From *Appia* to *R-Appia*: Refactoring a Protocol Composition Framework for Dynamic Reconfiguration

Liliana Rosa
Luís Rodrigues
Antónia Lopes

DI–FCUL TR–07–4

March 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

From *Appia* to *R-Appia*: Refactoring a Protocol Composition Framework for Dynamic Reconfiguration

Liliana Rosa
Universidade de Lisboa
lrosa@lasige.di.fc.ul.pt

Luís Rodrigues
Universidade de Lisboa
ler@di.fc.ul.pt

Antónia Lopes
Universidade de Lisboa
mal@di.fc.ul.pt

March 2007

Resumo

Appia is a protocol composition and execution framework that aims at simplifying the design, implementation, and configuration of communication protocols. In this paper, we report our experience in applying *Appia* to build adaptive communication systems. The insights gained with the experience have lead us to build a new version of the framework, called *R-Appia*, that includes a significant number of features aimed at supporting the dynamic reconfiguration of communication protocols. This paper describes and discusses the new functionality of this framework.

1 Introduction

Distributed applications rely on communication services, typically implemented by protocol stacks. A simple example of such a stack is given by the use of TCP over IP, and some data link protocol (e.g. Ethernet). Today, it is easy to find applications that can operate on top of multiple stacks (e.g. simultaneously use both UDP and TCP) or that require stacks far more complex. For instance, fault-tolerant applications often use group communication stacks which include many different agreement and ordering protocols [5].

Protocol composition and execution frameworks aim at simplifying the design, implementation, and configuration of communication protocols. One of the main goals of such frameworks is to promote the design of communication services in a modular way, by encouraging communication functionality fragmentation in different modules, allowing them to be composed in different ways. As a result, the designer has the opportunity to compose protocol stacks that exactly match the application needs. In runtime, the framework's execution environment supports the exchange of data and control information between modules, providing a number of auxiliary services, such as timer management. Some relevant protocol composition frameworks are *Consul* [14], *Cactus* [6], *Horus* [17], *Ensemble* [16], *Appia* [13], and *Mena et al* [12].

There exists a growing class of systems that require adaptive communication services, i.e., communicating services that are able to adapt their behavior in reaction to changes in the environment. In dynamic settings, both the application goals and the operational envelope of a communication service (e.g., node availability, transmission error rate, available bandwidth) may change in runtime. Therefore, at any moment, the current protocol composition may become inadequate, and the system needs to be reconfigured to use a more suitable protocol composition.

The contribution of this paper is the identification of a set of features, typically not provided by existing protocol composition frameworks, that are key to support the dynamic reconfiguration of protocol compositions. The work described here was based on our experience with the development of *Morpheus* [15], a system designed to support policy-driven

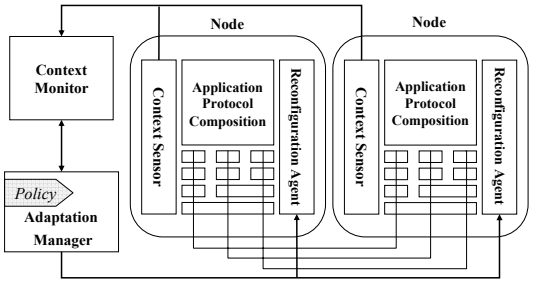

Figura 1: System Architecture of Morpheus

adaptation of protocol stacks that builds on *Appia* framework. Since *Appia*, as most of the relevant protocol composition frameworks, follows a vertical protocol composition approach and an event-driven model, the work described in this paper applies to these frameworks in general. Thus, the components added to *Appia*, aimed at simplifying the construction of adaptive communication systems, can also be added to other frameworks. Moreover, the identified key features must also be present in the frameworks aiming at adaptation. This paper describes these components and key features, analyzing their strengths and weaknesses, and the necessary changes that lead to the development of a new framework version named *R-Appia*.

The rest of the paper is structured as follows. Section 2 describes *Morpheus* architecture. Section 3 introduces the original *Appia* framework, and section 4 resumes motivation for building *R-Appia*. *R-Appia* refactoring is described in two parts. Section 5 describes add-on features and Section 6 resumes changes made to the original framework. Section 7 surveys related work, and Section 8 presents paper conclusions.

2 Proposed Approach to Adaptation

This section provides a general overview of the approach to adaptation that was adopted in *Morpheus*, the system that was used to experiment the suitability of the *Appia* framework for supporting dynamic reconfiguration.

2.1 Overview

The architecture of *Morpheus* is depicted in Figure 1. The system is composed by a set of nodes that participate in a distributed multi-party application. The communication between application entities is supported by a given protocol composition.

The goal of *Morpheus* is to provide support for the adaptation of this protocol composition through a policy-oriented approach. This is achieved as follows. In each node, a number of *context sensors* acquire the information about the local context. The sensed information is delivered to a *context monitor* that manages and interprets it in terms of more high-level concepts, and makes this information available to the *adaptation manager*. This manager is responsible for selecting the most appropriate protocol composition to be used in each node at any given point in time. When a reconfiguration is required, the manager also determines the reconfiguration strategy to be applied, i.e., the coordination between different nodes during the reconfiguration process, as well as the synchronization with the ongoing execution of the application. In the following sections, we describe the key aspects of *Morpheus*'s architecture.

