Building Adaptive Systems with Service Composition Frameworks

Liliana Rosa, Luís Rodrigues, and Antónia Lopes

Faculty of Sciences, University of Lisbon, Portugal
lrosa@lasige.di.fc.ul.pt {ler,mal}@di.fc.ul.pt

Abstract. Frameworks that support the implementation and execution of service compositions are a fundamental component of middleware infrastructures that support the design of adaptive systems. This paper discusses the requirements imposed by adaptive middleware on service composition frameworks, and discusses how they have been addressed by previous work. As a result, it describes the design of a novel adaptation-friendly service composition framework that takes into consideration the requirements at three different levels: service programming model level, adaptation-friendly services level, and kernel mechanisms level.

1 Introduction

Today's applications need to be designed to operate in a wide range of heterogeneous devices, including servers, PCs, PDAs, or mobile phones. Given this diversity, it is fundamental to be able to design and deploy adaptive applications. An adaptive application is able to change its behavior to better match the (functional and non-functional) expectations of the user. For instance, by adjusting the multimedia quality exchanged among different participants, according to the available network bandwidth.

Unfortunately, building distributed applications that can monitor changes in their execution environment, as well as in the user requirements, and react to those changes by adapting their behavior is an inherently complex task. A task that can be greatly simplified by the use of appropriate adaptive middleware. A key component of a middleware platform to support the construction and execution of adaptive applications is a software framework that facilitates the composition of services. By allowing services to be composed in different manners, and supporting the dynamic reconfiguration of service compositions, it becomes easier to adapt the behavior of applications that are built in a modular manner.

Network protocols have been specified for a long time in a modular way, using the layer abstraction. Typically, a communication system is built from a vertical composition of multiple protocols layers. Therefore, it comes as no surprise that many software frameworks to build configurable communication services have been designed, implemented, and used in different contexts. Some of the most relevant protocol composition frameworks are *x*-kernel [1], *Cactus* [2], *Horus* [3],

Ensemble [4], *Appia* [5], *Eva* [6], and *Samoa* [7]. Given the significant amount of experience that has been gathered with these systems, they become obvious candidates to inspire the construction of a service composition framework to include as part of a middleware platform to support adaptive applications.

This paper looks at existing protocol composition frameworks from the point of view of their adequacy to support the implementation of adaptive services. Based on our experience in building a generic architecture to support adaptation [8], we identify a number of requirements that need to be satisfied by any service composition framework. We then analyze how existing protocol composition frameworks address these requirements. The contribution of this paper is the identification of a set of features, lacked by many of the existing protocol composition frameworks, that are key to support the dynamic reconfiguration of service compositions. Moreover, we describe how we have addressed these requirements in the implementation of an adaptation-friendly service composition framework named *RAppia*.

The rest of the paper is structured as follows. Section 2 introduces protocol composition frameworks. Section 3 identifies a set of requirements imposed by adaptive middleware on composition frameworks, and Section 4 analyses how these are addressed by existing frameworks. The design and implementation of *RAppia* is described in Section 5. Finally, Section 6 concludes the paper.

2 Protocol Composition Frameworks

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, that can be composed in several ways. As a result, the designer has the opportunity to compose communication services that exactly match the application needs. A second important goal of these frameworks is to provide an efficient execution environment for protocol compositions, by providing runtime services that support the exchange of data and control information among components, time management services, buffer management services, etc.

The reader should be aware that there are similarities among composition frameworks and general purpose operation systems. Typically, an operation system includes a kernel, services that can be implemented partially in the kernel and partially in user level (such as windows management), a number of user level services (for instance, the command interpreter and system utilities), and a programming model (processes, file system interface, synchronization primitives, etc). In some sense, a service composition framework is a specialized operating system (in fact, one of the first protocol composition services was even called a "kernel" [1]). Thus, in this paper, when we refer to service composition frameworks we analyze them taking a global perspective, considering both the kernel functionality, the services typically provided with the framework, and the programming model enforced by it.

Multiple protocol composition frameworks have been built [1–7]. Although all of these frameworks aim at achieving similar goals and are based on the same foundations, inspired by the original work of *x-kernel* [1], there are some significant differences among them.

