MakeQuiescent* and *Resume*), and to handle state transfer (*GetState* and *SetState*), which are useful when switching service instances. An example usage of these events for replacing a service is depicted in Figure 2.2. In this case, the *MakeQuiescent* event

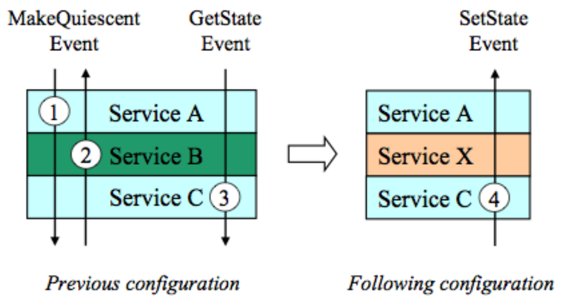

Figure 2.2: Service replacement with state transfer in *RAppia*.

is propagated down through the stack (1), and then reversed until the top (2), after which all of the services are in a quiescent state. After that, each service's state is stored through the *GetState* event (3), service B is replaced by X, and then the state is restored again through *SetState* (4). There is also an event that allows service parameters to be updated, such as the timeout value of a failure detector (*SetParameter*).

Other two useful properties enforced by the programming model are the use of protocol type hierarchies and message headers. Type hierarchies provide a way for protocols to be organized and tagged given their provided properties, and allow, for example, to reason about alternative implementations present in the system. By being included in the messages exchanged between peers, message headers also allow the exchange of control information. In *RAppia*, these headers are implemented as a *pool* from which services can retrieve or add fields.

Regarding adaptation-friendly services, *RAppia* includes two: a generic and configurable *context sensor* and a *reconfiguration monitor*. The context sensor is a service that handles the local capture of context information and allows for its dissemination. For dissemination, two main mechanisms are provided, one to query nodes about their state and another to send asynchronous notifications to other nodes (periodically, for example). The reconfiguration monitor defines a control channel through which it receives reconfiguration commands. These are: *Make-Quiescent*, which instructs the monitor to place a given service in a quiescent state; *Resume*, which resumes the normal activity of a service; *SetState* and *LoadState*, that allow the current state to be transfered between two different service implementations; and *Reconfigure*, which instructs the monitor to add, remove, or replace a given service.

With respect to kernel mechanisms, *RAppia* has two other unique features: the automatic buffering of events whose recipient is a service in a quiescent state, and the automatic update of event routes. The first allows services to handle events received in a quiescent as soon as it resumes, and it does not force the whole service channel to be put in a quiescent state. On the other hand, the update of event routes allows that events are only delivered to the interested services, including for those added in runtime. This allows for an optimized event flow.

2.5 Discussion

Among the several frameworks mentioned, only Cactus and Ensemble provide support for dynamic reconfiguration. However, in Cactus the adaptation process is fixed (using the graceful adaptation protocol described), and cannot be tailored to suit specific reconfiguration needs. For example, there is no support for adaptations that do not require coordination and affect only individual nodes. Also, the framework does not provide generic support for context information, and is restricted to the use of fitness functions to determine the best protocol implementation. Similarly to Cactus, Ensemble does not support generic context capture and monitoring, as well as reconfigurations that target specific nodes and not the whole system. This happens since new reconfigurations are distributed to all nodes in the same fashion as a view change for group communication. Another important drawback is that to reconfigure a protocol stack Ensemble requires all protocols to reach a quiescent state, which interrupts communication even when simple adaptations are executed, such as adding protocols that do not interfere with the message flow.

Summary

In this chapter we have surveyed work on group communication with emphasis on systems that already offer some form of support for dynamic adaptation. In this context we have also addressed the problem of providing an adequate infrastructural support to build this sort of adaptive communication stacks.

The next chapter will introduce a new iteration of the RAppia framework. This new system will attempt to improve on the current implementation, and support the set of features essential to the reconfiguration of protocol stacks, with emphasis on group communication and Appia's set of protocols that implement view synchrony.

3 RAppia 2.0

3.1 Introduction

This chapter describes the design and implementation of a new release of the RAppia system and the changes that were performed on the Appia group communication protocol stack to operate with the resulting new kernel. It starts by identifying the main concepts of the Appia framework, on which RAppia was based, and a brief description of the previous RAppia implementations. Then, the main components of RAppia 2.0 are described. After that we describe the main changes to existing protocols to support dynamic adaptation, with emphasis on Appia's original group communication stack. Finally, we present the platform's usage through some relevant examples.

3.2 Basic Appia Concepts

In this section we provide an overview on Appia's basic concepts, to allow the reader to better understand the aspects mentioned throughout the chapter.

Appia is a framework that allows the composition and execution of protocols, which communicate by the exchange of events. A *layer* is a representative of a protocol, and defines which events it generates and requires. A *session* is an instance of a protocol, and corresponds to its concrete implementation. Therefore, it maintains the state required to the protocol. A *QoS* is defined as a stack of layers, while a *channel* corresponds to a stack of sessions (a *QoS* instantiation).

As mentioned, these protocols interact by exchanging events. These events have two main properties: a *channel* and a *direction*. The first corresponds to the session stack, in which the event will flow, while the second is the direction the event is taking, either up or down. Sessions can produce their own events, specifying a given a direction, or *handle* incoming events produced by other sessions. When a session handles an event, it simply makes the required processing, and forwards the event to the next session. In particular, a session does not need to know which other implementations are present above and below.

An essential component to the Appia system is the *event scheduler*. This component is responsible for picking up events produced or forwarded by sessions, and delivering them to the next. Since sessions do not handle all types of events but only the ones defined in their corresponding layer, Appia provides a mechanism to implement this behaviour. When a channel is created, the *route* for each type of event is defined. This route specifies which sessions the event should be delivered to in the channel, and therefore handled by those sessions. This component has two main operations, *insert(event)* and *consume()*. *insert(event)* is called when an event is forwarded by a session, resulting in the event being enqueued for processing. *consume()* is called on Appia's single-threaded cycle, which instructs the scheduler to obtain an event and deliver it to the next session in its route.

Another feature of Appia is that sessions can be shared across channels, allowing protocols to correlate events circulating in those channels (Miranda, Pinto, & Rodrigues 2001). An example of when this is useful is for a protocol that synchronizes audio and video events sent from a single application, and allows different qualities of service for each type of information. An example of this setup is presented in Figure 3.1. On the left we have the desired setup for this scenario, on the right the corresponding Appia implementation, with two separate channels that differ only on the protocol used in audio and video. All the other sessions are shared between the two channels.

3.3 The RAppia Architecture

RAppia's architecture was originally proposed in (Rosa, Rodrigues, & Lopes 2007c), where a number of extensions to the Appia protocol composition framework are proposed, and later in (Rosa, Rodrigues, & Lopes 2007b), where the framework is extended with a set of pluggable components that provide runtime support for dynamic adaptation.

These papers provide the basis to support dynamic adaptation of protocol compositions through a policy-driven approach, according to the system model represented in Figure 3.2. The main components of this architecture are: a *context sensor*, which is present at each of

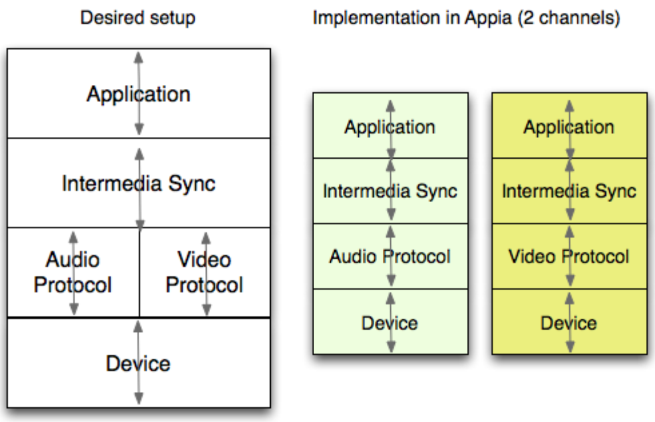

Figure 3.1: The implementation of an application that uses separate audio and video channels.