2.2 Reconfiguration Actions

The system assumes that the communication services are obtained through the vertical composition of multiple protocols, i.e., a communication service is provided by a stack of protocol layers.

The communication services in *Morpheus* can be adapted through the combined execution of different kinds of actions: protocol parameters adaptation, exchange of the implementation of a protocol layer, and change of the protocol stack structure.

2.3 Context Monitoring

The adaptation of the communication services in use by an application relies on the sensing of relevant contextual information elements, and on the aggregation and interpretation of these elements. The context information may include information from different sources, ranging from user preferences to hardware characteristics of devices hosting the application [4].

Information capture is performed by context sensors, typically coupled to the relevant software or hardware components whose state requires monitoring. For instance, in an wireless network, the signal strength may be captured by a sensor that has access to the network card device driver and, in a mobile device, the available battery power may be obtained through the operating system interface.

Data capture can be performed on-demand or continuously, often in a periodic manner [1]. Sensors can also spontaneously trigger events that signal infrequent occurrences, such as the failure of a component, or the fact that some control variable exceeded some predefined threshold.

The context information elements acquired by the sensors are collected and interpreted by the context monitor. By combining different elements, the monitor generates the context information required by the adaptation manager. For instance, the monitor may identify high-level properties such as "system unstable" based on low-level context information such as network error rate, connectivity information, etc.

The information produced and held by the context monitor is provided to the adaptation manager through an interface that supports both synchronous (queries) and asynchronous (callbacks) interactions.

2.4 Policies

The adaptation manager's decision on when and how to adapt the communication services is determined by a set of rules expressed using a policy-specification language. The policy language used is based on Event-Condition-Action (ECA) rules [11]. Each rule specifies the events that trigger a given adaptation, the set of conditions that must hold in order to apply the adaptation, and the set of actions that need to be performed. In these rules, the triggering events are tightly coupled with the events asynchronously notified by the context monitor, whereas the condition is expressed as a function of the state stored in the context monitor (that can be queried). Finally, reconfiguration actions specify how the protocol compositions need to be changed. Each action has a scope, that determines the set of nodes and communication channels affected by the reconfiguration.

A detailed description of a policy-specification language for the adaptation of protocol stacks, tailored to the *Appia* system, can be found in [18].

2.5 Reconfiguration Strategy

When the adaptation manager decides to carry out a protocol composition reconfiguration, it proceeds by sending directives to the reconfiguration agents at each involved node, following a specific strategy. These directives establish the coordination of participants in the

	Orchestration	Local Reconf. Mode
Zero Quiescence	Uncoordinated	Direct
Local Quiescence	Uncoordinated	Quiescence
Global Quiescence	Stop-and-go	Quiescence

Tabela 1: Strategy's elements

communication through a particular orchestration and also define how the reconfiguration must proceed locally, at each node. The latter aspect, which we designated by *local reconfiguration mode*, is concerned with: (1) placing the local protocol composition in a safe state to perform the reconfiguration (typically, a quiescent state), and (2) the management of the local state that must survive reconfiguration.

Different reconfiguration actions typically ask for different coordination approaches. For instance, whereas a timeout value of a failure detector protocol can be changed in several nodes without requiring node coordination, the reconfiguration of two nodes communicating using TCP, so that they begin using UDP instead, needs to be performed in a coordinated fashion, as communication would be impossible if one is using TCP and the other UDP.

Similarly, there are different ways of proceeding with the local reconfiguration of a protocol composition, with different costs and applicability constraints. For instance, changing the timeout value of a failure detector protocol in a given local protocol composition can be performed on-the-fly, whereas the replacement of the implementation of a protocol may require to place each affected node in a quiescent state, and even to capture and transfer part of the stacks's state to the new implementation.

Note also that the local reconfiguration modes are, to some extent, independent of the form of coordination adopted. More concretely, the need of reaching a quiescent state is independent of the coordination requirements. Quiescence is required in situations demanding a strong coordination of the participants of the communication, s.a. the exchange of the transport protocol, but also in a situation that does not require coordination. For instance, when an implementation of a protocol is replaced by another compatible implementation (e.g. to install a bug-fix on-the-fly), it may not be necessary to synchronize the reconfiguration in all nodes. In this case, each affected node may reach the quiescent state and perform the reconfiguration locally, without coordination with the remaining nodes.

The reconfiguration strategy of *Morpheus* is defined in terms of three combinations of different orchestrations and local reconfiguration modes, as shown in Table 1. Next, the different elements of *Morpheus* strategy are explained.

2.5.1 Uncoordinated Orchestration

The uncoordinated orchestration is used for all reconfiguration actions that can be realized through the local reconfiguration of the protocol composition at each node, without requiring coordination among nodes. Examples of reconfiguration actions with this property include changes to protocol parameters, or the addition/removal of layers that only have a local impact (e.g., a logging layer).