In *x-kernel*, *Horus*, *Ensemble*, *Cactus*, and *Appia*, communication among protocols is performed by the exchange of events. With the exception of *Cactus*, all frameworks support vertical protocol compositions, i.e., events are processed in order by all the protocols in the stack. In *Cactus*, events can be processed in parallel. In *x-kernel* and *Horus* events are delivered to all protocols (which may process them or just forwards them in the stack). *Ensemble* proposed some offline tools to extract fast-paths for most common events. *Appia* and *Cactus* allow each protocol to subscribe only the events it is interested in processing. On the other hand, both *Samoa* framework [9] and *Bast* [10] protocol library follow different approaches. *Samoa* also relies in protocol compositions but with a service-based design. Therefore, the framework kernel is particularly different from the remaining frameworks. In this case, the interaction between protocols is achieved using remote method invocations. In this approach, each module exports a set of executers, listeners, and interceptors, each being responsible for a different service: requests, replies, and notifications. In *Bast* each protocol is an object, thus, interaction relies in method invocation, and the composition model is not strictly vertical.

Among these frameworks, only *Cactus* [11], *Ensemble* [12], and *Samoa* [9] have addressed the problem of dynamic adaptation, supporting the runtime reconfiguration of protocol compositions. However, these efforts have considered only a limited set of protocols (for instance, group communication, in the case of *Ensemble*) and specific reconfiguration strategies. As we will discuss later in the text, none of these frameworks can claim to provide generic support for multiple reconfiguration strategies.

3 Adaptation Requirements

Our previous work on the development of a generic architecture to support the adaptation of service compositions [8, 13], gave us insight on the needs, challenges, and goals that adaptation brings. The highlights of adaptation are related to context monitoring, to detect changes that will trigger adaptation, and adaptation management, that conducts and performs all the process of reconfiguring the composition. Our experience allowed to identify several requirements that have to be satisfied by composition frameworks. We note that different requirements impact different aspects of the composition framework: some require changes to the runtime support provided by the framework (also known as the framework kernel), some can be satisfied by adding additional services to the framework, others affect the programming model enforced by the framework. These requirements are identified and described in detail in the following sections.

3.1 Context Monitoring

The context information characterizes the execution context. Since the execution context may change with time, this information has a dynamic nature. Thus, given that the execution context is dynamic, a particular configuration of the application, that was adequate in given context may become inadequate later on, and require adaptation. Therefore, it is of utmost importance to maintain the context under constant monitoring, such that the context information reflects the current state of the environment.

The context information may include information from different sources, ranging from user preferences to hardware characteristics of devices hosting the application [14]. This information can be generated by the services themselves, or captured from other origins, such as the operating system or the device. The information itself can be used to infer other context properties, i.e. higher level context information, such as system stability, based on low-level context information such as network error rate, connectivity information, etc. Context information capture can be performed on-demand, or continuously, in a periodic manner [15]. Moreover, services can produce notifications that signal infrequent occurrences, such as the failure of a component, or that some control variable exceeded a predefined threshold. From these observations on context capture, the following requirements can be identified:

> **Requirement 1**: the composition framework should support a programming model that makes easier for sources of context information to make this information easily accessible (in particular when these sources are the composable service implementations themselves).

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

To perform adaptation is not enough to gather context information; it is also necessary to analyze the collected information in order to detect relevant changes. The analysis can be directly embedded in the mechanisms used to collect the context information or may be performed by an external component. In either way, the following requirement can be identified:

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

3.2 Reconfiguration Actions

In this paper, we are concerned with the construction of adaptive distributed systems whose adaptation logic can be separated form the core application logic. In this way, it is assumed that the structure of the application is organized into

two discrete layers, with the core application logic built on the top of a composition of domain-specific and general middleware services. Adaptiveness results from the dynamic reconfiguration of this composition of services, in reaction to changes in the users' preferences or in the execution context.

There are two main ways in which the application may be adapted. To start with, the behavior of each individual service may be adapted, usually by setting predefined configuration parameters [16, 17]. Furthermore, when an appropriate composition framework is used, one may also change the services included in the service composition and the way these services are composed [18, 19]. When we restrict ourselves to communication services, reconfiguration of the composition boils down to the addition, removal, or exchange of protocols. Therefore, we identify the following requirement:

> **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.

When applying a reconfiguration action, the correctness of the service composition has to be preserved. To achieve this goal, several issues need to be addressed during the reconfiguration process. A first issue is related to the amount of required synchronization among the nodes involved in the reconfiguration. For instance, in some cases, each node may perform the local reconfiguration of the service composition without explicit coordination with other nodes; in other cases, a node may not be allowed to proceed with the local reconfiguration until it becomes aware that all the other nodes are also ready to reconfigure. Another issue is related to the state information that may have to be transferred from one system configuration to the other. The third issue is related to the dynamics of each individual service during reconfiguration. Namely, in some cases, a service may be required to be placed in a quiescent state before reconfiguration is performed. Note that different services impose different constraints on the way issues above are handled and, for any given service, different reconfiguration actions may also impose different constraints [13]. Thus, the mechanisms enumerated should be rich enough to satisfy a wide range of constraints, such that the reconfiguration may be performed with the minimal interference on the execution of the services in the composition. This results in the following requirement:

> **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.