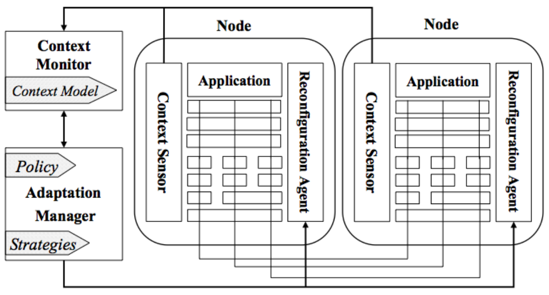

Figure 3.2: RAppia's System Model

the system's nodes and captures local context information; a *context monitor*, which gathers this context information from the sensors; the *adaptation manager*, which uses the obtained information and selects the most appropriate protocol composition to be used; and finally the *reconfiguration agent*, which performs the actual adaptations.

3.4 Previous Implementations of RAppia

The work described in this thesis builds upon two previous implementations of the RAppia system. The first was a proof-of-concept implementation of the RAppia's architecture, and the later an attempt to expand and complete the first implementation with more robust and flexible mechanisms. These implementations are briefly described in the next sections.

3.4.1 RAppia 0.5

The first set of RAppia features was introduced in (Rosa, Rodrigues, & Lopes 2007c). This report presented a set of add-ons to the original Appia framework. The first was the *Context Sensor* component, which defined a new set of Appia events, *GetValueRequest*, which is used to obtain information on a given variable, *GetValueReply*, which returns that information, and *TrapIndication*, which provides asynchronous notifications of context changes. The second component was a *Channel Reconfigurator*, which performed reconfiguration on channels by placing them in a quiescent state. This process was done through the use of *MakeQuiescent*, *GetState*, *SetState*, and *Resume* events. This version also had a *Buffering Layer*, whose aim was to hide reconfigurations from the application, by temporarily storing messages. Lastly, it had a specific protocol programming model, which defined the set of operations that a protocol should implement in order to be reconfigurable. These operations consisted basically of responding to the events defined by the *Channel Reconfigurator*.

Regarding kernel changes, this version introduced the use of a *header pool* model, in which protocols add and remove headers from messages instead of Appia's original push and pop implementation. It also introduced the dynamic update of event routes, in which the routes for a given event are updated after a channel reconfiguration.

This initial version also addressed the implementation of the Adaptation Manager component in (Rosa, Rodrigues, & Lopes 2007b) and (Rosa, Lopes, & Rodrigues 2006), which was mainly based on *adaptation policies* and *reconfiguration strategies*. These policies were composed of adaptation rules, inspired by Event-Condition-Action rules (McCarthy & Dayal 1989).

However, this version had a few missing features:

- It only supported the buffering of events between the application and remaining stack, and not in any session. This was due to the buffering layer being in a fixed position.
- Runtime channel creation was supported, but limited to the use of Appia's original mechanisms of XML specification. This method had the disadvantage that the adaptation manager was unable to know when the channel was ready for use.
- The Sensor component did not have specific operations to obtain the services present in a given channel, or the concrete implementation of a given service type.

- It had the event scheduler of the original Appia implementation, which was not designed with reconfiguration of services in mind.
- It was never applied to adaptations in the group communication stack.
- It had limited fault recovery.

3.4.2 RAppia 1.0

RAppia 1.0 was under development when the work reported in this thesis was initiated. It was an ongoing effort to add some missing features to RAppia 0.5. The work on RAppia 1.0 was being performed by volunteer students on a best-effort basis. As a result, it was incomplete and not well tested. Namely, the following limitations have been observed.

Regarding reconfiguration actions, it only supported the modification of service parameters, such as updating a timeout field to a new value. The runtime adaptation of protocol stacks was not implemented, and no support for placing sessions in a quiescent state existed. It also did not have generic support for obtaining context information, such as the value of a service parameter or the inspection of a stack composition.

This RAppia version also lacked support for the definition, evaluation and application of adaptation policies, an important tool to perform reconfigurations suiting a given system state.

The Adaptation Manager, a central component which gathers context information and applies the appropriate reconfigurations, did not have any kind of support for nodes to connect to it, and track if reconfiguration actions sent to the nodes were in fact applied.

Some tests performed on this platform also revealed some other problems: the event scheduler had a bug that could make it reorder the processing of events, leading to errors in protocols (namely, the group communication protocols), and the implementation of the header pool serialization was incorrect. The scheduler also did not support buffering for individual sessions, or handle adaptations in multiple channels.

This version already had, however, the main components present in the developed platform, namely the *Adaptation Manager*, and the *Sensor* and *Reconfigurator*, used to obtain context information and apply reconfiguration actions, respectively. It also had two interesting features, the graphical visualization of channel compositions, and a basic adaptation console, which allowed the user to manually send adaptation commands to the system's nodes.

3.5 RAppia 2.0

Given the limitations of the RAppia 1.0, a decision was made to produce a new release of the system, reimplementing some of its components from scratch. This section presents the design and implementation of RAppia 2.0. It starts by giving a brief overview of the main components that have been developed and their interaction. Then we provide a more detailled description of their implementation.

3.5.1 Overview

RAppia 2.0 system has two main elements focusing on dynamic adaptation: the *Adaptation Manager* and the *Reconfiguration Agent*; and two key core components, the *Interpreter*, and a new *Event Scheduler*.

The *Adaptation Manager* is a central component responsible of tracking the context information obtained from each of the nodes it controls, and reason on this information. If needed, and according to the policies defined by the programmer, it issues reconfiguration events to each of the system's nodes.

The *Reconfiguration Agent* is present on each system node and interprets the reconfiguration events received, as well as it answers context query events, for example requesting the value of a given service's parameter.

Finally, the *Interpreter* and *Event Scheduler* are two key core components regarding reconfiguration. The Interpreter focuses on applying the specific reconfiguration actions received, such as making changes to the channel. The new Event Scheduler, designed from scratch, has several characteristics that make possible the reconfiguration of protocol stacks, in particular, its organization of channels and a specific event processing order which we will later describe in detail.

3.5.2 Adaptation Manager

As stated, the *Adaptation Manager* is a central component present in the system, which performs two main functions: tracking context information obtained from its controlled nodes, and detect system states that demand adaptation, issuing the reconfiguration commands necessary.

It is composed of two main services, the *Manager*, and the *Context Monitor*, translated into two distinct Appia services.

3.5.2.1 Manager

The *Manager* service performs several functions. Firstly, it handles connection requests from nodes, and keeps track of these connections to send either context information requests or adaptation events. The Manager is also where the policies are defined and evaluated, with the evaluation happening when a ContextEvent is received form the Context Monitor. These policies have a Event-Condition-Action (McCarthy & Dayal 1989) structure. The event is a trigger event received from the Context Monitor, the condition could be for example *"currentTotalOrder is SequencerTotalOrder"*, and finally the action the corresponding Adaptation Event sent to the system nodes.

The Adaptation events, sent to the connected nodes, correspond to a single high level reconfiguration, such as *switch to sequencer total order*, and are composed by several finer grain actions. These actions can be, for example, *start buffering below service A*, *create temporary channel B*, *activate channel B on service A*, etc. We will provide a complete example, illustrating the use of these adaptation events and respective actions in Section 3.7.

Adaptation events can be broadcast (for example when a temporary channel is needed in every node), or sent point-to-point to a single node (for example for adding a flow control service to a specific node with high debit). When an adaptation event e_n is broadcast, the Manager has to wait for the confirmation of all the nodes before sending adaptation event e_{n+1}. Similarly, when an adaptation event is sent to a single node, it is that node that is required to confirm the adaptation before the Manager can proceed.

3.5.2.2 Context Monitor

The other service present in the Adaptation Manager is the Context Monitor. As its name suggests, the Context Monitor's main function is to obtain context information from the system's nodes, in order to gather a global view of the system state. In order to do this, the programmer specifies which information is required, and defines the period of refreshing it. The Context Monitor then proceeds to request the specified information from all the connected nodes periodically.

Regarding context queries, the Context Monitor allows to: i) obtain the value of a parameter present in a given service/channel ii) find the specific implementation of a given service type (for example find which specific total order protocol is present in a channel) iii) check if a given service type is present in a channel (for example, to check if a flow control service exists) and iv) obtain values for specific metrics tracked by the sensor service present in the nodes.

Periodically, the Context Monitor calculates system averages from the obtained values, for example message delivery latency, node throughput, node CPU usage, etc. These averages are weighted, giving a higher importance to more recent samples, allowing to establish a more correct view of the system status when policies are evaluated.