This orchestration involves the following simple exchange of messages between the adaptation manager and the reconfiguration agents. The manager sends a STEP-ONE message to each participant. This message carries information about the changes to be performed at the local protocol composition, as well as information about the reconfiguration mode to be used, as follows:

Direct This is the simplest and quickest mode of performing a local reconfiguration of a protocol composition. However, its applicability is quite limited, as it defines that it is not necessary to reach a quiescent state nor perform any state transfer. When the reconfiguration agent receives a STEP-ONE message indicating that a *direct* reconfi-

guration mode must be used, it immediately sends back a DONE-ONE message to the adaptation manager.

Quiescence This mode defines that the reconfiguration agent starts by placing the protocol composition in a quiescent state. In addition, it also defines the part of the state that needs to be captured and transfered to the new stack configuration. Ultimately, it may define that no state transfer is required (as happens in the direct mode). Once the quiescent state is achieved, the reconfiguration agent collects the state of the running stack that needs to be transfered. Afterwards, it sends back a DONE-ONE message to the adaptation manager.

The message STEP-ONE has an additional parameter *step* indicating whether other steps are required before the node can proceed with the reconfiguration. The messages sent in the uncoordinated orchestration have a *step*'s value indicating that no more steps are needed and, hence, after sending the DONE-ONE message back to the adaptation manager, the local reconfiguration agent can reconfigure the local protocol composition independently from the remaining nodes. Once the local reconfiguration is concluded, each node sends a COMPLETE message to the adaptation manager, and resumes communication, if the quiescence mode was used.

2.5.2 "Stop-and-go" Orchestration

The "stop-and-go" orchestration enforces a strong coordination among all nodes involved in the reconfiguration. In this orchestration, the manager starts by sending a STEP-ONE message to all participants. As in the uncoordinated orchestration, this message carries information about the changes to be performed at the local protocol composition as well as information about the reconfiguration mode to be applied. However, in this case, the value of the parameter *step* indicates that the node can only proceed with the reconfiguration of the local protocol composition after receiving a STEP-TWO message.

As shown in Table 1, in *Morpheus* this orchestration is only combined with the *Quiescent* mode and, therefore, when a participant receives the STEP-ONE message it locally puts the system in a quiescent state. This forces the communication to be temporarily interrupted. As described before, when the participant is in a quiescent state it replies back to the manager with a DONE-ONE message. As soon as the manager collects DONE-ONE messages from all participants, it sends a STEP-TWO message to inform the local reconfiguration agents that they may proceed. Subsequently, each participant performs the reconfiguration locally and replies with a COMPLETE message. Finally, as soon as the manager collects COMPLETE messages from all participants, it issues a RESUME command, allowing communication to restart.

This orchestration is highly intrusive as it forces protocols to temporarily interrupt the message flow; furthermore, the cost grows linearly with the number of participants. However, certain adaptations, such as TCP replacement by UDP, or exchanging from insecure to secure communication can hardly be achieved by other means.

3 The *Appia* Framework

In this section we provide a brief introduction to the original *Appia* system. *Appia* [13] is a framework that supports implementation and execution of modular protocol compositions. We have selected the *Appia* system to offer runtime support to *Morpheus* mainly because it has been developed "in house" but also because it has a number of features particularly well suited for our goals. Moreover, it is available in the Java programming language, helping rapid prototyping.

Basic Concepts Each *Appia* module is a layer, i.e. a micro-protocol responsible for providing a particular communication service. These layers are independent and can be combined. A combination of layers constitutes a protocol stack that offers a given quality of service (QoS). We use QoS in the broad sense, encompassing order, reliability, security, etc. Once a QoS has been defined, by composing the appropriate layers, it is possible to create one or more communication channels with that QoS. A channel with a given QoS is associated with a stack of *sessions*: for each protocol layer there is a session responsible for maintaining the state required for the corresponding protocols execution.

Sessions interact through the exchange of events. Events in *Appia* are object-oriented data structures. The *Event* class has two fundamental attributes: *channel* and *direction*. The first is a reference to the channel where the event will flow, and the second indicates in which direction the event is flowing along the stack. *Direction* can only take *Up* or *Down* values. Note that a session just forwards an event up or down in a channel, without having explicit knowledge of the concrete protocol that is executed above and below in the stack. This allows the stack to be reconfigured without changing the code of each protocol. Moreover, events can also have a *Message*, which carries information from one end of the channel to the other.

Shared Sessions In *Appia*, two or more channels that have the same given layer may share it, using the same session. In this case, the protocol may correlate events exchanged in different channels with the help of the state maintained by the shared session. With this feature it is possible, for instance, to implement multiplex/demultiplex layers. In such a case, the multiplex session may receive data from the upper channel, code it in some form (or simply fragment it), and send distinct fragments in parallel using different channels; on the receiving end, the session would buffer the received data from both underlying channels until it would be able to reconstruct the original data.