3.3 Selection of Adaptation Targets

We are interested in building *distributed* adaptive applications. Therefore, service compositions will be executed in multiple nodes of the system. As a result, when

a reconfiguration needs to be performed, it may need to affect all nodes or just a subset of the nodes involved in the application. Furthermore, only a subset of the service composition may be affected by the reconfiguration.

When specifying the adaptation logic of a system, it is very hard to specify it in a generic and reusable manner if one is required to explicitly name each individual instance of every service that is affected by the adaptation. On the contrary, it is much more powerful to specify the adaptation target indirectly, for instance, using service type hierarchies or using meta-information [20] to tag all services with their properties. The service composition framework may contribute to simplify the implementation of an adaptive system if it provides the programming abstractions and the runtime mechanisms that allow to map these high level abstractions (such as service type hierarchies) in run-time artifacts, for instance, using a reflective approach. Thus:

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

4 Adaptation Support in Existing Composition Frameworks

To understand the suitability of protocol composition frameworks for adaptation, it is important to analyze how each of the requirements identified in the previous section already is, or can be satisfied, by existing protocol composition frameworks.

4.1 Addressing the Requirements

Requirement 1: *the composition framework should support a programming model that makes easier for sources of context information, in particular when these sources are the composable service implementations themselves, to make this information easily accessible.*

Most composition frameworks that have been developed to support protocol composition are event-based, i.e., different services communicate by exchanging events. Thus, the preferable method to make context information available is via the exchange of context events. The event model simplifies the implementation of context notifications: a service that wants to provide a notification about a relevant change in the context information needs simply to create and trigger a new *ContextNotification* event. When context information needs to be read on demand, each service must be ready to process *ContextQuery* events and respond with *ContextAnswer* events.

At first sight, it may seem that every protocol composition framework is equally fitted to satisfy this requirement. However, there are a number of implementation and modelling issues that have a significant impact on how this support is provided. To start with, context information is often service specific.

Thus, the programmer will likely need to refine the base events provided by the framework. Thus, the composition framework cannot limit the set of events exchanged among services to a set of fixed events defined a priori (as, for instance, the *Horus* system). Furthermore, when context is read on-demand, a *ContextQuery* event needs to be delivered to all services that can potentially answer the query. To avoid the event to be delivered to every service of the composition and avoid a performance overhead, the framework should allow each service to explicitly list which events it is interested in (to our knowledge, only *Cactus* and *Appia* support this feature). Finally, the framework should encourage programmers to proactively provide support for context gathering in the service implementations. Thus, the events such as *ContextNotification*, *ContextQuery*, and *ContextAnswer* should make part of the service implementation model. To our knowledge, none of the existing protocol composition frameworks provides this feature explicitly.

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

When building distributed adaptive applications the adaptation policy typically depends on the global context, i.e., of the aggregate context information collected from the different participants in the system. Therefore, it is not enough to support the local gathering of information. Each node should provide support for exporting context information to other nodes. To support on-demand reading of context information, each node must accept remote invocation from other nodes. To disseminate context information, nodes should be connected to a context dissemination bus. This type of support can be added to any of the existing composition frameworks, given that it may be implemented as a set of additional services. Still, to our knowledge, no composition framework includes such services in their distributions, although a fairly detailed pattern language [21] could be used to provide the necessary support.

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

As soon as it is possible to gather and distribute local context information, it becomes possible to analyze and interpret this information to extract the relevant information for the adaptation. Although the analysis can be potentially executed in a single central location, that collects all the context information gathered from all the nodes in the system, in some cases this approach may introduce inefficiencies in the system. For instance, consider that, for adaptation purposes, one is concerned with the average value of a context variable measured in a specific node in the system. The average could be computed at a central location, based on multiple remote readings of the context variable. However,

it is possible to save signaling traffic, if the average is computed directly at the source node of the context information. To support the later approach, it is required that the context gathering and dissemination subsystem can be built as a composition of services itself. This is possible to achieve with any of the existing protocol composition frameworks.