The Context Monitor also generates context events. These context events are sent to the Manager service in order to trigger a policy evaluation, and consist of simple threshold tests, for example, "*systemCapacity* $>$ *0.8*". Therefore, the programmer has to establish which triggers are relevant and could indicate that a reconfiguration is needed. These context events carry the latest system state stored in the Context Monitor, giving the policies access to the current system status.

3.5.3 Reconfiguration Agent

Present at each node there is a *Reconfiguration Agent*, which is composed of a *Reconfigurator* and a *Sensor*. Both these services are integrated into an existing stack, making use of Appia's feature of sharing a service among several channels. The services communicate with the Adaptation Manager through a dedicated channel, the *reconfiguration channel*, and use a dedicated TCP instance. The integration of these components into an existing stack is represented in Figure 3.3.

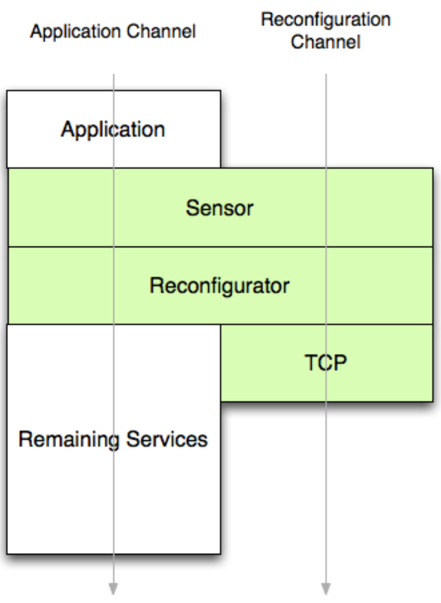

Figure 3.3: The integration of the Reconfigurator and Sensor components into an existing stack.

3.5.3.1 Reconfigurator

The Reconfigurator main tasks are i) translate adaptation events into their respective actions and ii) translate the actions into *Reconfiguration Action* channel events. These channel events are then processed by the *Interpreter* component, which performs the actual reconfiguration. The Reconfigurator is also responsible for sending confirmation events, *Adaptation Response Events*. These events carry, for each requested action, the indication if it was successful or not.

Added to that, this service also establishes the connection to the Adaptation Manager on startup.

3.5.3.2 Sensor

The *Sensor* service enables the capture of context information on a node. It is this service that receives the Context Query events sent by the Context Monitor and responds with the corresponding Context Replies. The way it gathers the various information is the following. Service parameters are obtained using Java's reflection mechanisms, by inspecting the requested parameter's name on a given service and channel. To find if a given service type is present in a channel, the Sensor has to analyze the service stack and inspect each session's type to find a match, returning either true or false in the reply. To find the implementation of a service type, it has to search the channel for the provided type, and return the concrete implementation of the service, such as “SequencerTotalOrder”. Regarding other collected statistics, the Sensor can: count the number of messages circulating in the channel it is integrated in, the latency of messages delivered locally (applicable with group communication messages), the CPU and memory usage, measurement of the RTT to other sensors in the system, and detection of false positives and node faults (by analysis of the circulating views). The *Sensor* was developed such that other information can be easily added.

3.5.4 Interpreter

As mentioned before, the Interpreter is RAppia’s component responsible of executing the reconfiguration actions. We now provide a description of each of the reconfiguration actions supported.

Add service the addition of a new service involves the instantiation of a layer and the corresponding session. If the session requires some of its parameters to be initialized, they should be obtained either from a session of the same type, or by implementing the *start(Channel)* method, which is executed on every new session introduced at runtime. For example, a total order service usually requires the current installed view, which can be transferred from an existing protocol in the channel. Upon the addition, the channel is updated, as well as its event routes.

Remove service the removal of a service also implies a channel and route update. In this scenario, new events will be delivered to sessions according to the new generated route. Existing events that have the removed service in their route are discarded. Therefore, the programmer has to enforce that events still circulating in the channel can actually be dropped, or make sure that no such events exist by using other reconfiguration actions.

Replace service at this time, replaced services should be compatible with the new service added, that is, share the same service type. This is necessary to guarantee that the state is correctly transferred between the two instances. Upon the replacement, the channel composition is updated, as well as its event routes.

Become quiescent quiescience is applied to a channel. The procedure is as follows. A *BecomeQuiescent* action is received by the Interpreter, specifying the affected channel. The Interpreter sends a *BecomeQuiescent* event down the channel. Each service establishes a quiescent state by coordinating with other nodes. After quiescience is reached, the service sends the event down. When all sessions are quiescent, the *BecomeQuiescent* event is handled by the channel's bottom which notifies the Interpreter that the process is complete.

Create channel in order to create a new channel, the programmer has to specify its identifier and initial QoS, in the form of a service list. There are two ways to indicate each service: i) specify a service class and an indication that it should be a new instance; ii) specify a service class and indicate that it should be an existing instance, by also identifying the channel the instance is present in. The new channel is then registered in the RAppia and started. Note that the confirmation that the channel was created is only sent after all services have been initialized for the new channel.

Remove channel removing a channel implies that all its pending events will be lost. New events that are set to circulate in this channel will be discarded by the event scheduler. The programmer should make sure that no events are circulating in the removed channel (for example by making is quiescent first).

Set value given a channel identifier, a concrete service, a parameter name, and a new value, the set value action simply consists of updating the parameter with the new value using reflection mechanisms.

Start/stop buffering the programmer indicates a channel identifier, a concrete service, and a *location*, which can be either *ABOVE* or *BELOW*. Since services have both an up and a down queue for the events being processed, this action consists of blocking the corresponding queue and not allowing its events to be processed until a stop buffering action is received. These actions are performed by the event scheduler, since it is the component that is responsible for delivering events to sessions.

Switch channel the switch channel action is performed on only one service, the total order switching service, and consists of starting the switching procedure by specifying the old and the new channels. This switching service corresponds to the algorithm described in Section 2.3.8.

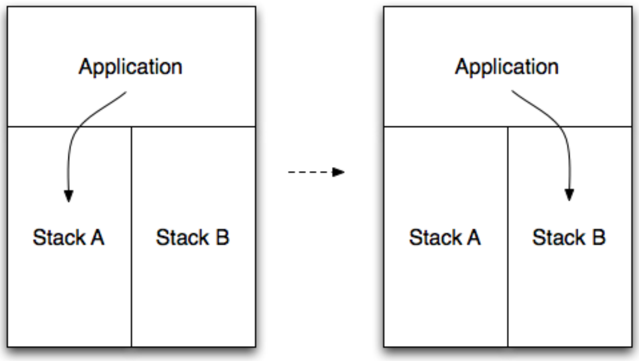

Figure 3.4: The execution of an *activate reconfiguration* action.

Pending reconfiguration this type of action signals a given service that it should expect a reconfiguration activation to a new channel. This type of action should precede an activate reconfiguration action.

Activate reconfiguration this reconfiguration signals a given service to set its active channel to the one specified. This means that all new events should be forwarded to the new channel. This type of action is used in services present in multiple channels. In Figure 3.4 an application is instructed to change its active channel, resulting in new events being forwarded to "Stack B" instead of "Stack A".

For asynchronous actions, which do not have an immediate response from the Interpreter and subsequently the Reconfigurator, there is a callback feature that services use to signal that the actions were performed successfully. This feature is used by the switch channel, become quiescent, and create channel reconfigurations. When these are completed, the sessions or components that end the procedure invoke specific methods on the Interpreter, which then confirms to the Adaptation Manager that the action was successful.

Note that since these actions are atomic, the programmer should take the necessary precautions of specifying all the steps required to make sure that no protocol is violated. For instance, before inserting a new service, there could be a need to use a start buffering action in the adjacent services.

3.5.5 Event Scheduler

We now give a few more details on RAppia 2.0's event scheduler, and the main differences from Appia's original implementation.

First of all, the services present in the scheduler are represented with up and down queues. These queues contain the events processed by their respective service. For example, if a service creates or forwards an event with a *down* direction, the event is placed on the service's down queue. This feature allows better control of events and their delivery to sessions, such as blocking up and down queues individually. A service representation in the scheduler is also unique for each service instance. If a service is present in multiple channels, the up and down queues are shared. In particular, when the scheduler controls multiple channels with shared sessions this is called a *channel composition*, which contains an ordered list of all the services.