Protocol Programming Model *Appia* enforces a methodology for programming communication protocols. Each protocol is coded as a set of event handlers. The typical event handler receives an event, adds/removes some protocol header to/from the message associated with the event, and forwards the event in the same channel. Each protocol layer has to declare which event types it requires to process, and which type of events it creates. All protocols receive a *Init* event that is automatically generated when the channel is created and becomes ready to operate.

Note that some protocols may temporarily store events in the session state, for instance, to forward correctly out of order received messages, or to assemble a message from different segments. Protocols that manage multiple channels, such as in the multiplexing example of the previous section, may forward events in a channel different from the one the event was received.

AppiaXML *AppiaXML* is an extension of *Appia* that allows the runtime to dynamically instantiate a channel based on its XML description. With *AppiaXML* it is possible to completely specify a protocol composition, as well as the initial values for the control parameters that configure each session. This functionality is particularly useful to build adaptive systems, as it provides the practical means for loading the correct configuration in each node. In particular, when the adaptation manager wants to deploy a new channel on any given node, it can send the channel configuration over the wire as a XML description.

***Appia* Kernel** The *Appia* kernel is the runtime support of the protocol composition framework. It is responsible for scheduling all events that flow in the communication channels. It also provides other useful runtime services such as timer management. For optimized performance, *Appia* uses the information provided by all protocol layers, regarding which events

they process, to create an *event route* for each type of event that flows in a communication channel. The route allows to deliver events only to the layers that handle them.

***Appia* Protocols** *Appia* has been used to support a wide range of applications, including multi-user object-oriented environments, distributed real-time games, collaborative mobile applications, and database replication. These prototypes have been developed at *University of Lisbon* in the context of national and international research projects, involving several researchers and students. Therefore, a significant library of protocols that can be used with the system is available. One of the most relevant protocol sets is a reliable group communication suite. Available group communication services include membership services, reliable multicast services, view-synchrony, ordering services (causal and total order), among others.

4 From *Appia* to *R-Appia*

Appia framework is appropriated for adaptation by multiple reasons. First of all, it promotes the design of communication services as compositions of small independent protocols. Such protocols can be combined in multiple forms, without any change in the source code, allowing fine-grained adaptation of the communication service. Secondly, it supports channel configuration and activation via XML specifications. This eases the design of the adaptation manager, as new configurations can be sent to the reconfiguration agents using XML strings. Lastly, it includes a library of multiple protocols, which allowed us to build early prototypes for adaptation with minimal coding effort.

Still, despite these qualities, during the development of *Morpheus*, we soon found out that several elements were lacking or requiring a major revision to provide adequate support for building adaptive systems, namely:
(1) It does not include a standard mechanism for capturing context information from running stacks, making hard to gather elements that are part of the state of the protocol execution, which are an important source of context information when one targets the adaptation of the communication service. In fact, most protocols maintain control variables, such as error-rate, round-trip time estimates, etc., that are crucial to assess which changes are necessary to optimize the system behavior.
(2) It only supports the static composition of protocols. For performance reasons, all events that flow in a channel carry an *event-route*, the set of sessions that process the event. This feature makes hard to reconfigure a channel while events are flowing in it, as it would require to update the event route of all running events.
(3) It does not include any built-in component to coordinate the reconfiguration in multiple nodes. As seen before, this coordination is fundamental in many reconfiguration scenarios that affect multiple communicating participants.
(4) It does not include a standard mechanism for forcing channels to reach a quiescent state. As discussed before, local reconfiguration of the protocol composition associated with a channel may require the communication channel to reach and temporarily remain in a quiescent state.
(5) It does not provide adequate primitives for the management of the state that must survive reconfiguration.

The next two sections describe in detail a number of components we have developed and a set of changes that we have performed to *Appia* system in order to overcome the problems above and, in this way, obtain a framework that provides adequate support for building adaptive communication systems.

5 Basic *R-Appia* Features

In this section, we present the features that were developed as add-ons of the *Appia* framework, without requiring modifications to the *Appia* kernel.

5.1 Context Sensors

In order to support the capture of context information from running protocol compositions, several components are necessary. We have designed these components and added them to *Appia*, as described below.

To start with, three new event types were defined, namely *GetValueRequest*, *GetValueReply*, and *TrapIndication*. The name of these events are inspired by network management operations such as SNMP [3]. The event *GetValueRequest* is used to obtain information on a specific variable, identified by its name or a standard MIB identifier [9]. *GetValueReply* event returns the requested information. Finally, a *TrapIndication* event asynchronously notifies a change in a specific condition. All protocols that can be potential sources of useful context information should declare the attributes that can be read and the notifications they are able to generate.

