Adaptive Group Communication

(extended abstract of the MSc dissertation)

Tiago José Pinto Taveira
Departamento de Engenharia Informática
Instituto Superior Técnico

Advisor: Professor Luís Rodrigues

***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.**

I. Introduction

Group communication denotes a set of membership, communication, and coordination services that support the development of distributed applications based on process groups [1]. It has been widely used to build multi-participant applications, such as collaborative applications [2], and fault-tolerant replicated services, for instance, database replication systems [3].

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) [4]. 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 [5], Ensemble [6], Cactus [7], Appia [8], 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.

II. Related Work

A. 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 [2]. 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 [3]. Finally, processes may coordinate to distribute tasks among them [9].

There are two primary services provided by a group communication system [10]: *membership* and *multicast communication*.

The purpose of the membership service is to provide to each participant up-to-date information about current members of the group [11]. 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.

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 [12]: (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.

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 [10]. 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 [13]. 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.

Group communication has been widely studied and many group communication systems have been implemented, including ISIS [14], Transis [15], Horus [5], Totem [16], Spread [17], Ensemble [18], Cactus [19], Phoenix [20], JGroups[1], Appia [8], 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).

B. Adaptation in Group Communication

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

1) 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 [22] 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 [23], [24].

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

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 [25] and gossip-style failure detection [26]. 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 [27].

2) 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, and therefore other protocols. This could be done by completely switching total order implementations, or by having a total order protocol that is able to adapt itself in a single, monolithic implementation.

C. Protocol Composition Frameworks

An important tool for providing services tailored for the execution environment is a protocol composition framework. 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 [28] (which inspired much of the subsequent work on this subject), Horus [5], Ensemble [18], Cactus [29], and Appia [8]. Typically, in these systems, protocols communicate by the exchange of events.

D. Adaptive Protocol Composition Frameworks

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 [30], that protocol composition frameworks need to satisfy to provide support for dynamic adaptation:

- the composition framework should support a programming model that makes easier for sources of context information to make it easily accessible.
- 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.
- the composition framework should include, or be augmented with, services that are able to analyze the context information and report relevant changes.
- 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.
- 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.
- 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 [30], 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 [6]. *Cactus* [29] employs an architecture of adaptive layers, which are constructed as a collection of *adaptive components* (ACs) which have both a *component adaptor module* (CAM) and *alternative adaptation-aware algorithm modules* (AAMs). The switch itself is handled by micro-protocols defined in these components.

E. Switching protocols

In order to switch protocols implementations, switching protocols are usually used in adaptation frameworks. Two examples are a generic switching protocol that has been described in [31] and a switching protocol specific to total order multicast [32].

The generic switcher 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.

The second example of a total order switcher has the advantage that the message flow is not stopped during the transition. The algorithm 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.

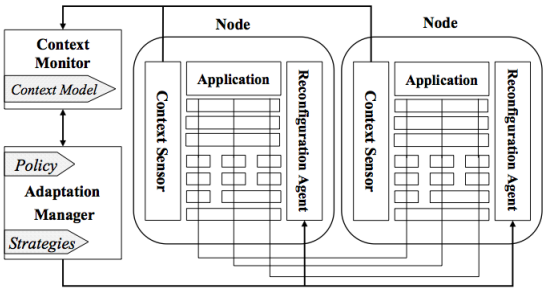

Figure 1. RAppia's System Model

III. RAppia 2.0

RAppia's architecture was originally proposed through a number of extensions to the Appia protocol composition framework, and later in [33], where the framework is extended with a set of pluggable components that provide runtime support for dynamic adaptation.

This initial work provided the basis to support dynamic adaptation of protocol compositions through a policy-driven approach, according to the system model represented in Figure 1. The main components of this architecture are: a *context sensor*, which is present at each of 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.

A. Previous RAppia implementations

The work described in this thesis builds upon two previous implementations of the RAppia system. The first was 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.

Both frameworks had limitations though. The first implementation, RAppia 0.5, 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. Also, 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 also had the event scheduler of the original Appia implementation, which was not designed with reconfiguration of services in mind. Furthermore, RAppia 0.5 was never applied to adaptations in the group communication stack.

RAppia 1.0, the second implementation, 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. For example, 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. As for context information, it did not have generic support for obtaining context information, such as the value of a service parameter or the inspection of a stack composition.

The limitations identified then led to the creation of a new implementation, RAppia 2.0.

B. Architecture Overview

The 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 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.

C. Supported Stack Operations

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 II-E.
- **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.

D. Event Scheduler

We now give a few more details on RAppia'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. 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.

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.

IV. Evaluation

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.

A. 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.

Appia 4.1.0	RAppia 2.0
48.32 μs	188.23 μs

Table I
TIME MESSAGES TAKE TO GO THROUGH ALL LAYERS IMPLEMENTING VIEW-SYNCHRONY.

The switching protocol used was described briefly in Section II-E, and has a more complete description in [32]. 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.

B. 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.

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 I 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).

2) 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 II and Table III.