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.*

Although all existing protocol composition frameworks support offline configuration of the service compositions, only a few support the modification of the composition in runtime. From those that support dynamic reconfiguration, some severely restrict the way a composition may be reconfigured in runtime. For instance, *Ensemble* only supports the replacement of a vertical composition (a protocol stack) to another (even when both stacks have several layers in common), avoiding the problems caused by having part of the composition operational while the rest is being changed. From this point of view, *Cactus* is the most flexible of all existing composition framework, as it allows for services to be added and removed in runtime without restrictions.

The reconfiguration process can be also simplified if the addition, removal, and exchange of services to a given composition can be controlled from a remote node (for instance, a reconfiguration manager). This means that the composition framework should include a monitor able to interpret reconfiguration commands that may be activated, for instance, via remote invocations. To our knowledge, none of the existing frameworks supports such interpreter.

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.*

Several protocol composition frameworks, such as *Ensemble*, *Cactus*, or *Samoa*, have implemented concrete instances of the mechanisms enumerated above. However, these mechanisms are usually designed with the goal of implementing a small number of predefined reconfiguration strategies, i.e, a particular sequence of operations such as coordination, enforce quiescent state, state transfer, etc. For instance, *Ensemble* implements a reconfiguration strategy that requires the composition of each node to reach a quiescent state; the state is then captured; a new composition is instantiated and the state loaded into the configuration at every node; finally, the new composition is restarted. *Cactus* and *Samoa* offer more efficient strategies but, in practice, the mechanisms supported only serve the predefined, built-in, strategies, and are only applicable in a limited number of situations. To our knowledge, no composition framework as attempted to offer

a library of mechanisms required to support the coordination among nodes, to transfer service state information between services, and to enforce a quiescent state of a service that can be combined in different manners to implement multiple strategies.

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

Some existing protocol composition frameworks offer these mechanisms. These mechanisms can be based on reflection techniques, provided by the meta-level architectures offered by the language in which they are implemented. Although well developed reflective mechanisms are used in different contexts [22, 23], some even involving protocol compositions [24], their use is rudimentary in protocol composition frameworks, due to complex issues, s.a. protocol composition consistency and dependencies, or event flow. *Ensemble*, *Cactus*, and *Appia* frameworks allow to identify the protocols based on their names. *Samoa* framework supports the separation between the notion of protocol specification and protocol implementation but this is not enough when adaptation is not limited to the exchange of protocol implementations of the same protocol specification (the single adaptation action that is currently supported in *Samoa*).

4.2 Discussion

When discussing how the requirements are addressed by existing protocol composition frameworks, we have also identified that each requirement can be satisfied at a different level of abstraction. Some requirements may require specific support from the protocol composition framework runtime (for instance, the ability to change the composition in runtime). Other requirements can be satisfied by a number of complementary services that can be implemented on top of an existing composition frameworks. Finally, other requirements are better satisfied by enforcing a particular service programming model. We have observed that, although most of these requirements have been previously addressed by different frameworks, none of the existing composition framework satisfies completely the full set of requirements. Moreover, some of these requirements identified in the context of protocol composition frameworks also apply to component-based frameworks. However, these requirements have to be address in a different manner.

5 An adaptation-friendly Composition Framework

As a result of the previous analysis, we have implemented a service composition framework, named *RAppia*, that fulfills the set of requirements we have identified. This service composition framework has been built as an extension to one of the protocol composition framework surveyed: the *Appia* [5]. In the next paragraphs we describe the design and implementation of *RAppia*.

5.1 *RAppia* Basics

RAppia is a service composition framework implemented in the Java programming language. It inherits the composition model from the *Appia* protocol composition framework, that is common to many other similar frameworks (such as *x-kernel*, *Horus*, and *Ensemble*). In *RAppia* services can be composed in a layered manner, creating stacks of services. Typically, services at the bottom of a service composition offer more basic functionality (such as reliable multicast communication) and services at the top of the service composition support higher level abstractions (such as distributed shared object, publish-subscribe, etc).

An instance of a service composition is named a *service channel*. Each layer of a service channel is an instance of the corresponding service in the service composition. Thus, a service channel consists of a stack of service instances. Each instance maintains the state required to provide the desired service. Note that an application may create multiple service channels with the same composition (for instance, to maintain multiple shared objects).

Service instances interact through the exchange of events. Events in *RAppia* are object-oriented data structures. The *Event* class has two fundamental attributes: *channel*, and *direction*. The first is a reference to the service channel where the event will flow, and the second indicates in which direction the event is flowing along the service stack. Note that a session just forwards an event up or down in a channel, without having explicit knowledge of the concrete service that is executed above and below in the stack. This allows the stack to be reconfigured without changing the code of each service implementation.