Furthermore, we have implemented a *GenericSensor* layer that can be added to any protocol stack (see Figure 3). This layer uses the *Appia* functionality that allows multiple channels to share a single session. More precisely, a *GenericSensor* session is shared by one control channel and multiple observed channels. In the initialization phase, the *GenericSensor* layer receives the name of the control channel and a set of tuples; each tuple contains a variable name and a monitoring period. After initialization, the layer operates by generating *GetValueRequest* for all the requested variables with the requested intervals; this is trivially achieved using the *Appia* timing services. Events *GetValueRequest* are injected in all observed channels. Subsequently, all *GetValueReply* and *TrapIndication* events are encapsulated in a *ContextNotification* event that is sent via the control channel to the context monitor. Finally, the layer also accepts *GetValueRequest* events received on the control channel and propagates them to all observed channels. Naturally, the generic sensor is only useful if the relevant protocols in the reconfigurable channel are able to accept *GetValueRequest* events and generate *GetValueReply* and *TrapIndication* events.

The reader may notice that this generic sensor layer is quite flexible, as it allows: (1) to select just at deployment time the most appropriate control channel to send context information back to the context monitor; (2) to periodically monitor any control variable of any protocol with any monitoring period; (3) to collect and export any trap event; (4) to perform queries on context variables on demand. This demonstrates that it is possible to build generic components that can be used in a variety of scenarios.

It is also possible to build specialized sensors and use them together with our generic sensor. For instance, assume that some protocol does not not provide average values on some control variable; it provides instantaneous values instead. It would be possible to build a sensor that would make multiple readings and send an average value back to the context monitor. Although the average could be performed at the monitor itself, the use of a specialized sensor such as this may save network resources by reducing the number of signaling data that crosses the network.

5.2 Channel Reconfigurator

As we have previously noted, *Appia* optimizes the flow of events in a channel by precomputing the route of each event. The route states which sessions are visited by the event when it transverses the channel. This optimization poses difficulties on channel reconfiguration while events are active. Here, we present a channel reconfiguration technique that circumvents this problem by forcing the channel to reach a quiescent state, where no

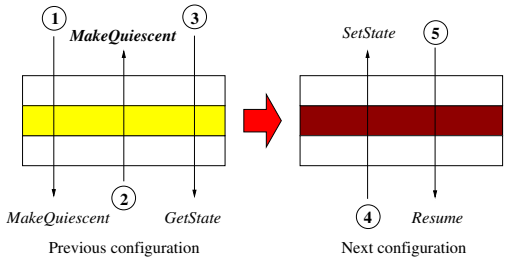

Figura 2: Get and set session state

events are active. Later on, in Section 6, we discuss some modifications to the *Appia* kernel that allowed us to address this problem in a more sophisticated way.

The devised channel reconfiguration technique is partially realized by a local component called the *channel reconfigurator*. This component accepts requests of channel reconfiguration and performs the following steps: first, it flushes all events flowing in the channel, enforcing a quiescent state; afterwards, the state of each session is collected; then the channel is deleted, and a new channel is created with the new configuration (the event routes are recomputed for the new configuration at this point); finally, the activity on the channel is resumed.

To support this technique, the protocols used in the reconfigurable channels must accept and process the following events:
– **MakeQuiescent** is propagated in the channel in the *Down* direction. Before forwarding this event downwards in the channel, a session must ensure that it will not produce more events and that all events it was holding have been either sent or captured in a state variable that can be transfered to the new incarnation of the entire channel. When the event reaches the bottom of the channel, its direction is reversed. When the *MakeQuiescent* event reaches a session traveling upwards, it indicates that all underlying sessions are in a quiescent state. Therefore, when the *MakeQuiescent* event reaches the top of the channel, the entire channel is in a quiescent state.
– **GetState** is propagated in the channel in the *Down* direction. When the event is received, each session adds a state object to the event, which includes all the required state information to be transfered to the next incarnation of the channel.
– **SetState** is propagated in the channel in the *Up* direction. Each session reads the corresponding state object and initializes its state variables accordingly.
– **Resume** is propagated in the channel in the *Down* direction. It starts the activity in the channel.

The sequence of events is illustrated in Figure 2. The reader should notice that this technique to reconfigure a channel can be applied with different coordination strategies. In the remaining of this paper we call this technique *Channel Quiescence*.

5.3 Buffering Layer

The purpose of the buffering layer is to hide the reconfiguration from the application. In particular, it hides the fact that the communication has to be temporarily interrupted to reconfigure the channel. This is achieved by having the buffering layer to store all messages produced while the channel is being reconfigured (i.e., between the *MakeQuiescent* event and the *Resume* event). The buffering layer is implemented by a session that shares two channels: an interface channel, that is used by the application and never is reconfigured (or stopped), and a protocol channel, that can be reconfigured. On steady state, the buffering layer only forwards events from the interface channel to the protocol channel and vice-versa. This is illustrated in Figure 3. We have implemented a generic buffering layer that can be used whenever reconfiguration needs to be hidden from the application (note that it is also

possible to implement the application as an *Appia* layer, in which case the application may be prepared to receive and process the *MakeQuiescent* and the *Resume* events).