Regarding event routes, Appia has an optimization that makes events bypass sessions that do not accept it. Now, events pass through all sessions present in their channel, in order to allow better control during adaptation. Although this results in a performance hit, it has an important advantage. Consider the following situation. Consider a channel with services A, B, C, D, and E (top to bottom). The up queue of service A is blocked, and there are events e_1, e_2, e_3 and e_4 present in the up queue of service E. Since events need to be processed in order, e_1 will be the first. If the next service that handles e_1 is service A, events e_2, e_3, and e_4 will have to wait until the queue is unblocked. If events go through all services, events e_2, e_3, and e_4 have the chance of being handled by intermediate services in the channel, allowing them to move up while service A is blocked.

Another important aspect of this new implementation is that channel compositions are *totally* ordered. That is, if channel c_1 has services A, B, C, and channel c_2 has services A, D, C, the scheduler will represent the composition either A, B, D, C or A, D, B, C (Figure 3.5). This happens in order to guarantee the causality in event processing when multiple channels are present. For example, consider the channels c_1(A, C) and c_2(A, B, C), and events e_1 (on channel c_2) and e_2 (on channel c_1) on the up queue of service C. e_1 will be processed first, since it is on the head of the queue, and be placed on the up queue of service B after being handled. When the scheduler proceeds to process event e_2, it will be handled by service A and placed on its up queue. Therefore, event e_1 will processed after e_2, which results in an order switch. Making events go through all services in the composition avoids this issue.

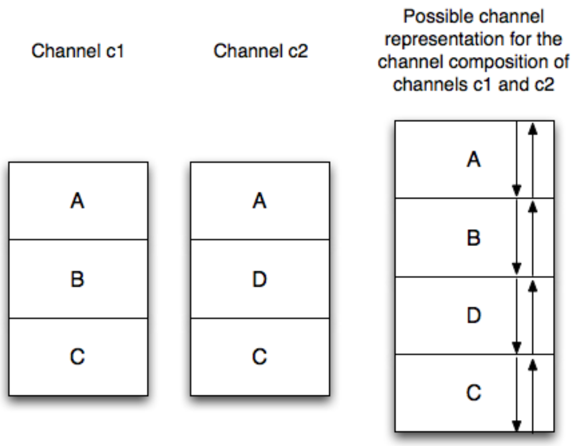
Channel c1
Channel c2
Possible channel representation for the channel composition of channels c1 and c2
A
B
C
A
D
C
A
B
D
C

Figure 3.5: The representation of a channel composition for two channels containing shared sessions.

For establishing a totally ordered composition from all the channels, we used a topological sort algorithm, by representing each channel as a directed graph. For example channel C_1(A, B, C) is represented as a graph with vertexes A, B, C and edges (A,B) and (B,C). This algorithm must be executed whenever there is a change in the composition, such as the addition of a new service or channel.

With these aspects in mind, event processing consists of going through the channel composition and finding pending events. When processing the up direction, the composition is inspected top to bottom, and the first session that has up events is the one processed. Inversely, when processing the down direction, the composition is inspected bottom to top, and the first session that has down events is processed. This, along with composition order, guarantees that event causality is preserved.

3.6 Refactoring Existing Protocols for Adaptation Support

Another difference between Appia and RAppia is that services should provide the implementation of a set of services if they are to be be used in reconfigurations. In particular they should respond to the following methods when needed:

start(Channel) start is executed when a service becomes part of a new channel or it is added to an existing channel.

stop(Channel) stop is executed whenever the service is no longer part of the channel specified. The service should make sure that no events are sent through this channel.

activateReconfiguration(Channel) this method corresponds to the *activate reconfiguration* action already described.

pendingReconfiguration(Channel) this method corresponds to the *pending reconfiguration* action already described.

becomeQuiescent(Channel) executing this method on a service should start a quiescence procedure, coordinated among all nodes that have the service running. It corresponds to the processing of a becomeQuiescent event.

resume(Channel) causes the service to resume normal operation on the specified channel, after being previously set to a quiescent state.

getState() obtains the transferable state of a given service.

setState(State) sets the state of a given service according to the *State* passed as argument.

getType() obtains the type of the service, such as *ApplicationType* or *TotalOrderType*.

The service should also set some extra information in their description (the *layer* in Appia terms). This information is: i) the service type (such as ApplicationType or TotalOrderType), ii) which parameters are adaptable (such as the name of a variable that defines a timeout time), and iii) the parameters that can be transferred to other services with the same type (such as views).

3.6.1 Appia Group Communication Protocols

Since the main test stack for RAppia was Appia's group communication protocols, they needed to be ported to the new platform, which involved some changes to their characteristics and implementation.

First of all, all protocols were changed to use the new header pool system, where headers are added and retrieved, and abandon the previous push and pop model. This new usage is as follows:

```
ev.getMessage().addHeader("TokenAgg-sequenceNumber", seq);
(...)
long seq = (Long) evx.getMessage().getHeader("TokenAgg-sequenceNumber");
```

Note that it is convenient for the programmer to use a protocol tag, such as "TokenAgg" in the above example, in order to avoid name collisions. If a collision does occur, the platform allows it but produces a warning message.

Protocols that were involved in reconfigurations had a few extra characteristics. For the failure detector protocols it was necessary to define which parameters were adaptable (in particular, the timeout parameter for the fixed-timeout detector), and the views transferred. For total order protocols state transfer of views was also added. An example of these additional characteristics for a total order protocol is presented next. The first piece of code specifies which parameters are transferable to other services of the same type, the second the type of this service.

```
transferableState = new Attribute[]{
  new Attribute("isBlocked",   Boolean.class),
  new Attribute("localState", LocalState.class),
  new Attribute("viewState",   ViewState.class),
};

serviceType = TotalOrderType.class;
```

The test application, also implemented as an RAppia service, needed to respond to the activate reconfiguration and pending reconfiguration actions, in order to support its usage in multiple channels during a reconfiguration. For example:

```
public void activateReconfiguration(Channel channel) {
  pendingChannels.remove(activeChannel);
  oldChannels.add(activeChannel);
  activeChannel = channel;
}

public void pendingReconfiguration(Channel channel) {
    pendingChannels.add(channel);
}
```

Added to that, all protocols that could be added in runtime or integrated in new channels needed to handle the *start(Channel)* method. This is the case for total order protocols, the switching and multiplexer protocols, and piggyback and flow control protocols.

In particular, executing *start(Channel)* on the switching and multiplexer protocols was particularly important, since this allowed to identify in runtime which channels are above and below these services. For example, for the switching protocol, and a channel c_1(Switching, B, C), executing *start(*c_1*)* enables the detection that the switching protocol is on the top of the channel, and therefore c_1 is one of the bottom channels.

The *becomeQuiescent(Channel)* method was also implemented for the test application and the sequencer and token-based total order protocols, since these were the ones required to be removed in runtime, in a scenario where total order implementations were changed.

Making the application protocol quiescent simply means that the application should no longer produce events on the specified channel. For total order protocols the process is more complex, since all nodes using the protocols need to be coordinated. One assumption is that no more events will be generated by the application, since the *BecomeQuiescent* channel event travels from the top to the bottom of the channel. The quiescence algorithms for the sequencer and token-based total order are described below.

3.6.1.1 Sequencer-based Total Order Quiescence

The quiescence procedure for the sequencer-based total order protocol is as follows.

Pre-conditions No more events will be generated by the application, and the only events that can be processed are: events that do not have an assigned order and are stored in a buffer, events transmitted by the sequencer that are in transit (which are not stored but delivered immediately), and events that specify the order of a stored event. No other events are procuded by the sequencer.

Procedure When a *BecomeQuiescent* event is received, the service enters a "quiescenceSetup" state. If the local buffer is empty, an *EmptyBuffer* event is sent immediately to the other system nodes. When receiving the *EmptyBuffer* event, nodes also enter the "quiescence-Setup" state, and proceed to check if their buffer is empty periodically. When the buffer is empty in all nodes the *BecomeQuiescent* event can be forwarded down.

3.6.1.2 Token-based Total Order Quiescence

The quiescence procedure for the token-based total order protocol is as follows.

Pre-conditions No more events will be generated by the application, and the only events that need to be delivered are the ones stored in local buffers. The only event circulating is the token.

Procedure When a *BecomeQuiescent* event is received, the service enters a "quiescenceSetup" state. When the token is received, the pending messages are delivered, and the token is rotated with an indication that the quiescence procedure should be started. When members receive the token and quiescence indication, they deliver their pending messages, set their state as "quiescent" and rotate the token. When the initial node receives the token again the process is complete and the *BecomeQuiescent* event can be forwarded down.

