Adaptive Group Communication

Tiago José Pinto Taveira
`tiagotaveira@ist.utl.pt`

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 report studies the techniques that allow the construction of an adaptive group communication stack, which is able to reconfigure itself dynamically in response to changes in the operational envelope, in order to optimize its performance under variable conditions.

1 Introduction

Group communication denotes a set of membership, communication, and coordination services that support the development of distributed applications based on process groups [4]. It has been widely used to build multi-participant applications, such as collaborative applications [41], and fault-tolerant replicated services, for instance, database replication systems [35].

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) [37]. 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 [40], Ensemble [5], Cactus [9], Appia [32], among others.

The performance of these 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 this service identified about sixty protocols [13]. 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.

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.

This report addresses the problem of building such adaptive group communication services. It departs from a brief overview of group communication and then provides several examples of why adaptation may help in improving the performance of these systems. In order to be able to perform dynamic adaptation it is required that the communication protocols are implemented in a protocol composition and execution framework that facilitates the runtime reconfiguration. The report also surveys such frameworks.

Finally, the report discusses how an existing group communication stack can be augmented to support dynamic adaptation. More precisely, it addresses how the group communication stack of the *Appia* [32] system can be migrated to *RAppia* [43], a version of the same framework with explicit support for dynamic adaptation, and identifies a number of services that can be used to illustrate the benefits of dynamic adaptation in this context.

The rest of this report is organized as follows. Section 2 identifies the goals of this work. Section 3 describes the related work relevant to adaptive group communication. Section 4 proposes an architecture for an adaptive group communication system, and Section 5 states how the resulting system will be evaluated. Finally, Section 6 includes a schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of building an adaptive group communication service. More precisely:

> *Goals:* This work aims at analyzing, implementing, and evaluating an adaptive version of the *Appia* group communication system.

To achieve this goal, we will start by porting the current *Appia* group communication stack to the *RAppia* framework. Subsequently, we will develop reconfigurable versions of some of the existing layers and define the policies that control the adaptation of these layers.

> *Expected results:* This work will provide: (i) a prototype implementation of an adaptable group communication stack for the *RAppia* system; and (ii) provide an evaluation of this stack in face of dynamic workloads and network conditions, using the ModelNet[1] framework.

[1] https://modelnet.sysnet.ucsd.edu/

3 Related Work

This section 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 *RAppia*, a protocol composition and execution framework designed specifically to support dynamic adaptation.

3.1 Group Communication Concepts

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 [41]. 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 [35]. Finally, processes may coordinate to distribute tasks among them [28].

There are two primary services provided by a group communication system [12]: *membership* and *multicast communication.*

3.1.1 Membership Service

The purpose of the membership service is to provide to each participant up-to-date information about current members of the group [24]. 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 [6].

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 [3] and Phoenix [31]. Partitionable membership was first introduced as part of Transis [15], and is also present in Totem [34], Horus [40], and Appia [32], 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 [8], *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 [10] 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.

3.1.2 Reliable Multicast Service

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 [8]: (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 [6]:

– *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 [12]. 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 [29]. 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 [13]:

- *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 [13]. Given that not all applications require the more expensive uniform version, both variants have been implemented in existing systems (such as *Horus* or *Appia*).

3.1.3 Examples of Group Communication Systems

Group communication has been widely studied and many group communication systems have been implemented, including ISIS [3], Transis [15], Horus [40], Totem [34], Spread [1], Ensemble [7], Cactus [23], Phoenix [31], JGroups[2], Appia [32], 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] http://www.jgroups.org/

3.1.4 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 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 [13], token based (part of the *privilege-based* class of algorithms [13]) and a statistically estimated total order (SETO) [46]. The failure detector currently has two implementations, the φ accrual failure detector [22] and a heartbeat failure detector.

3.2 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 [13]. 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.

3.2.1 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 [30].

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

3.2.2 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 [27] 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 [2, 22].

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 [17] and gossip-style failure detection [39]. 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 [21].

3.2.3 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 report.