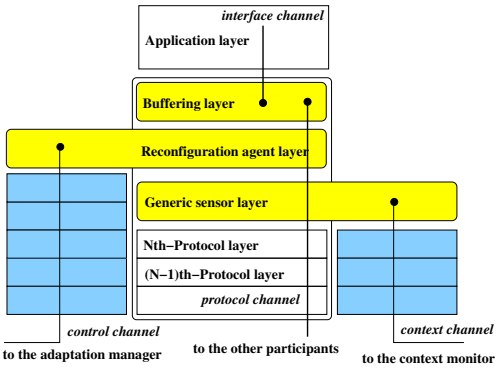

Figura 3: Reconfigurable channel

5.4 Protocol Programming Model

Ideally, it would be useful to build adaptive communication systems out of legacy protocol stacks that were designed without having reconfiguration needs in mind. From our experience, this is very hard to achieve for non-trivial protocol compositions. Our reconfiguration schemes require the protocol implementation to support the following functionalities:
– Provide the context information that might be relevant to express adaptation policies. This is achieved by accepting *GetValueRequest* events and generating *GetValueReply* and *TrapIndication* events, in order to implement sensor functionality. In fact, this functionality should be provided even if dynamic adaptation is not required, as these are basic services of any manageable object (from a systems' management perspective).
– Accept events that allow the relevant configuration parameters to be changed in runtime, for instance, by allowing timeout values to be adjusted dynamically. This can be implemented by accepting *SetValueRequest* events, something that is also part of standard system management interfaces but that was seldom used by *Appia* programmers.
– Have the ability to achieve a quiescent state when requested. This is required for several implemented reconfiguration strategies. Interestingly, group communication protocols also require a quiescent state in order to make a membership view change. Therefore, many protocols designed for *Appia* were already prepared to reach a quiescent state on demand. Unfortunately, all remaining protocols lacked support for this feature.
– Have the ability to store and load the protocol state across different "incarnations" of a protocol session, when a channel is reconfigured. Since this was not a requirement in the original *Appia* framework, existing protocols have no support to allow their state to be captured and loaded.

5.5 Discussion

The set of changes described in the previous sections result from the generic support of adaptation in *Appia*, but can also be applied to other frameworks. The performed changes reflect *Appia*'s strengths and weaknesses.

On one hand, it allowed to build a basic reconfigurable version without any change to the kernel. This was possible thanks to two important features of *Appia*: the ability to have sessions that share multiple channels, and the ability to add new events to the framework (in some frameworks, such as [16], the set of events is fixed *a priori*).

On the other hand, we have also seen that, in order to speed up the event flow in protocol stacks, some of the optimizations in *Appia* (such as the preprocessing of event routes) require a channel to reach a quiescent state before reconfiguration is possible.

The set of changes described above allowed us to realize the three strategy elements of *Morpheus*, previously presented in Table 1.

Zero Quiescence This strategy can be implemented with a simple single step orchestration:

1. When the agent receives the STEP-ONE message from the adaptation manager, it performs the reconfiguration actions listed in the message. Then, the agent replies back with the DONE-ONE message.

This strategy can only be applied in reconfiguration actions that involve changing configuration parameters of protocols, when these parameters can be changed on-the-fly. As discussed before, with the channel reconfiguration technique presented in this section, any reconfiguration that changes the set of protocols that constitute a channel requires to place the channel in a quiescent state.

Local Quiescence This strategy can be used only when the change to the set of protocols that constitute a channel does not imply a change in the structure of the messages that transverse the stack (next section discusses this in more detail). For instance, it can be used to add or remove layers that log events for future analysis. When this strategy is taken, the local reconfiguration agent executes the following steps:

1. When the agent receives the STEP-ONE message from the adaptation manager, it inserts a *MakeQuiescent* event in the target channel. When this event is received back by the agent it collects the channel state with the *GetState* event. When the channel state has been captured, the agent deletes the channel. Then, the agent replies back with the DONE-ONE message.

2. It creates a new channel with the new configuration. Then, inserts the *SetState* event in the channel to load the state captured from the previous configuration into the new configuration. Later, it replies with a COMPLETE message and resumes communication by inserting the *Resume* event in the new channel.

It is important to note that, when using only the basic *R-Appia* features, the channel state is always captured due to the channel reconfiguration technique used. In the next section, it is shown how this problem is overcome by changing *Appia* kernel.

Global Quiescence In this case the local reconfiguration agent starts by executing the step 1. previously described, and, afterwards, executes the following steps:

2. When the agent receives a STEP-TWO command, it creates a new channel with the new configuration. Afterwards, it inserts the *SetState* event in the channel to load the state captured from the previous configuration into the new configuration. Then, it replies with a COMPLETE message.

3. Finally, when the agent receives a RESUME command it inserts the *Resume* event in the new channel.

6 Advanced *R-Appia* Features

We now describe a set of changes that were performed in *Appia*'s kernel to weaken the applicability constraints of the *Zero Quiescence* strategy and obtain more efficient implementations of the other two strategies. This allows to improve the efficiency of the reconfiguration process as it is possible to apply the more lightweight strategy in a larger number of situations and use more efficient strategies in the remaining cases.