3.7 Platform Example

In this section we give a brief overview of how to use the platform, providing a complete example of a scenario where total order protocols are switched.

3.7.1 Gathering Context Information

In order to gather context information relevant to the scenario, we specify in the Context Monitor which parameters we would like to obtain periodically.

In this case we signal that we wish to obtain debit, throughput, CPU usage, memory usage, message latency, and a set of other parameters, in this case which is the concrete implementation of the TotalOrderType, and if the PiggybackType and FlowControlTypes are present in channel whose identifier is "channel A".

```
queryList.add(ContextQueryEvent.MESSAGESIN);
queryList.add(ContextQueryEvent.MESSAGESOUT);
queryList.add(ContextQueryEvent.CPU);
queryList.add(ContextQueryEvent.MEMORY);
queryList.add(ContextQueryEvent.LATENCY);

queryList.add(ContextQueryEvent.PARAMETER);
HashMap<String, Object> parameters = new HashMap<String,Object>();
parameters.put("channelIds",    new String[]{"channelA"});

queryList.add(ContextQueryEvent.SERVICE_QUERY);
parameters.put("serviceTypes", new Class []
    {TotalOrderType.class});

queryList.add(ContextQueryEvent.SERVICE_EXISTS);
parameters.put("services", new Class[]
    {PiggybackType.class, FlowControlType.class} );
```

3.7.2 Policy Definition and Application

For the adaptation procedure to take place, a trigger needs to be defined in the Context Monitor that starts the evaluation procedure. In the example, if the *currentSystemCapacity* value is below 0.59, a *ContextEvent* with the current system state is sent to the Adaptation Manager, which triggers the policy evaluation (example simplified):

```
if (systemCapacity < 0.59) {

  currentState = new State(
      avg_latency,
      capacity,
      avg_msgsInSecond,
      avg_load,
      currentServices);

  ContextEvent cEv =
    new ContextEvent("capacityMetric",
                       Status.below,
                       0.59,
                       currentSystemCapacity,
                       currentState);
  cEv.setDir(Direction.DOWN);
  cEv.setChannel(reconfigurationChannel);
  cEv.setSourceSession(this);
  cEv.init();
  cEv.go();
}
```

The policy itself contains sequences of condition-action pairs. For example, if the metric that triggered the evaluation was *capacityMetric* and the value was *below* the trigger threshold, the policy checks the protocol in use and capacity, and returns adaptations events with the actions to switch to the sequencer protocol. In this case, the value of the system capacity corresponds to a factor of utilization, which represents the percentage of resources used with the current load and protocol. If this value is low enough, a sequencer-based total order is selected, since it provides lower latency.

```
if (metricLabel.equals("capacityMetric") && (status == Status.below)) {
  if (state.isInUse("TokenTotalOrder")) {
    if (currentValue < 0.38) {
      resultingAdaptationEvent = switchToSequencerFromToken();
    }
  }
}
```

3.7.3 Switching Total Order Services

We now describe the process of switching between total order services in RAppia 2.0, as complete example of how to use the adaptation support the platform provides. Since this type of adaptation involves several steps, we will describe each of them, from a policy definition point of view.

The main objective is to not stop the communication flow during the adaptation. In order to do this, we opted for a process that instead of buffering or stopping the application from sending events, it makes it send messages through three different groups of channels: the first is the one that contains the current total order being used; the second is a channel that contains the total order switcher (briefly described in Section 2.3.8) and a multiplexer, as well as the two total order services simultaneously; and lastly the final channel containing the desired total order. During the switching procedure, where both total order protocols are present, the communication is not stopped since both channels are used simultaneously. These groups of channels are represented in Figure 3.6, with the shared sessions highlighted (adaptation components omitted for simplicity).

The multiplexer mentioned has two important advantages. The first is that the view-synchronous stack can be shared when both total order services are being used. The second is that events can be correctly forwarded to the correct channel when coming from the network. To do this, each event with the down direction that goes through the multiplexer gets an extra header indicating its channel. When the event comes with an up direction, the event is simply forwarded to channel present in that header.

Each of the following steps corresponds to a single *AdaptationEvent*, and therefore requires the confirmation of the nodes present in the system. The first parameter in the *AdaptationEvent* constructor specifies whether the event is to be broadcast to all the system's nodes or not. Note that most these steps do not affect the application, since they are performed in background.

```
new AdaptationEvent[] {
  new AdaptationEvent(true, createTemporaryChannels()),
  new AdaptationEvent(true, activateIntermediateConfig()),
  new AdaptationEvent(false, switchChannel()),
  new AdaptationEvent(true, activateFinalConfig()),
  new AdaptationEvent(false, becomeQuiescent()),
```

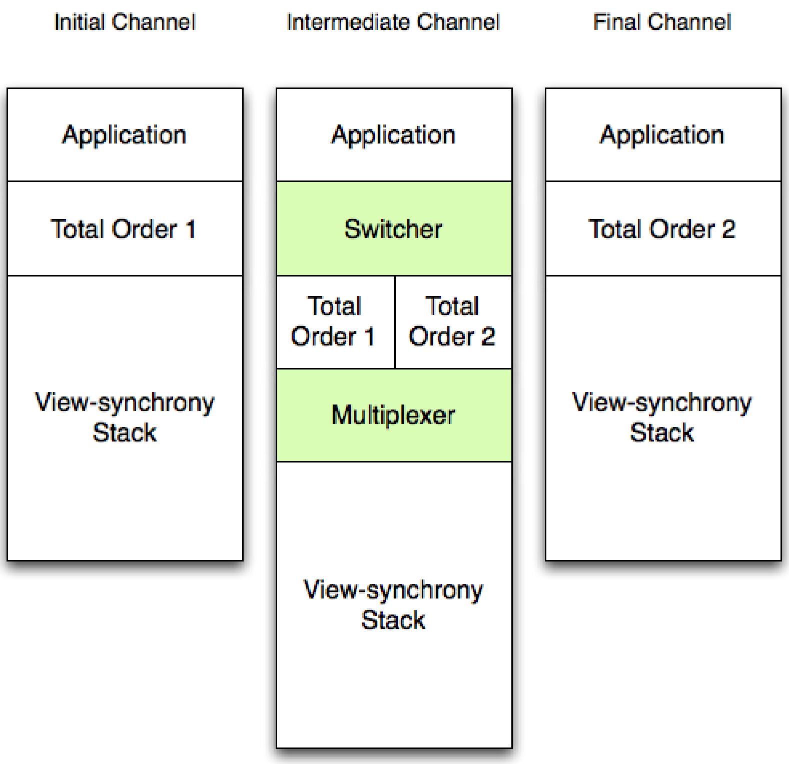

Figure 3.6: The stacks used in the reconfiguration of total order services, with the shared sessions highlighted.

```
  new AdaptationEvent(true, removeTemporaryChannels()),
}
```