When building distributed applications, many services are distributed. Furthermore, many services require the exchange of messages among different nodes. The information that needs to be sent over the wire is included in a special field of the events used for inter-service communication called a *Message*.

In *RAppia*, two or more service channels that share a given service may opt to share the same instance of that service. A shared service implementation may correlate events exchanged in different service channels with the help of locally maintained state.

Grounded on these basic mechanisms, the adaptation support is built considering three different aspects: the service programming model, adaptation-friendly services, and kernel mechanisms. These aspects are described next.

5.2 Service Programming Model

The adaptation requirements have been taken into consideration in the programming model used to implement services for *RAppia*. This has been reflected into three separate aspects: the set of events that need to be taken into consideration by each service implementation (which address *requirements 1 and 5*), how service properties are exposed (which addresses *requirement 6*), and how service implementation may exchange control information in a distributed setting (which is related to *requirement 5*).

Event Processing In *RAppia* a service is implemented as a set of event handlers. In runtime, when events are delivered to a service, the appropriate handler is called. Typically, a handler does some processing and forwards the event to the next service in the composition. The framework does not restrict the type hierarchy of events that can be triggered and exchanged in the system. Still *RAppia* defines a number of "system" events that should be handled by any service implementation. These include events to provide easy access to context information produced by the protocols (see *requirement 1*), events to handle state transfer and to place the service in a quiescent state (see *requirement 5*). More precisely, the following events are defined by *RAppia*:

- *ContextQuery*, *ContextAnswer*, and *ContextNotification* events. The first event is used to query a service for specific context information (such as the available bandwidth of a node at the present time), the second to reply to the query event(the reply with the bandwidth reading), and the later to allow a service to provide an asynchronous notification of context information (for instance, a drop in the bandwidth to zero). It is interesting to notice that although many composition frameworks define a number of mandatory events (for instance, an *Init* event used to initialize a service), to the best of our knowledge, no previous framework has been concerned with this sort of functionality, even if this is extremely relevant as these are basic services of any manageable object (from a systems' management perspective).
- *SetParameter* event. This event is used to update configuration parameters in runtime such as, for instance, timeout values.
- *MakeQuiescent* and *Resume* events. The first event is used to request a service to reach a quiescent state (as we have noted, often reconfiguration can only be performed if the service is in a quiescent state). This event is propagated in the channel in the *Down* direction. When the event reaches the bottom of the channel, its direction is reversed and when it reaches the top of the channel, the entire channel is in a quiescent state (as depicted in part of Figure 1). The second event, *Resume*, is used to resume the service after reconfiguration.
- *GetState* and *SetState* events. These events allow to transfer the service state from one instance to another, whenever the reconfiguration requires instances to be swapped (for instance, to install a software update). As illustrated in Figure 1, *GetState* event 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 state information to be transfered. The *SetState* event is propagated in the channel in the *Up* direction, after reconfiguration. Each session reads the corresponding state object and initializes its state variables accordingly.

Type Hierarchies The definition of adaptation targets meta-information, namely for individual services and service channels, can be achieved through type hierarchies. The meta-information from services is defined based on the properties of

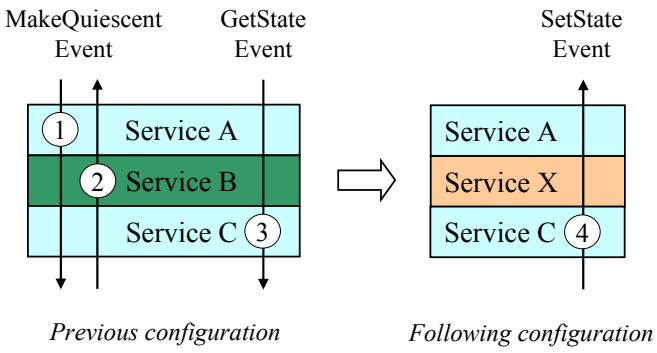

Fig. 1. Replacing service B by X: reaching quiescence and state transfer.

the services such as: group communication, ordering, reliable, etc. Each service is tagged with the properties that it offers, from a well known set. The association of meta-information with service channels cannot be based in the same principle since channels with the same composition can be used for different purposes. Therefore, the meta-information is based on the type of task they perform, for example: control, audio, text, video, etc.

The association of meta-information with services and service channels allows to define type hierarchies, based on the tag hierarchy. Therefore it will exist a hierarchy of service types and another of service channel types. These hierarchies are domain dependent, in the sense that applications with different domains may require different hierarchies. Further details on service type specification and hierarchies can be found in [25].