6.1 Pool of Headers

In the *Appia* system, when a node needs to send data to another node, it has to build an object of the *Message* class. A message operates like a stack: when a message transverses downward a protocol stack, each layer may push a protocol header to the message. This header must be popped by the corresponding entity on the remote node, when the messages transverse the protocol stack upwards. *Appia*'s message interface is heavily inspired in early protocol frameworks such as *x*-kernel [8] and adopted in other systems [17, 16].

The modeling of a message as a stack of headers implies a strong coordination among participants during reconfiguration, since implicitly the configuration on the sending side must be the same as the configuration at the receiving side for any message exchanged. This is unfortunate, as there are multiple examples of reconfigurations that may be performed without strong coordination.

Consider the example of a *Stats* layer that is used for performance analysis. The stack includes a layer that timestamps every message with control information (such as time of transmission) that is later analyzed at the recipient size. The *Stats* provides no functionality for the application. Its purpose is to gather statistics about protocol performance. Therefore, it should be possible to remove this layer with minimal coordination, such that the addition or removal actions would cause a minimal impact on the ongoing application message flow. However, if headers are pushed and popped from the message, this is impossible. Consider a layer *A* above *Stats*. If the *Stats* layer is removed first from the sending side, when the recipient tries to pop the corresponding header, it will get the header from layer *A*. If the layer is removed first from the recipient side, when the layer *A* tries to pop its header it will get the *Stats* header instead.

To solve these problems, we decided to change radically the way the information is carried in an *Appia* message. In *R-Appia* a *Message* is composed of a *pool of headers*. Each header in the pool is identified by a textual label. The methods available to handle headers are "*addHeader(label,header)*", "*getHeader(label)*", "*removeHeader(label)*", and "*hasHeader(label)*". The method "*addHeader(label,header)*" adds an header associated with the given label; "*getHeader(label)*" reads the contents of the header associated with the given label; "*removeHeader(label)*" removes from the pool the header associated with the given label, and "*hasHeader(label)*" checks if the message contains the header with the given label. The management of the label namespace is orthogonal to the *R-Appia* operation. However, *R-Appia* requires each layer to declare the labels of the headers it produces and requires, which mimics the *Appia* conventions to received and produced events. Therefore, the runtime can detect clashes in the header label namespace.

The pool of headers approach solves the problems we have described above because headers no longer need to be pushed/popped in an order that is configuration dependent. In the example above, if the *Stats* is only removed from the recipient side, layer *A* can still read its own header; on the other hand if the *Stats* is only removed from the sender side, the corresponding *Stats* session on the recipient side can use the method *hasHeader* to check that the header is missing and skip the processing of the message.

6.2 Dynamic Update of Event Routes

As discussed in Section 5.2, *Appia* only supports the static composition of protocols as it pre-computes routes for all events that flow in a communication channel. This feature forces to place a channel in a quiescent state every time we need to change the set of sessions, which is an overkill for some kinds of reconfigurations. For instance, using again the logging layer as an example, it should not be necessary to stop the traffic to add or remove this layer, given that it has no interaction with the remaining protocols and no effect on the quality of service of the data stream. Therefore, we have decided to change the *Appia* kernel to support the dynamic composition of protocols. In order to explain the change, we shall

beforehand briefly describe the operation of the *Appia* kernel.

The core of *Appia* consists of a single threaded *event scheduler*. The scheduler maintains, for each channel, a list of events that have been generated by the different sessions in the protocol stack. Each event e in the scheduler stores information about the stack of sessions that handle the event (stk_e) and also the next session to be visited on this stack (nxt_e). This information is called the *event route*. Note that the event route of each specific event is a subset of all sessions in the protocol stack (given that some sessions are not interested in processing some events).

When a channel is reconfigured without being set to a quiescent state, all the event routes stored in the event scheduler have to be updated. To do so, it is necessary to change the route of every event e according to the reconfiguration performed in the stack. If a new session accepting e is added to the protocol stack, that session is added to stk_e; similarly, if a session accepting e is removed from the protocol stack, the session is also removed from stk_e. Additionaly, nxt_e needs to be changed accordingly, taking into account the next session to visit before reconfiguration and the relative position of the added/removed session in the stack. The routes of all events in the scheduler are updated atomically.

6.3 Strategies Revisited

As mentioned before, the modification to the *Appia* kernel was driven by the goal of being able to broad the applicability of *Zero Quiescence* strategy and obtain more efficient implementations of the other two strategies.

On one hand, while with the basic *R-Appia* mechanisms *Zero Quiescence* could only be applied to change protocol parameters, now it can also be used to perform any reconfiguration that changes the channel QoS, and does not require quiescence nor coordination. For instance, to add or remove the *Stats* protocol referred in the example of Section 6.1, one can now use the *Zero Quiescence* strategy.