3.2.4 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 [22] (which has the particularity of using as output a *suspicion level*), and an adaptable heartbeat failure detector [2]. Both algorithms adjust timeout values in runtime in an attempt to provide a better service. For total order, Rodrigues et al. [42] and Chockler et al. [11] 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 [30]) and specialized switching protocols (for instance, a switching protocol that only works for switching among total order protocols [33]) have been implemented. A cost analysis of these alternatives can be found in [20].

3.3 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 [26] (which inspired much of the subsequent work on this subject), Horus [40], Ensemble [7], Cactus [25], and Appia [32]. Typically, in these systems, protocols communicate by the exchange of events.

3.3.1 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 [43], 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.

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 [43], 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 [5]. 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.

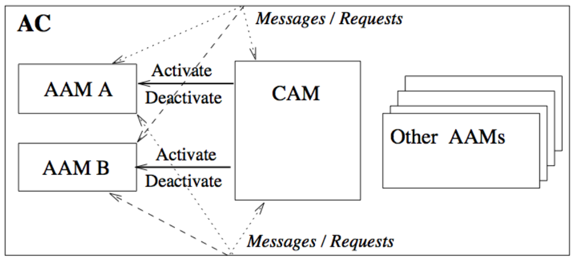

Fig. 1. The structure of an Adaptive Component (AC) in *Cactus*. It is composed of a component adaptor module (CAM) and alternative adaptation-aware algorithm modules (AAMs).

3.3.2 Cactus Framework and Switching Protocol

The *Cactus* architecture consists of a number of system layers, which can be adaptive or non-adaptive [9]. 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 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.

3.3.3 Other Switching Protocols

Another generic switching protocol has been described in [30]. 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 [20]. 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 [38] 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 [33]. 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.

3.3.4 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* [44]. 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.

3.4 *RAppia*

RAppia [43] 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 [43]: 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. In this case, the

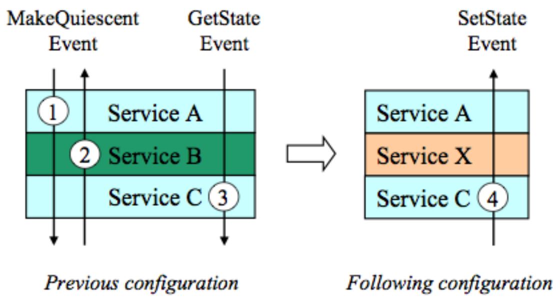

Fig. 2. Service replacement with state transfer in *RAppia*.

MakeQuiescent event 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: *MakeQuiescent*, 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.

4 Proposed Architecture

In order to achieve our goals, we will use the configurable group communication stack that has been developed for the *Appia* system and augment it to support runtime adaptation. For this purpose, we will first need to port the existing protocols to the *RAppia* system, as we will leverage of the adaptive mechanisms of *RAppia* to implement our solutions. Furthermore, we will use a policy based adaptation strategy. Finally, we will design, implement, and evaluate adaptive versions of some of the services of the group communication stack. These aspects are detailed in the following paragraphs.

4.1 Porting the Group Communication Stack to *RAppia*

As mentioned earlier, *RAppia* includes several mechanisms that support the dynamic adaptation of a communication stack. Therefore, our solution will be based on this framework.

The group communication stack that is currently available in *Appia* needs to be ported in order to run in the *RAppia* framework. For instance, the protocols need to be changed to support the insertion and removal of message headers introduced by *RAppia*, to be augmented with handlers for reconfiguration events, such as *MakeQuiescent* and *Resume*, and for obtaining context information, such as *ContextQuery* and *ContextAnswer*. The first task of our work will be to perform and validate these changes.

4.2 Policy-Driven Adaptation

We will use a policy-driven approach to control the adaptation of the group communication stack. There will be a centralized component that will be responsible for collecting context information and feed this information to an engine. This engine will then evaluate a policy that defines which adaptations need to be performed. The policy should be provided during the application's deployment, and defined as an independent module so that it can be easily changed. This can be achieved by implementing the policy as a separate service in the composition. The policy engine will not be the focus of the work. Instead, results from other members of the research group will be used [44].