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

3 nodes	
Adaptation event	Average Time
Temporary channel creation	15.54 ms
Activation of the intermediate channel	3.24 ms
Start the switching procedure	39.05 ms
Activation of the final channel	3.28 ms
Set the old total order quiescent	29.07 ms
Remove temporary channels	4.82 ms
Total	95.48 ms

Table II
TIME SPENT FOR EACH ADAPTATION EVENT IN A TOTAL ORDER SWITCH SCENARIO, FOR 3 NODES.

5 nodes	
Adaptation event	Average Time
Temporary channel creation	17.85 ms
Activation of the intermediate channel	4.36 ms
Start the switching procedure	44.74 ms
Activation of the final channel	5.285 ms
Set the old total order quiescent	33.31 ms
Remove temporary channels	5.47 ms
Total	109.19 ms

Table III
TIME SPENT FOR EACH ADAPTATION EVENT IN A TOTAL ORDER SWITCH SCENARIO, FOR 5 NODES.

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 IV.

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

Table IV
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.

V. 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.

ACKNOWLEDGMENTS

This work was partially supported by the Redico project and by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds. Parts of this work have been performed in collaboration with other members of the Distributed Systems Group at INESC-ID, namely, Liliana Rosa.

REFERENCES

[1] K. P. Birman, "The process group approach to reliable distributed computing," *Commun. ACM*, vol. 36, no. 12, pp. 37–53, 1993.

[2] I. Rhee, S. Y. Cheung, P. W. Hutto, A. T. Krantz, and V. S. Sunderam, "Group communication support for distributed collaboration systems," *Cluster Computing*, vol. 2, no. 1, pp. 3–16, 1999.

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

[4] D. Powell, "Group communication (introduction to the special section)," *Commun. ACM*, vol. 39, no. 4, pp. 50–53, 1996.

[5] R. van Renesse, K. P. Birman, and S. Maffeis, "Horus: a flexible group communication system," *Commun. ACM*, vol. 39, no. 4, pp. 76–83, 1996.

[6] K. P. Birman, R. Constable, M. Hayden, J. Hickey, C. Kreitz, R. Van Renesse, O. Rodeh, and W. Vogels, "The horus and ensemble projects: Accomplishments and limitations," Ithaca, NY, USA, Tech. Rep., 1999.

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

[8] H. Miranda, A. Pinto, and L. Rodrigues, "Appia, a flexible protocol kernel supporting multiple coordinated channels," in *Proceedings of the 21st International Conference on Distributed Computing Systems*. Phoenix, Arizona: IEEE, 2001, pp. 707–710.

[9] R. I. Khazan, A. Fekete, and N. A. Lynch, "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: Springer-Verlag, 1998, pp. 258–272.

[10] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Comput. Surv.*, vol. 33, no. 4, pp. 427–469, 2001.

[11] M. A. Hiltunen and R. D. Schlichting, "A configurable membership service," *IEEE Trans. Comput.*, vol. 47, no. 5, pp. 573–586, 1998.

[12] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[14] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," *SIGOPS Oper. Syst. Rev.*, vol. 21, no. 5, pp. 123–138, 1987.

[15] D. Dolev and D. Malki, "The transis approach to high availability cluster communication," *Commun. ACM*, vol. 39, no. 4, pp. 64–70, 1996.

[16] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos, "Totem: a fault-tolerant multicast group communication system," *Commun. ACM*, vol. 39, no. 4, pp. 54–63, 1996.

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

[18] S. Cadot, F. Kuijlman, K. Langendoen, K. van Reeuwijk, and H. Sips, "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: ACM, 2001, pp. 56–63.

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

[20] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm, "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, 1995.

[21] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[22] V. Jacobson, "Congestion avoidance and control," in *SIGCOMM '88: Symposium proceedings on Communications architectures and protocols*. New York, NY, USA: ACM, 1988, pp. 314–329.

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

[24] M. Bertier, O. Marin, and P. Sens, "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: IEEE Computer Society, 2002, pp. 354–363.

[25] P. Felber, X. Défago, R. Guerraoui, and P. Oser, "Failure detectors as first class objects," in *DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications*. Washington, DC, USA: IEEE Computer Society, 1999, p. 132.

[26] R. V. Renesse, R. Minsky, and M. Hayden, "A gossip-style failure detection service," in *IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing Middleware'98, 15-18*, 1998.

[27] N. Hayashibara, A. Cherif, and T. Katayama, "Failure detectors for large-scale distributed systems," in *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2002, p. 404.

[28] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

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

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

[31] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. L. Constable, "Protocol switching: Exploiting meta-properties." in *ICDCS Workshops*, 2001, pp. 37–42.

[32] J. Mocito and L. Rodrigues, "Run-time switching between total order algorithms," in *Proceedings of the Euro-Par 2006*, ser. LNCS. Dresden, Germany: Springer-Verlag, 2006, pp. 582–591.

[33] L. Rosa, L. Rodrigues, and A. Lopes, "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: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, pp. 1–10.