On the other hand, because the new *Appia* kernel supports channel's QoS reconfiguration on-the-fly, without creating a new incarnation of the channel, the implementation of *Local Quiescence* and *Global Quiescence* does not involve any more the capture and transfer of the whole state of the channel. For instance, to perform any kind of reconfiguration that consists in replacing one protocol set by another set of compatible protocols, only protocols belonging to the set are relevant for state transfer. In fact, it is only necessary to capture the state of the control variables that need to be transfered from the old session to the new session to allow the new protocol to resume execution from the point where the previous protocol was. This information is produced by the adaptation manager and sent to the local reconfiguration agent through the STEP-ONE message. Consider, for example, the exchange between two different public-key encryption protocols for digital signatures. Each protocol stores different public keys, besides the node's private key. During execution, public keys from different nodes are distributed. Thus, switching between protocols requires that the known public keys are inherited by the new protocol, as well as the private key.

7 Related Work

Among the protocol composition frameworks referred in Section 1, only *Cactus* [6] and *Ensemble* [16] offer dynamic reconfiguration. In *Cactus* such support is made available through the dynamic loading of micro-protocols, allowing to change the provided service during runtime. However, the framework lacks basic context handling support, as well as differentiated reconfiguration strategies.

Ensemble framework offers dynamic reconfiguration support by means of a switching protocol, that forces protocols in the old stack to a quiescent state, and distribute and start the new stack. However, the approach relies in changing all stacks with a specific

identification to a new configuration, which is the same on every node. Moreover, automated adaptation decision process, context capture and monitoring, and support for uncoordinated reconfiguration (such as header pools) are not addressed.

Dynamic reconfiguration of component-based software systems [2, 10, 7] raises many issues similar to those of protocol composition frameworks. Adding, and removing both components, and links during runtime also requires to address concerns such as consistency, automation of the adaptation process, and state transfer. However, techniques developed to build adaptive systems in component-based platforms are not easily transposable to protocol composition frameworks, given the specificity of these frameworks.

8 Conclusions

Building adaptive communications systems is a task that can be strongly simplified if appropriate support is provided by an underlying protocol composition framework. In this paper we have reported our experience in building adaptive communication sub-systems. We have identified a set of relevant features, typically not provided by existing frameworks, that are necessary to implement a comprehensive set of reconfiguration strategies. Based on these observations, and using *Appia* as the runtime support framework, we describe the re-design of the framework, in order to build a modified and augmented version that we have called *R-Appia*. The new framework supports the implementation of a wider range of reconfiguration strategies, leading to the deployment of more flexible and efficient communication sub-systems. We are currently using the *R-Appia* system to build adaptive collaborative applications in the context of the MICAS project.

Acknowledgments

This work was partially funded by FCT project MICAS – Middleware for Context-aware and Adaptive Systems – (POSI/EIA/60692/2004) through POSI and FEDER.

Referências

[1] A. Acharya, M., and J. Saltz. Sumatra: A language for resource-aware mobile programs. In *MOS '96*. Springer-Verlag, 1997.

[2] T. Batista and N. Rodriguez. Dynamic reconfiguration of component-based applications. *PDSE*, 2000.

[3] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. Simple network management protocol (snmp), 1990. RFC 1157.

[4] G. Chen and D. Kotz. A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA, 2000.

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

[6] M. Hiltunen, R. Schlichting, C. Ugarte, and G. Wong. Survivability through customization and adaptability: The cactus approach. In *DARPA Information Survivability Conference & Exposition*, volume 1. IEEE, 2000.

[7] H.Liu and M.Parashar. Component-based programming model for autonomic applications. In *ICAC'04*. IEEE Computer Society Press, 2004.

[8] N. Hutchinson and L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Trans. Softw. Eng.*, 17(1):64–76, 1991.

[9] M. Johns and M. Rose. Identification mib, 1993. RFC 1414.

[10] A. Ketfi, N. Belkhatir, and P. Cunin. Automatic adaptation of component-based software: Issues and experiences. In *PDPTA '02*, pages 1365–1371. CSREA Press, 2002.

[11] D. McCarthy and U. Dayal. The architecture of an active database management system. In *ACM-SIGMOD International Conference on Management of Data*. ACM Press, 1989.

[12] Sergio Mena, André Schiper, and Paweł T. Wojciechowski. A step towards a new generation of group communication systems. In *Proceedings of Middleware '03: the 4th ACM/IFIP/USENIX International Middleware Conference*, volume 2672, pages 414–432. Springer, June 2003.

[13] H. Miranda, A. Pinto, and L. Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *ICDCS*, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[14] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Tucson, AZ (USA), 1991.

[15] J. Mocito, L. Rosa, N. Almeida, H. Miranda, L. Rodrigues, and A. Lopes. Context adaptation of the communication stack. *International Journal of Parallel, Emergent and Distributed Systems (IJPEDS)*, 21(2):169–181, 2006.

[16] R. Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using ensemble. *Softw. Pract. Exper.*, 28(9):963–979, 1998.

[17] R. Renesse, K. Birman, and S. Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39(4):76–83, 1996.

[18] L. Rosa, A. Lopes, and L. Rodrigues. Policy-driven adaptation of protocol stacks. In *International Conference in Autonomic Systems*, Silicon Valley, CA, USA, 2006. IEEE Computer Society.