The work to be performed for each adaptive protocol to be included in the group communication stack consists of identifying: (i) the relevant context to trigger the adaptation of the component; (ii) the relevant policies for that component; (iii) how can the runtime adaptation be performed, in particular, what is the required coordination among nodes involved in the adaptation, and which switching protocols should be used (if any).

4.3 Adaptive Protocols

In this work we plan to design and implement three different adaptive components of the group communication stack, namely: the failure detector, the protocols implementing reliable multicast (both regular and uniform), and total order multicast. The goal is to establish the needed reconfiguration mechanisms in *RAppia* for each of these components, in order to improve their performance in certain scenarios. In the following sections we will describe these aspects for each of the reconfigurable components.

4.3.1 Failure Detector

Appia includes two implementations of failure detection, the φ accrual failure detector [22] and a fixed-pace heartbeat-based failure detector. The goal is to add a third, with a different communication pattern. A possibility is to use one with an hierarchical structure [21], or a *lazy failure detector*, such as the one described in [18]. The latter attempts to aggregate control messages with regular application messages in order to minimize traffic. The heartbeat failure detector will also be improved to support the runtime adaptation of the heartbeat period.

When to switch One of the most relevant indicators for an implementation switch of the failure detector is the current network latency. Given a higher latency, the timeouts should also be adapted to higher values. Another indicator is the number of messages being exchanged, as an example of network load. If this load is higher, it makes sense to use an implementation that is more conservative regarding the number of used control messages.

Capturing context One way to measure the latency to other members from a given node is to periodically measure the round trip time between them. This can be achieved by using explicit control messages, such as echo requests, and measure the time they take. As for the network load, one can count the number of messages being sent and received from the group, either on the service implementations that use this indicator or as an independent service. Since *Appia* already includes a TCP layer that measures the number of messages being exchanged, this will probably be used. Both the latency and number of exchanged messages with the group can be obtained locally on a single peer.

How to switch Switching failure detectors will involve little or no coordination with other group members, since multiple implementations can probably coexist in most cases. Furthermore, since failure detection has little interference in the normal communication flow, it can happen without stopping other components. Despite this, the switching should avoid the increase in the detection latency.

4.3.2 Reliable Multicast

Another component that we plan to re-implement to support dynamic reconfiguration is the reliable multicast layer. In *Appia*, reliable (regular) multicast is implemented by the *stable* protocol. In each process, this protocol maintains information about which messages are delivered in other peers. When processes detect that they are missing a given message, they request it to a member that possesses that message. Similarly, the uniform multicast implementation also maintains information about delivered messages on other peers, but in this case to aid the decision of which messages can be delivered. This control information is propagated either by piggybacking control data to normal application messages, or by sending explicit control messages periodically.

This component can be optimized in two ways. One is to dynamically adjust the timer that controls the periodic information exchange, making this exchange happen quicker or slower. Another is to implement alternative protocols, for example protocols that use explicit acknowledgments for each received message.

When to switch For this component, the main aspect is the tradeoff between how many control messages are exchanged and how fast one recovers from the loss of a data message. So, if the network load is light, one can use more control messages and, therefore, propagate information quicker, allowing the system to reach a stable state faster. Inversely, when the network load becomes higher, we should switch back to a more conservative approach.

Capturing context As seen for the failure detector, one of the ways to obtain information about the network load is by counting the number of messages being exchanged. In this case we can also count them explicitly, or try to re-use the structures about delivered messages on other peers and infer load information. This could also be done by adding more control data to the one already present, such as message delivery timestamps (to retrieve information such as messages per second). As previously mentioned, *Appia*'s TCP layer implementation may also be a suitable option to obtain this kind of information.

How to switch To modify the period for the exchange of control information, there is obviously no need to change the protocol implementation; this value can be changed using the *SetParameter* event. If it is necessary to change the protocol implementation, it will be mandatory to ensure that the guarantees of reliable delivery are preserved during the switch. This may require the layer to be put in a quiescent state before the protocol switching occurs.

4.3.3 Total Order Multicast

For the total order layer three different protocols will be used and runtime switching among these will be supported. The protocols will be the following:

– A *sequencer-based* total order protocol. This protocol has been briefly described in Section 3.