Message Headers Most composition frameworks support a *message* abstraction that can be used by service implementations to exchange data with remote peers. In a service channel, each service may add/remove its own data to/from the message. The information added/removed by each service layer is typically called the *service header*.

There are two main approaches to manage service headers that have been implemented in existing protocol composition frameworks. One approach models the message as a stack of headers, exporting a push/pull interface to add/remove headers. This is the approach most widely adopted. Unfortunately, this solution is not very adaptation-friendly as it requires a strong coordination during reconfiguration (for instance, a header cannot be pushed unless the corresponding service is active in the remote node to perform the matching pull). Another approach, adopted in the *Cactus* [2] framework, consists in modelling the message as a *pool of headers*. This approach is more flexible, given that the header can be add/removed in different orders. *RAppia* adopted this approach.

Each header in the pool is identified by a textual label. The methods available to handle headers are "*addHeader(label,header)*", "*getHeader(label)*", "*remove-*

Header(label)", and "*hasHeader(label)*". The method "*addHeader(label,header)*" adds a 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 *RAppia* operation. However, *RAppia* requires each protocol 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.

5.3 Adaptation-Friendly Services

RAppia includes two adaptation-friendly services: a generic and configurable context sensor (that addresses *requirement 2*) and a reconfiguration monitor (that addresses *requirements 4* and *5*). These services are described in the next paragraphs. Note that these services could also be adapted to be integrated in other composition frameworks, for instance, to *Cactus*.

Context Sensor The context sensor is a service that is able to locally handle the capture of context information from running service compositions (as described in *requirement 2*). The context sensor is depicted in Figure 2, and works as follows.

The context sensor belongs to multiple service channels: a remote invocation channel, a context notification dissemination channel, and one or multiple sensed service channels, whose purpose is described below.

- The remote invocation channel is used to allow remote nodes to query context information on the sensed service channels. The context sensor receives context queries from this channel and forwards it to all sensed service compositions. Subsequently, it collects the correspondent context answers and sends back a reply on the sensor invocation channel.
- The context notification dissemination channel is used to disseminate to one or more remote nodes context notifications generated by any of the sensed compositions. The generic sensor simply intercepts any notification generated by one of the sensed compositions and forwards it to the notification dissemination channel. The sensor is oblivious to the composition of the notification dissemination channel. By selecting an appropriate dissemination channel, notification can be sent point-to-point to a centralized context monitor, in multicast to multiple nodes, or injected in a publish-subscribe infrastructure.
- The sensed service compositions channels are one or more channels whose context is locally monitored by the generic sensor.

Furthermore, the sensor can be also requested to perform periodic readings of on-demand readable context information and autonomously generate notification

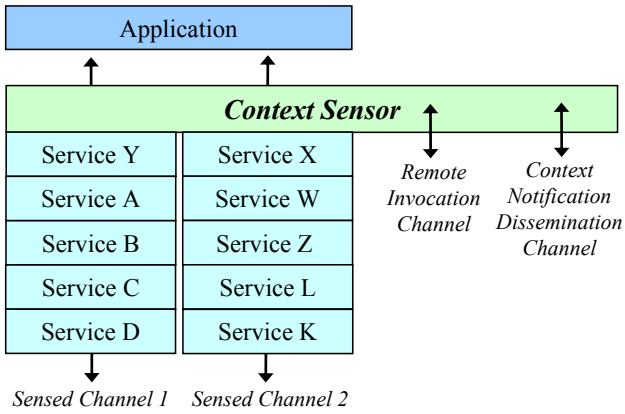

Fig. 2. Context sensor.

with a configurable period. Therefore, the sensor is prepared to, upon request, generate context notifications for variables that otherwise, would have to be read using explicit polling.

Finally, by carefully composing the notification dissemination channel, the programmer may easily introduce local processing at the sensed node to reduce network traffic. For instance, by adding a filter service to notification dissemination channel, one can prevent notifications, whose value is below a given threshold, to be disseminated to the network. In a similar manner, it is possible to include more sophisticated services in the notification channel, for instance, to compute the average of multiple notifications.

Reconfiguration Monitor The reconfiguration monitor is a service that interacts directly with the kernel of the composition service framework and exports a control channel through which it receives multiple *reconfiguration commands*. The reconfiguration monitor is depicted in Figure 3. Each reconfiguration command instructs the monitor to take one or more particular steps of a given reconfiguration sequence. The commands exported by the reconfiguration monitor are as follows.