Temporary channel creation The first step involves the creation of all auxiliary channels. The true/false parameter indicates whether the service should be a new instance or not. If the parameter is "false" (an existing instance), the programmer should specify the channel in which the service is present. If only only one instance exists, this additional parameter is not required.

```
  Action action1 = new Action(1,
    ActionType.CREATE_CHANNEL,
    "channel_B",
    Where.NO_SESSION,
    Where.NO_LOCATION,
    new Object[] {
      new Object[] {SwitchingLayer.class,        true},
      new Object[] {ReconfiguratorLayer.class, false},
      new Object[] {SensorLayer.class,           false},
      new Object[] {ApplicationType.class,       false},
  });

Action action2 = new Action(2,
  ActionType.CREATE_CHANNEL,
    "channel_C",
    Where.NO_SESSION,
    Where.NO_LOCATION,
    new Object[] {
      new Object[] {VSyncMultiplexerLayer.class, true},
      new Object[] {TotalSequencerLayer.class,     false},
      new Object[] {SwitchingLayer.class,          false},
    });

Action action3 = new Action(3,
  ActionType.CREATE_CHANNEL,
    "channel_D",
    Where.NO_SESSION,
    Where.NO_LOCATION,
    new Object[] {
      new Object[] {VSyncMultiplexerLayer.class, false},
      new Object[] {TotalTokenAggLayer.class,    true},
      new Object[] {SwitchingLayer.class,        false}
    });

Action action4 = new Action(4,
  ActionType.CREATE_CHANNEL,
  "channel_E",
  Where.NO_SESSION,
  Where.NO_LOCATION,
  new Object[] {
    new Object[] {TcpCompleteLayer.class, false, "channel_A"},
    new Object[] {GroupBottomLayer.class, false},
    new Object[] {GossipOutLayer.class, false},
    new Object[] {SuspectLayer.class, false},
    new Object[] {IntraLayer.class, false},
    new Object[] {InterLayer.class, false},
    new Object[] {HealLayer.class, false},
    new Object[] {StableLayer.class, false},
    new Object[] {LeaveLayer.class, false},
    new Object[] {VSyncMultiplexerLayer.class, false},
  });


Action action5 = new Action(5,
  ActionType.CREATE_CHANNEL,
  "channel_F",
  Where.NO_SESSION,
  Where.NO_LOCATION,
  new Object[] {
    new Object[] {TcpCompleteLayer.class, false, "channel_A"},
    new Object[] {GroupBottomLayer.class, false},
    new Object[] {GossipOutLayer.class, false},
    new Object[] {SuspectLayer.class, false},
    new Object[] {IntraLayer.class, false},
    new Object[] {InterLayer.class, false},
    new Object[] {HealLayer.class, false},
    new Object[] {StableLayer.class, false},
    new Object[] {LeaveLayer.class, false},
    new Object[] {TotalTokenAggLayer.class,   false},
    new Object[] {ReconfiguratorLayer.class, false},
    new Object[] {SensorLayer.class,          false},
    new Object[] {ApplicationType.class,      false},
  });
```

Activation of the intermediate channel Consists of sending a activateReconfiguration action to the application, specifying which channel it should use from now on. The channel corresponds to the intermediate channel containing the switcher, both total order protocols, and the multiplexer.

```
  Action action1 = new Action(1,
    ActionType.ACTIVATE_RECONFIG,
    "channel_B",
    ApplicationType.class,
    Where.NO_LOCATION,
    new Object[] { "channel_B" });
```

Start the switching procedure Trigger the switching procedure on the Switching service. In this case, switch from “channel C” to “channel D”.

```
  Action action1 = new Action(1,
    ActionType.SWITCH_CHANNEL,
    "channel B",
    SwitchingSession.class,
    Where.NO_LOCATION,
    new Object[]{"channel C","channel D"});
```

Activation of the final channel Set the application's active channel to "channel F", which contains the desired (final) total order protocol. Note that the switch is complete at this point, despite the further removal of temporary channels.

```
  Action action1 = new Action(1,
    ActionType.ACTIVATE_RECONFIG,
    "channel F",
    ApplicationType.class,
    Where.NO_LOCATION,
    new Object[] { "channel F" });
```

Set the old total order quiescent Make the channel that contains the previous total order algorithm quiescent. In this case, only the application, switcher, and first total order are made quiescent, since the view-synchronous stack will still continue to be used by the new total order.

```
  Action action1 = new Action(1,
    ActionType.BECOME_QUIESCENT,
    "channel B",
    ApplicationType.class,
    Where.NO_LOCATION,
    null);
```

Remove temporary channels Remove all channels that are no longer necessary to the normal operation of the new protocol. The sessions that are removed are placed in a quiescent state with the previous Adaptation Event. All channels are removed except "channel F", which contains the final total order protocol.

```
Action action1 = new Action(1,
    ActionType.REMOVE_CHANNEL,
    "channel_A",
    originalChannelId,
    Where.NO_SESSION,
    Where.FULLCHANNEL,
    null);

Action action2 = new Action(2,
    ActionType.REMOVE_CHANNEL,
    "channel_B",
    Where.NO_SESSION,
    Where.FULLCHANNEL,
    null);

Action action3 = new Action(3,
    ActionType.REMOVE_CHANNEL,
    "channel_C",
    Where.NO_SESSION,
    Where.FULLCHANNEL,
    null);

Action action4 = new Action(4,
    ActionType.REMOVE_CHANNEL,
    "channel_D",
    Where.NO_SESSION,
    Where.FULLCHANNEL,
    null);

Action action5 = new Action(5,
    ActionType.REMOVE_CHANNEL,
    "channel_E",
    Where.NO_SESSION,
    Where.FULLCHANNEL,
    null);
```

Summary

This chapter presented the elements of RAppia 2.0, a new implementation of the RAppia framework with a more complete set of features. The next chapter will provide an evaluation of this platform, in particular when compared to the original Appia framework.

4 Evaluation

4.1 Introduction

In this chapter we provide a set of tests aimed at evaluating and comparing the performance of the standard Appia framework with the adaptation-capable RAppia 2.0 framework. In particular, we perform a series of micro-benchmarks to analyse the impact of the adaptation components, and perform a real-world adaptation scenario, where total order implementations are switched. The structure of this chapter is as follows. Section 4.2 discusses the advantages and disadvantages of using the RAppia framework when compared to Apppia and other platforms. Section 4.3 describes the experimental settings of the tests and Section 4.4 the set of micro-benchmarks performed, as well as the performance of the adaptation procedure when compared to the type of adaptation provided by Appia.

4.2 Adaptation Discussion

Appia supports two very limited forms of runtime adaptation: i) switching between two individual protocol stacks (for example containing different total order implementations), with the use a special *switching service*, and ii) the adaptation of service parameters with the use of Java Management Extensions (JMX).

The support for stack switching requires that all the desired implementations are present in the system from startup, that is, all the individual channels and service instances must be decided before the adaptation actually occurs. This structure implies that all the processing and communication requirements of the protocols affects the system. For example, if the system is using a sequencer-based protocol as its current implementation, and has the option of switching to a token-based implementation, the token protocol functions normally albeit not processing application messages. Therefore, the normal behaviour of the token protocol, which implies the token rotation by sending group messages, is still present and has an impact on the system and therefore the sequencer-protocol. Another important disadvantage of this type of adaptation support is that if the system encounters a scenario which is not prepared to handle the adaptation is simply not possible. For example, consider a system that has only a stack with a sequencer-based total order protocol and a stack with a sequencer-based total order plus a piggybacking mechanism to aggregate messages. If the system evolves to a scenario where the number of messages being sent increases greatly, the system is unable to cope with these new circumstances, since for example a token-based protocol, which supports a much higher throughput, is not present. If on the other hand we have a library of available protocols, and these can be added and removed according to the system status without being present at startup, this provides a much higher level of flexibility, and does not impose an overhead on the protocol stack currently being used. Furthermore, Appia does not provide mechanisms to decide when to switch between implementations, so any type of policies or decision-related the programmer requires need to be defined in the protocols themselves or in extra helper layers, created from scratch.

Regarding the Java Management Extensions, Appia allows its use for obtaining and modifying service parameters. This however implies extra changes to the protocols, by implementing the appropriate JMX interfaces. Therefore, it does not serve as a generic mechanism to gather context information.

RAppia 2.0 provides the adaptation tools lacking in Appia, both for dynamic addition and removal of protocols, stack switching, policy definition and application, obtaining and reasoning on context information, and all the other features already described that provide enough flexibility to the system programmer.

4.3 Experimental Settings

The experiments were performed with a clusted of 5 nodes, each one equipped with an 8-core Intel Xeon E5506 CPU at 2.13GHz with 8 GB of RAM, running Ubuntu Server 10.04 with the 2.6.32-24 Linux kernel. All nodes were connected through a Gigabit Ethernet LAN.

The total order protocols used in the adaptation tests were a sequencer-based protocol, and a token-based protocol with the following implementations.

- *Sequencer-based* total order protocol. This protocol operates by electing a group member to play a special role in the algorithm, the *sequencer* role. The idea is that messages are multicast first by the sender and a sequencer number is assigned to each message by the sequencer, when it receives the message. Messages are delivered to the application in the order of the sequencer number.

- *Token-based* total order protocol. In this protocol a token is rotated among group members. Only the node that owns the token is allowed to send messages. The token ensures that there are no concurrent messages being sent and, therefore, defines a total order of message delivery. Note that the token protocol implicitly implements a piggyback layer, as multiple application messages that are requested to be sent while the node is waiting for the token can be aggregated by the node when it gets the token.

The switching protocol used was described briefly in Section 2.3.8, and has a more complete description in (Mocito & Rodrigues 2006). This protocol was already present in the Appia framework, and was ported to RAppia by implementing the appropriate methods and supporting more than two channels.