- A *token-based* total order protocol [13]. The algorithm consists in having a token message that carries a sequence number circulating among the group members. When a process wants to broadcast a message, it must wait for the token to arrive. After that, he sets one sequence number for each of its messages and sends them to the group. When finished, the token is updated and sent to the next process.
- A *symmetric* total order protocol [14, 36]. In the symmetric approach, ordering is established by all processes in a decentralized way, using logical clocks [29] or vector clocks [19]. Processes use these mechanisms to deliver messages according to their partial order, and concurrent messages are totally ordered using a deterministic algorithm [33].

Each protocol will be implemented by a different, independent, *RAppia* component. Therefore, all group members will have to use the same protocol, which means that switching between any two of these protocols needs to be coordinated.

When to switch The *sequencer* approach is advantageous for when only a single member of the group, the assigned sequencer, is sending messages. This happens since the sequence number added to messages is generated locally and does not involve an extra control message sent to another peer. The *token-based* implementation is appropriate when there is a high debit of messages from all group members, achieving the best throughput with both uniform and regular guarantees (although at the cost of added latency) [16]. Finally, the *symmetric* protocol performs better in scenarios where every member is sending messages at the same pace and the network latency is large [33].

Capturing context The main context information to be captured is the rate of messages being sent on each process (messages per second) and the network latency. Using events like *RAppia*'s *ContextQuery* and *ContextAnswer*, we can periodically request the number of messages other group members are sending, and assert whether a reconfiguration is suitable for the current system status.

How to switch In Section 3.3, some switching protocols that could be used to commute between total order implementations have been already presented. Possibilities include the use of the algorithm described in [33], which uses both the old and the new protocols simultaneously in order to reduce the impact on the message flow; a generic switching protocol [30], which requires messages to be buffered during reconfiguration; and a protocol replacement algorithm [45], which imposes a communication delay. In all these approaches, the required coordination between nodes during the transition is already implemented.

5 Evaluation Methodology

The tool that will be used to evaluate the proposed system is the ModelNet network emulator. ModelNet allows to emulate multiple network topologies and their respective link characteristics, including bandwidth, queueing, propagation delay, and drop rate. In particular, these properties can be adjusted dynamically, a specially useful feature for testing adaptive systems. Another important aspect of ModelNet is that it supports the emulation of unmodified applications, which will simplify the system testing.

For the failure detector component, the performance baseline to be considered is the current heartbeat protocol, since it is the simplest. The comparison metrics relevant in this case are: (i) how are the accuracy and completeness properties improved and (ii) the number of partitioned views due to detected faults.

Regarding reliable multicast, the baseline will also be the current implementations present in *Appia*. Performance improvements to this component will consider: (i) the required time for the system to reach a stable state (either regarding uniformity or regular reliability), and (ii) the number of extra control messages used (which imposes extra bandwidth requirements).

For total order, several scenarios will have to be tested, since *Appia* currently includes several implementations and only one can be configured to be used at deployment. Possibilities include comparing each of the included protocols to the proposed reconfigurable architecture for this component. The main metric to be considered is the overall system throughput of messages sent to the group.

Regarding policies, other good indicators of the system performance could be: (i) the time it takes for components to recognize a change to the current configuration is appropriate, and (ii) the time between the detection of this change and the completion of the reconfiguration process.

Another aspect to be measured in all components is the imposed overhead of switching between different implementations, both when this switch involves coordination with other group members and when no such coordination is needed.

6 Schedule

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 29 - May 2: Perform the complete experimental evaluation of the results.
- May 3- May 23, 2009: Write a paper describing the project.
- May 24 - June 13: Finish the writing of the dissertation.
- June 14, 2009: Deliver the MSc dissertation.

7 Conclusions

The goal of this project is to implement and evaluate an adaptive group communication protocol stack. In this report 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.

As a result, we propose an architecture that is based on augmenting the group communication stack of the *Appia* system. To support our work we leverage on the mechanisms provided by the *RAppia* protocol composition framework. Furthermore, we propose to use a policy-driven adaptation strategy. We have identified three layers of this stack that can benefit from dynamic adaptation. For each of them we identified when a specific implementation is appropriate, and how the switch between different implementations can be done. Finally, we established guidelines to evaluate the resulting system.