- *MakeQuiescent*: this command instructs the monitor to put one or more services in a quiescent state, using the *MakeQuiescent* event.
- *Resume*: this command instructs the monitor to resume the activity of a service that was previously put in a quiescent state.
- *Store/LoadState*: these commands determine the capture of state information, and the loading in the end of the reconfiguration. For this purpose the monitor uses the *GetState* and *SetState* events.
- *Reconfigure*: this command instructs the monitor to reconfigure the composition of a given service channel. The reconfiguration involves one or more

of the following actions: remove a service from the service channel, to add a service to a service channel, or to replace an instance of a service by an instance of an alternative service.

For more details on the reconfiguration monitor and the commands please refer to [13].

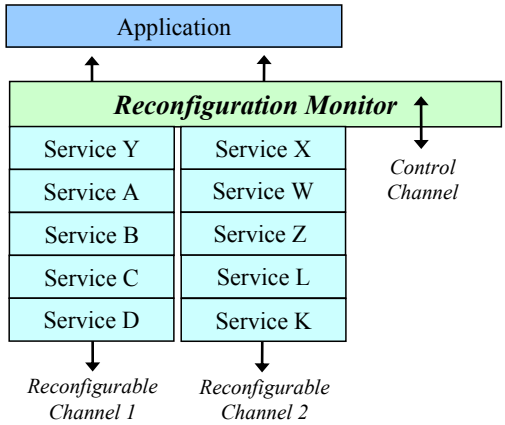

Fig. 3. Reconfiguration monitor.

5.4 Kernel Mechanisms

To address *requirement 4*, the kernel of the *RAppia* composition framework includes two adaptation-friendly mechanisms that, to our knowledge, are not supported by any other composition framework: automatic buffering of events addressed to services in a quiescent state and automatic update of event routes, as described below.

Event Buffering As we have discussed previously, in order to reconfigure a service one may be required to put that service in a quiescent state. Typically, when in a quiescent state, the service is unable to process new events. Therefore, the *RAppia* kernel is able to recognize when a service is in a quiescent state and buffer all events addressed to that service. As soon as the service is resumed, the *RAppia* kernel restarts the delivery of events to the service. This functionality allows a service to be reconfigured without forcing the entire service channel to be put in a quiescent state.

Dynamic Update of Event Routes The *RAppia* kernel is able to use information about which events are handled by each service to optimize the flow of events in a service composition. In particular, for each type of event, an event route is created. This ensures that an event is only delivered to the services that are interested in handling that event.

In an adaptive setting, the composition of a service channel may change in runtime. Furthermore, *RAppia* does not require the entire composition to be set in a quiescent state in order to perform the reconfiguration. Therefore, the *RAppia* kernel is built such that event routes are automatically recomputed when a reconfiguration occurs.

5.5 Discussion

We have implemented a prototype of the *RAppia* framework with the described features. This prototype is currently being used to build middleware systems for mobile networks, whose dynamic settings demand adaptation support. In this middleware, the *RAppia* adaptation-friendly services play an important role. Sensors can be configured to capture different context information, and the reconfiguration monitor allows to develop several different strategies to apply the reconfiguration actions, that are tailored to the service being reconfigured. Moreover, these mechanisms allowed us to build both a context monitor (to reason about context information), and an adaptation manager to control the adaptation process. A detailed description of these additional middleware components is outside the scope of this paper (the interested reader is referred to [13]).

6 Conclusions

Service composition frameworks are a significant component of any adaptive middleware infrastructure. Given the large experience in the design and implementation of composition frameworks oriented for communication protocols, it is interesting to use them as the basis for a adaptation-friendly service composition framework. This paper has identified a set of requirements imposed by adaptive middleware on composition frameworks. Subsequently, we have analyzed how these requirements have already been addressed in the context of protocol composition frameworks. Based on this analysis we propose an adaptive friendly service composition framework that has been obtained by extending an existing protocol composition framework with an augmented programming model, new adaptive services and a set of adaptation-friendly kernel mechanisms.

Acknowledgments

The authors are grateful to the anonymous referees for their comments on a previous version of this paper. This work was partially funded by FCT project MICAS – Middleware for Context-aware and Adaptive Systems – (POSI/EIA/60692/2004) through POSI and FEDER.