4.4 Adaptation Mechanisms Performance

In this section we present a set of benchmarks performed on both Appia and RAppia 2.0, in order to identify the overhead the adaptation mechanisms and components impose on the original platform.

4.4.1 Event Scheduling

The first test measured the time it takes a message to go through all the protocols implementing view-synchrony, with the intent of evaluating the overhead imposed by the new event scheduler. This test was made in a group of 5 nodes, with each node sending a fixed 60 messages per second. The times presented in Table 4.1 correspond to the difference between the instant the message left the application layer, and the instant the messages reach the bottom layer (in this case, the TCP layer).

Appia 4.1.0	RAppia 2.0
48.32 μs	188.23 μs

Table 4.1: Time messages take to go through all layers implementing view-synchrony.

4.4.2 Message Model

The second test measures the relative impact of the header pool model regarding message size, and also provides an overview on the size of messages carrying sample context information and adaptation events.

For the size comparison of messages circulating in Appia's view-synchronous stack and RAppia's implementation of that same stack, we sent sample messages from the application with no extra headers, and measured the final size of the message before sending it to the network. This results are presented in Table 4.2.

Appia 4.1.0	RAppia 2.0
88 bytes	492 bytes

Table 4.2: Message size in a view-synchronous stack for both Appia 4.1.0 and RAppia

In this case, RAppia always has an extra overhead over Appia. This is because each of the headers placed in the pool has an identifier, and this identifier is also carried in the message. For example, if a protocol places an integer in the pool, this integer occupies 4 bytes in Appia, but 4 bytes plus the number of characters in its label in RAppia. RAppia also implements this header pool with use of a Java HashMap object which has its own weight in the message size. It is also important to note that Appia's implementation of the push/pop header model optimizes most of the common data types, using the fact that headers are contiguous in memory and by employing low level operations. With the current implementation in RAppia, the optimization available is to tweak the value of the HashMap's initial size and the value of its expansion[1].

We also measured the size of each of the adaptation events sent from the Adaptation Manager to the nodes in the total order switching scenario, which was presented earlier in Section 3.7.3. This size was also measured on the instance before the message was sent to the

[1]See Java's HashMap documentation in http://java.sun.com/j2se/1.5.0/docs/api/java/util/HashMap.html

network. Results are shown in Table 4.2.

Adaptation event	Size
Temporary channel creation	2606 bytes
Activation of the intermediate channel	859 bytes
Start the switching procedure	803 bytes
Activation of the final channel	838 bytes
Set the old total order quiescent	695 bytes
Remove temporary channels	822 bytes

Table 4.3: Size of adaptation events sent from the Adaptation Manager to each of the system nodes, for a total order protocol switch.

Some of these operations are of course highly dependent on the actions specified. For example, an adaptation event for channel creation will necessarily have different sizes if the QoS of that channel has 2 or 20 layers. The same happens with the event removing the temporary channels. Other operations have a more fixed size. This is the case of channel activation, which is only dependent on the size of the protocol reference and the size of the new channel's identifier.

Regarding context information, we present the message size of a group of relevant context information, in order to give the reader an overview of the size requirements of these types of messages. The context information present in these messages is: CPU usage, memory usage, debit and throughput, observed message delivery latency, a service implementation query, a service existence query, an application parameter, and two parameters of the token implementation (time with the token, and time waiting for the token). The size of a query and corresponding response for this context information is presented in Table 4.4.

Message type	Size
Context query	1780 bytes
Context reply	1789 bytes

Table 4.4: Size of messages requesting and sending context information, between the Context Monitor and the Sensor components.

4.4.3 Adaptation Performance

In this section we compared the time it takes to switch total order implementations in the Appia and RAppia 2.0 platforms, for the sequencer-based and token-based protocols. The values consider the average time to switch between sequencer-based and token-based configurations. This total order switching scenario was tested with 3 and 5 nodes, with each node sending group messages at a constant rate of 60 messages per second. The switching algorithm was the same for both Appia and RAppia, differing only in its concrete implementation to match each platform's requirements.

For RAppia 2.0 we measured the time each of the adaptation steps requires, and the time spent in the whole adaptation procedure, from sending the first adaptation event to receiving the confirmation of the last. The adaptation time was measured for the switch from the sequencer-based total order to the token-based approach, and vice-versa, as well as the average time between these two values. Results for 3 and 5 nodes and presented in Table 4.5 and Table 4.6.

3 nodes			
Adaptation event	Average Time	Sequencer to Token	Token to Sequencer
Temporary channel creation	15.54 ms	16.28 ms	14.8 ms
Activation of the intermediate channel	3.24 ms	3.6 ms	2.88 ms
Start the switching procedure	39.05 ms	37.11 ms	41 ms
Activation of the final channel	3.28 ms	3.11 ms	3.46 ms
Set the old total order quiescent	29.07 ms	4.93 ms	53.22 ms
Remove temporary channels	4.82 ms	4.46 ms	5.17 ms
Total	95.48 ms	69.95 ms	121.02 ms

Table 4.5: Time spent for each adaptation event in a total order switch scenario, for 3 nodes.

5 nodes			
Adaptation event	Average Time	Sequencer to Token	Token to Sequencer
Temporary channel creation	17.85 ms	21.2 ms	14.49 ms
Activation of the intermediate channel	4.36 ms	5.14 ms	3.57 ms
Start the switching procedure	44.74 ms	46.15 ms	43.33 ms
Activation of the final channel	5.285 ms	6.15 ms	4.42 ms
Set the old total order quiescent	33.31 ms	5.81 ms	60.81 ms
Remove temporary channels	5.47 ms	5.51 ms	5.43 ms
Total	109.19 ms	88.63 ms	129.75 ms

Table 4.6: Time spent for each adaptation event in a total order switch scenario, for 5 nodes.

The first conclusion is that there is noticeable difference between switching to the sequencer-based and token-based approaches. This is mainly due to the quiescence procedure that is performed on these protocols. Since the token-based protocol requires one full turn of token after the procedure is started, this takes a longer time than the same operation for the sequencer. This results in the switch from the token to the sequencer taking a longer time.

Channel creation also has a relevant impact on the adaptation time. This is because the channel composition present in the event scheduler needs to be updated, in order to generate a new total order of all the protocols controlled. This corresponds to a topological sort algorithm.

For Appia we measured the time between triggering the adaptation in the local stack, and receiving the confirmation that the switch is complete. Results are shown in Table 4.7.

3 nodes	5 nodes
Complete adaptation	Complete adaptation
28 ms	34 ms

Table 4.7: Time spent in Appia for the adaptation process in a total order switch scenario, for 3 and 5 nodes.

This adaptation time consists of the time the switching algorithm requires, since there is no need for channel creation or removal (the protocols are already there). Also, there are no transmission delays for triggering the reconfiguration, since the adaptation is triggered locally. In particular, the adaptation process for Appia consists only of the *switchChannel* step and corresponding confirmation for the RAppia platform.

Summary

In this chapter we presented the evaluation of the RAppia 2.0 platform, focusing on its comparison to the original Appia implementation and scenarios with the view-synchronous protocol stack. We evaluated the impact of the new event scheduler and the use of the new header pool model, as well as sample sizes of adaptation and context information messages. Finally, we compared Appia and RAppia 2.0 with a real scenario, the switch of total order implementations.

The next chapter presents the final remarks regarding the RAppia 2.0 platform, as well as future work.

5 Conclusions

5.1 Conclusions

Adaptive protocol composition frameworks provide the programmer a suitable tool to design and implement systems that deal with dynamic environments. In order to cope with the variability of the environment, these frameworks often require a set of mechanisms that allow context monitoring, policy definition and evaluation, and adaptation of the protocol compositions. This thesis addressed the problem of building such frameworks and using them in the context of group communication.

To this end, we introduced several relevant aspects regarding both group communication and dynamic adaptation. We started by introducing the fundamental concepts behind a group communication system, including its common services and applications, and explained why it is suitable to have adaptation mechanisms in this kind of systems, and systems that deal with variable conditions in general. Next, the thesis provided an overview over the main frameworks that support runtime adaptation, discussing the key requirements addressed by each approach.

Based on the analysis of the related work, the thesis proposes a new implementation of the RAppia protocol execution and composition framework, that eliminates several shortcomings of the previous RAppia releases. Furthermore, the complete group communication stack of the Appia system was re-factored to operate on the new framework.