Acknowledgments I am are grateful to N. Carvalho, C. Fonseca, and L. Rosa for the fruitful discussions and comments during the preparation of this report. This work was partially supported by project "Redico" (PTDC/EIA/71752/2006).

References

1. Amir, Y., Stanton, J.: The spread wide area group communication system. Technical Report CNDS 98-4, The Center for Networking and Distributed Systems, John Hopkins University (1998)
2. Bertier, M., Marin, O., Sens, P.: Implementation and performance evaluation of an adaptable failure detector. In: DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks. pp. 354–363. IEEE Computer Society, Washington, DC, USA (2002)
3. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. SIGOPS Oper. Syst. Rev. 21(5), 123–138 (1987)
4. Birman, K.P.: The process group approach to reliable distributed computing. Commun. ACM 36(12), 37–53 (1993)
5. Birman, K.P., Constable, R., Hayden, M., Hickey, J., Kreitz, C., Van Renesse, R., Rodeh, O., Vogels, W.: The horus and ensemble projects: Accomplishments and limitations. Tech. rep., Ithaca, NY, USA (1999)
6. Cachin, C.: Security and Fault-tolerance in Distributed Systems. IBM Zurich Research Lab (2005)
7. Cadot, S., Kuijlman, F., Langendoen, K., van Reeuwijk, K., Sips, H.: 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. pp. 56–63. ACM, New York, NY, USA (2001)
8. Chandra, T.D., Toueg, S.: Unreliable failure detectors for reliable distributed systems. J. ACM 43(2), 225–267 (1996)
9. Chen, W.K., Hiltunen, M., Schlichting, R.: Constructing adaptive software in distributed systems. In: 21th International Conference on Distributed Computing Systems (21th ICDCS'01). IEEE, Phoenix, AZ (2001)

10. Chen, W., Toueg, S., Aguilera, M.K.: On the quality of service of failure detectors. IEEE Trans. Comput. 51(1), 13–32 (2002)
11. Chockler, G.V., Huleihel, N., Dolev, D.: An adaptive totally ordered multicast protocol that tolerates partitions. In: PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing. pp. 237–246. ACM, New York, NY, USA (1998)
12. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. 33(4), 427–469 (2001)
13. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. 36(4), 372–421 (2004)
14. Dolev, D., Kramer, S., Malki, D.: Early delivery totally ordered multicast in asynchronous environments. In: Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing. pp. 544–553. IEEE (1993)
15. Dolev, D., Malki, D.: The transis approach to high availability cluster communication. Commun. ACM 39(4), 64–70 (1996)
16. Dfago, X., Schiper, A., Urbn, P.: Comparative performance analysis of ordering strategies in atomic broadcast algorithms. IEICE Trans. on Information and Systems E86-D(12), 2698–2709 (2003)
17. Felber, P., Défago, X., Guerraoui, R., Oser, P.: Failure detectors as first class objects. In: DOA '99: Proceedings of the International Symposium on Distributed Objects and Applications. p. 132. IEEE Computer Society, Washington, DC, USA (1999)
18. Fetzer, C., Raynal, M., Tronel, F.: An adaptive failure detection protocol. In: PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing. p. 146. IEEE Computer Society, Washington, DC, USA (2001)
19. Fidge, C.J.: Timestamps in message-passing systems that preserve partial ordering. In: Proceedings of the 11th Australian Computer Science Conference. pp. 56–66 (1988)
20. Fonseca, C., Rosa, L., Rodrigues, L.: Custo da comutação dinâmica de protocolos de comunicação. In: Actas do primeiro Simpsio de Informtica (Inforum). Lisboa, Portugal (2009)
21. Hayashibara, N., Cherif, A., Katayama, T.: Failure detectors for large-scale distributed systems. In: SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems. p. 404. IEEE Computer Society, Washington, DC, USA (2002)
22. Hayashibara, N., Defago, X., Yared, R., Katayama, T.: The φ accrual failure detector. In: SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems. pp. 66–78. IEEE Computer Society, Washington, DC, USA (2004)
23. Hiltunen, M.A., Schlichting, R.D.: 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)
24. Hiltunen, M.A., Schlichting, R.D.: A configurable membership service. IEEE Trans. Comput. 47(5), 573–586 (1998)
25. Hiltunen, M.A., Schlichting, R.D., Ugarte, C.A., Wong, G.T.: Survivability through customization and adaptability: the cactus approach. In: DARPA Information Survivability Conference and Exposition (DISCEX 2000), pp. 294–307 (2000)
26. Hutchinson, N.C., Peterson, L.L.: The x-Kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering 17(1), 64–76 (1991)