References

1. Hutchinson, N.C., Peterson, L.L.: The x-kernel: An architecture for implementing network protocols. IEEE Trans. Softw. Eng. **17**(1) (1991) 64–76
2. Hiltunen, M.A., Schlichting, R.D., Ugarte, C.A., Wong, G.T.: Survivability through customization and adaptability: The cactus approach. discex **01** (2000) 0294
3. van Renesse, R., Birman, K.P., Maffeis, S.: Horus: a flexible group communication system. Communications ACM **39**(4) (1996) 76–83
4. 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, New York, NY, USA, ACM Press (2001) 56–63
5. 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 (ICDCS-21), IEEE Computer Society (2001) 707–710
6. Brasileiro, F., Greve, F., Tronel, F., Hurfin, M., Narzul, J.P.L.: Eva: An event-based framework for developing specialized communication protocols. In: NCA '01: Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01), Washington, DC, USA, IEEE Computer Society (2001) 108–120
7. Wojciechowski, P., Rütti, O., Schiper, A.: SAMOA: A Framework for a Synchronisation-Augmented Microprotocol Approach. In: Proc. of IPDPS '04 (18th International Parallel and Distributed Processing Symposium). Volume 01., Los Alamitos, CA, USA, IEEE Computer Society (2004) 64–74
8. Rosa, L., Rodrigues, L., Lopes, A.: Building adaptive services for distributed systems. Technical report, Dept. Informatics, University of Lisbon (2007)
9. Rütti, O., Wojciechowski, P.T., Schiper, A.: Service interface: a new abstraction for implementing and composing protocols. In: SAC '06: Proceedings of the 2006 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2006) 691–696
10. Garbinato, B., Guerraoui, R.: Flexible protocol composition in bast. In: ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (1998) 22–30
11. Chen, W.K., Hiltunen, M.A., Schlichting, R.D.: Constructing adaptive software in distributed systems. In: ICDCS '01: Proceedings of the The 21st International Conference on Distributed Computing Systems, Washington, DC, USA, IEEE Computer Society (2001) 635–643
12. van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., Karr, D.: Building adaptive systems using ensemble. Softw. Pract. Exper. **28**(9) (1998) 963–979
13. Rosa, L., Rodrigues, L., Lopes, A.: A framework to support multiple reconfiguration strategies. Technical report, Dept. Informatics, University of Lisbon (2007)
14. Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Technical report, Hanover, NH, USA (2000)
15. Acharya, A., Ranganathan, M., Saltz, J.H.: Sumatra: A language for resource-aware mobile programs. In: MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet, London, UK, Springer-Verlag (1997) 111–130
16. Kwon, Y., Fang, Y., Latchman, H.: Performance analysis for a new medium access control protocol in wireless lans. Wirel. Netw. **10**(5) (2004) 519–529

17. Kwon, Y., Fang, Y., Latchman, H.: Improving transport layer performance by using a novel medium access control protocol with fast collision resolution in wireless lans. In: MSWiM '02: Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems, New York, NY, USA, ACM Press (2002) 112–119
18. Ketfi, A., Belkhatir, N., Cunin, P.Y.: Automatic adaptation of component-based software: Issues and experiences. In: PDPTA '02: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press (2002) 1365–1371
19. Liu, H.: A component-based programming model for autonomic applications. In: ICAC '04: Proceedings of the First International Conference on Autonomic Computing (ICAC'04), Washington, DC, USA, IEEE Computer Society (2004) 10–17
20. Crawley, S., Davis, S., Indulska, J., McBride, S., Raymond, K.: Meta information management. In: FMOODS '97: Proceeding of the IFIP TC6 WG6.1 International Workshop on Formal Methods for Open Object-based Distributed Systems, London, UK, UK, Chapman & Hall, Ltd. (1997) 193–202
21. da Silva e Silva, F.J., Kon, F., Yoder, J., Johnson, R.: A pattern language for adaptive distributed systems. In: SugarLoafPLoP'2005: Proceedings of the 5th Latin American Conference on Pattern Languages of Programming, Campos do Jordão, Brazil (2005) 19–48
22. Chiba, S., Masuda, T.: Designing an extensible distributed language with a meta-level architecture. In: ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming, London, UK, Springer-Verlag (1993) 482–501
23. Fabre, J., Nicomette, V., Perennou, T., Stroud, R., Wu, Z.: Implementing fault-tolerant applications using reflective object-oriented programming. Technical report (1995)
24. Agha, G., Frølund, S., Panwar, R., Sturman, D.: A linguistic framework for dynamic composition of dependability protocols. In: Dependable Computing and Fault-Tolerant Systems VIII, IFIP Transactions, Springer-Verlag (1993) 345–363
25. Rosa, L., Lopes, A., Rodrigues, L.: Policy-driven adaptation of protocol stacks. In: ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems, Washington, DC, USA, IEEE Computer Society (2006) 5–12