Finally, we provided an evaluation of RAppia compared to the original Appia system, focusing on the impact that the required adaptation components impose on event scheduling, message size for both the header pool model and adaptation events, and the adaptation process. The main conclusion is that although RAppia does not perform as well as Appia, the added flexibility in scenarios where conditions are dynamic greatly outweighs this aspect.

5.2 Future Work

Some aspects of the framework can be improved. For policy definition and evaluation, the main support relies on ECA policies. Although some initial tests have been conducted with a goal-oriented policy system, RAppia could provide core mechanisms that would support these as well. Policies could also be defined using a more portable specification, such as in XML. This would allow an easier switch between an existing set of developed policies.

Regarding the view-synchronous stack of Appia, not all protocols have the quiescence procedure implemented. Having full support for this would allow a coordinated switch of other protocols in this stack, such as the *stable* layer, which controls the reliable multicast property, the *inter* layer, which defines how concurrent views are decided using a consensus algorithm, among others.

Another aspect that should be addressed is fault-tolerance, namely what should be the behaviour of the system in presence of crashes. It is important to address these issues in order to guarantee that the system is not stopped due to nodes crashing during reconfiguration procedures, or nodes being unable to apply the reconfigurations instructed.

References

Amir, Y. & J. Stanton (1998). The spread wide area group communication system. Technical Report CNDS 98-4, The Center for Networking and Distributed Systems, John Hopkins University.

Bertier, M., O. Marin, & P. Sens (2002). Implementation and performance evaluation of an adaptable failure detector. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, DC, USA, pp. 354–363. IEEE Computer Society.

Birman, K. & T. Joseph (1987). Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev. 21*(5), 123–138.

Birman, K. P. (1993). The process group approach to reliable distributed computing. *Commun. ACM 36*(12), 37–53.

Birman, K. P., R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. Van Renesse, O. Rodeh, & W. Vogels (1999). The horus and ensemble projects: Accomplishments and limitations. Technical report, Ithaca, NY, USA.

Cachin, C. (2005). *Security and Fault-tolerance in Distributed Systems.* IBM Zurich Research Lab.

Cadot, S., F. Kuijlman, K. Langendoen, K. van Reeuwijk, & H. Sips (2001). Ensemble: A communication layer for embedded multi-processor systems. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, New York, NY, USA, pp. 56–63. ACM.

Chandra, T. D. & S. Toueg (1996). Unreliable failure detectors for reliable distributed systems. *J. ACM 43*(2), 225–267.

Chen, W., S. Toueg, & M. K. Aguilera (2002). On the quality of service of failure detectors. *IEEE Trans. Comput. 51*(1), 13–32.

Chen, W.-K., M. Hiltunen, & R. Schlichting (2001). Constructing adaptive software in distributed systems. In *21th International Conference on Distributed Computing Systems (21th ICDCS'01)*, Phoenix, AZ. IEEE.

Chockler, G. V., N. Huleihel, & D. Dolev (1998). An adaptive totally ordered multicast protocol that tolerates partitions. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, New York, NY, USA, pp. 237–246. ACM.

Chockler, G. V., I. Keidar, & R. Vitenberg (2001). Group communication specifications: a comprehensive study. *ACM Comput. Surv. 33*(4), 427–469.

Défago, X., A. Schiper, & P. Urbán (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv. 36*(4), 372–421.

Dolev, D. & D. Malki (1996). The transis approach to high availability cluster communication. *Commun. ACM 39*(4), 64–70.

Felber, P., X. Défago, R. Guerraoui, & P. Oser (1999). Failure detectors as first class objects. In *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*, Washington, DC, USA, pp. 132. IEEE Computer Society.

Fonseca, C., L. Rosa, & L. Rodrigues (2009). Custo da comutação dinâmica de protocolos de comunicação. In *Actas do primeiro Simpósio de Informática (Inforum)*, Lisboa, Portugal.

Hayashibara, N., A. Cherif, & T. Katayama (2002). Failure detectors for large-scale distributed systems. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, Washington, DC, USA, pp. 404. IEEE Computer Society.

Hayashibara, N., X. Defago, R. Yared, & T. Katayama (2004). The φ accrual failure detector. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, Washington, DC, USA, pp. 66–78. IEEE Computer Society.

Hiltunen, M. A. & R. D. Schlichting (1998). A configurable membership service. *IEEE Trans. Comput. 47*(5), 573–586.

Hiltunen, M. A. & R. D. Schlichting (2000). The cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Nürnberg, Germany.

Hiltunen, M. A., R. D. Schlichting, C. A. Ugarte, & G. T. Wong (2000). Survivability through customization and adaptability: the cactus approach. In *DARPA Information Survivability Conference and Exposition (DISCEX 2000), pp. 294–307.*

Hutchinson, N. C. & L. L. Peterson (1991). The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*(1), 64–76.

Jacobson, V. (1988). Congestion avoidance and control. In *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*, New York, NY, USA, pp. 314–329. ACM.

Khazan, R. I., A. Fekete, & N. A. Lynch (1998). Multicast group communication as a base for a load-balancing replicated data service. In *DISC '98: Proceedings of the 12th International Symposium on Distributed Computing*, London, UK, pp. 258–272. Springer-Verlag.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*(7), 558–565.

Liu, X., R. van Renesse, M. Bickford, C. Kreitz, & R. L. Constable (2001). Protocol switching: Exploiting meta-properties. In *ICDCS Workshops*, pp. 37–42.

Malloth, C. P., P. Felber, A. Schiper, & U. Wilhelm (1995). Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, Texas, USA.

McCarthy, D. & U. Dayal (1989). The architecture of an active database management system. *SIGMOD Rec. 18*(2), 215–224.

Miranda, H., A. Pinto, & L. Rodrigues (2001). Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, pp. 707–710. IEEE.

Mocito, J. & L. Rodrigues (2006). Run-time switching between total order algorithms. In *Proceedings of the Euro-Par 2006*, LNCS, Dresden, Germany, pp. 582–591. Springer-Verlag.

Moser, L. E., P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, & C. A. Lingley-Papadopoulos (1996). Totem: a fault-tolerant multicast group communication system. *Commun. ACM 39*(4), 54–63.

Pedone, F., R. Guerraoui, & A. Schiper (1998). Exploiting atomic broadcast in replicated databases. In *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, London, UK, pp. 513–520. Springer-Verlag.

Powell, D. (1996). Group communication (introduction to the special section). *Commun. ACM 39*(4), 50–53.

Raynal, M., A. Schiper, & S. Toueg (1991). The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett. 39*(6), 343–350.

Renesse, R. V., R. Minsky, & M. Hayden (1998). A gossip-style failure detection service. In *IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing Middleware'98, 15-18.*

Rhee, I., S. Y. Cheung, P. W. Hutto, A. T. Krantz, & V. S. Sunderam (1999). Group communication support for distributed collaboration systems. *Cluster Computing 2*(1), 3–16.

Rodrigues, L., H. Fonseca, & P. Veríssimo (1996, May). Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, pp. 503–510. IEEE.

Rosa, L., A. Lopes, & L. Rodrigues (2006, July). Policy-driven adaptation of protocol stacks. In *Proceedings of the Self-adaptability and self-management of context-aware systems workshop (SELF)*, Santa Clara (CA), USA, pp. (to appear). IEEE.

Rosa, L., L. Rodrigues, & A. Lopes (2007a). Building adaptive systems with service composition frameworks. In *Proceedings of the 9th International Symposium on Distributed Objects and Applications (DOA)*, Algarve, Portugal.

Rosa, L., L. Rodrigues, & A. Lopes (2007b). A framework to support multiple reconfiguration strategies. In *Autonomics '07: Proceedings of the 1st international conference on Autonomic computing and communication systems*, ICST, Brussels, Belgium, Belgium, pp. 1–10. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Rosa, L., L. Rodrigues, & A. Lopes (2007c). From appia to r-appia: Refactoring a protocol composition framework for dynamic reconfiguration. Technical Report TR-07-4, DI-FCUL.

Sousa, A., J. Pereira, F. Moura, & R. Oliveira (2002). Optimistic total order in wide area networks. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, Washington, DC, USA, pp. 190. IEEE Computer Society.

van Renesse, R., K. P. Birman, & S. Maffeis (1996). Horus: a flexible group communication system. *Commun. ACM 39*(4), 76–83.