27. Jacobson, V.: Congestion avoidance and control. In: SIGCOMM '88: Symposium proceedings on Communications architectures and protocols. pp. 314–329. ACM, New York, NY, USA (1988)
28. Khazan, R.I., Fekete, A., Lynch, N.A.: 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. pp. 258–272. Springer-Verlag, London, UK (1998)
29. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
30. Liu, X., van Renesse, R., Bickford, M., Kreitz, C., Constable, R.L.: Protocol switching: Exploiting meta-properties. In: ICDCS Workshops. pp. 37–42 (2001)
31. Malloth, C.P., Felber, P., Schiper, A., Wilhelm, U.: 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)
32. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proceedings of the 21st International Conference on Distributed Computing Systems. pp. 707–710. IEEE, Phoenix, Arizona (2001)
33. Mocito, J., Rodrigues, L.: Run-time switching between total order algorithms. In: Proceedings of the Euro-Par 2006. pp. 582–591. LNCS, Springer-Verlag, Dresden, Germany (2006)
34. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., Lingley-Papadopoulos, C.A.: Totem: a fault-tolerant multicast group communication system. Commun. ACM 39(4), 54–63 (1996)
35. Pedone, F., Guerraoui, R., Schiper, A.: Exploiting atomic broadcast in replicated databases. In: Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing. pp. 513–520. Springer-Verlag, London, UK (1998)
36. Peterson, L.L., Buchholz, N.C., Schlichting, R.D.: Preserving and using context information in interprocess communication. ACM Trans. Comput. Syst. 7(3), 217–246 (1989)
37. Powell, D.: Group communication (introduction to the special section). Commun. ACM 39(4), 50–53 (1996)
38. Raynal, M., Schiper, A., Toueg, S.: The causal ordering abstraction and a simple way to implement it. Inf. Process. Lett. 39(6), 343–350 (1991)
39. Renesse, R.V., Minsky, R., Hayden, M.: A gossip-style failure detection service. In: IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing Middleware'98, 15-18 (1998)
40. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Commun. ACM 39(4), 76–83 (1996)
41. Rhee, I., Cheung, S.Y., Hutto, P.W., Krantz, A.T., Sunderam, V.S.: Group communication support for distributed collaboration systems. Cluster Computing 2(1), 3–16 (1999)
42. Rodrigues, L., Fonseca, H., Veríssimo, P.: Totally ordered multicast in large-scale systems. In: Proceedings of the 16th International Conference on Distributed Computing Systems. pp. 503–510. IEEE, Hong Kong (May 1996)
43. Rosa, L., Rodrigues, L., Lopes, A.: Building adaptive systems with service composition frameworks. In: Proceedings of the 9th International Symposium on Distributed Objects and Applications (DOA). Algarve, Portugal (2007)
44. Rosa, L., Rodrigues, L., Lopes, A.: A framework to support multiple reconfiguration strategies. In: Autonomics '07: Proceedings of the 1st international conference on Autonomic computing and communication systems. pp. 1–10. ICST (Institute

for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium (2007)

45. Rütti, O., Wojciechowski, P.T., Schiper, A.: Structural and Algorithmic Issues of Dynamic Protocol Update. In: Proc. of IPDPS '06 (20th International Parallel and Distributed Processing Symposium) (2006)
46. Sousa, A., Pereira, J., Moura, F., Oliveira, R.: Optimistic total order in wide area networks. In: SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems. p. 190. IEEE Computer Society, Washington, DC, USA (2002)