Building Adaptive Services for Distributed Systems

Liliana Rosa
Luís Rodrigues
Antónia Lopes

DI–FCUL TR–07–21

October 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.

Building Adaptive Services for Distributed Systems

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

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

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

October 2007

Abstract

There exists a growing class of distributed applications that require adaptive middleware services, i.e., services that are able to monitor changes in the execution environment, and in the user requirements, reacting to these changes by adapting their behaviour. This paper presents a framework that supports the definition, implementation, and execution of reconfigurable service compositions, and puts forward an approach to the construction of adaptive distributed applications. Adaptiveness is achieved through the dynamic reconfiguration of service compositions in accordance with high-level policies. The framework allows those reconfigurations to be carried out transparently to the application. This approach is illustrated using a messaging application.

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 behaviour to better match the (functional and non-functional) expectations of the user. For instance, changing of the multimedia quality exchanged among different participants, according to the available 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 behaviour, is an inherently complex task. A task that can be greatly simplified by the use of appropriate adaptive middleware technologies. This paper presents a middleware framework that supports the definition, implementation, and execution of reconfigurable service compositions, and puts forward an approach to the construction of adaptive distributed applications in this framework.

The framework supports the implementation of middleware services and their composition, and allows adaptation to be expressed externally to the rest of the system, through high-level policies. The framework incorporates mechanisms to monitor the relevant context information, and adapt service compositions according to a given adaptation policy. Services compositions residing in different hosts are realized in such a way that they can be reconfigured transparently to the application, through the coordination of the participant hosts according to different strategies. The approach is illustrated using an adaptive messaging application as a case study.

2 Approach Overview

2.1 Defining adaptive applications

The approach proposed in this paper addresses the construction of adaptive distributed systems whose *adaptation logic* can be separated form the *core application logic* [1], and defined uniquely in terms of changes in the users' requirements and the system's operational envelope. 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.

More concretely, the core application layer (or just application, for short) uses a set of channels assuming that each offers a given quality of service (QoS). A channel QoS is realized through a dynamic composition of middleware services: each channel is associated with a composition of services that can be reconfigured when it is not accomplishing what it is intended to do, or better functionality is possible. Given that the application only uses the channel abstraction, reconfigurations are performed transparently to the application.

In order to illustrate our approach and the framework we wish to put forward in this paper, we use a rich messaging application as the main example. Its specification will evolve, being fully described and discussed in Section 5. For now, consider that it supports common text messages and shared drawing among the participants of a chat group. The draw is composed of points, being each line a set of linked points. Dragging and clicking the mouse in the drawing area produces a message for each point (linked or not). The user may choose between a responsive mode that favors the delivery of drawing messages and a power-save mode that may involve the *Aggregation* of different messages and, hence, a delay in their delivery. Moreover, the application allows the user to decide, at any time, if he wants to publish its location or keep it private and if we wants to see the locations of the other participants that are currently publishing their location.

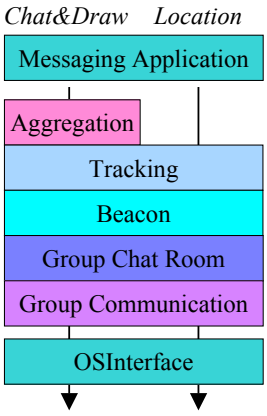

Figure 1: A snapshot of the local architecture of a messaging application

The messaging application core layer includes the user interface, and services to process text messages, drawing, and the acquisition and management of user's preferences. These core services use two channels:

- the channel *Chat&Draw* to send and receive chat and draw messages and to send the user's requirements concerning the delivery mode;

- the channel *Location* to send the user's requirements concerning the privacy of his location and to receive information about the location of the other participants of the chat group.

Each channel is associated with a reconfigurable composition of low-level, middleware services. To support the quality of service that is expected from the channels *Chat&Draw*, and *Location*, the lower layer of the messaging application includes a generic *Group Communication Service* that provides group membership and multicast services, and domain-dependent services. The latter are: a *Beacon Service*, that inserts the user location in all messages; a *Tracking Service*, that retrieves the other users locations from received messages; an*Aggregation Service* that joins several drawing points in a single message, to decrease the number of messages that are sent, resulting in power save; and *ChatRoom Service*, another abstract service, that provides a common interface to the lower level communication protocols in use.

There are three available services realizing different approaches for aggregating points: *Line Aggregation*, all points of the same line are sent in the same message; *Text Aggregation*, all pending points are sent when a text message is also sent (the one with minor cost); and *Periodic Aggregation*, all points produced during a specific time period are sent in the end of that period. There are also two available concrete subtypes of *ChatRoom*: *Group ChatRoom*, a chat room service appropriated to be placed on the top of group communication; and *PointToPointChatRoom*, a chat room service appropriated to be placed on the top of point-to-point communication. As it will be clear later, this is useful because, when only two participants are active, the chat group may operate on top of point-to-point channels, instead of using the more expensive group communication primitives.

Figure 1 illustrates possible service compositions associated with *Chat&Draw* and *Location* channels. These compositions would make sense to be in place when the user wants to make its location public to the other participants of the chat and has not selected the responsive mode.

As mentioned before, the adaptable behaviour of the application results from the reconfiguration of the service composition associated to the channels used by the core layer. In our approach, we consider that the adaptation logic that rules the reconfiguration of these service compositions is defined through high-level policies, using rules following the event-condition-action (ECA) [2] style. In the messaging application, the adaptation logic includes rules defining that:

- the *beacon service* is present in the composition associated to the *Location* channel only when the user's preferences is to have its location visible to the other participants of the chat;
- the *tracking service* is present in the composition associated to the *Location* channel only when at least one chat participant has a public location;
- an *aggregation service* is present in the composition associated to the *Chat&Draw* channel only if the user selected the power-save mode and, in this case, the different bandwidth conditions will decide which aggregation service is the most suitable.

As illustrated in the previous example, the definition of an adaptation policy presupposes the existence of a *service model* describing the services available for composition as well as a *context model* describing the context information that is sensed and made available for the process of decision making.

2.2 Providing Support for Adaptation

To offer automatic adaptation support is necessary a framework that supports the composition of services, allows the dynamic reconfiguration of those compositions and also offers

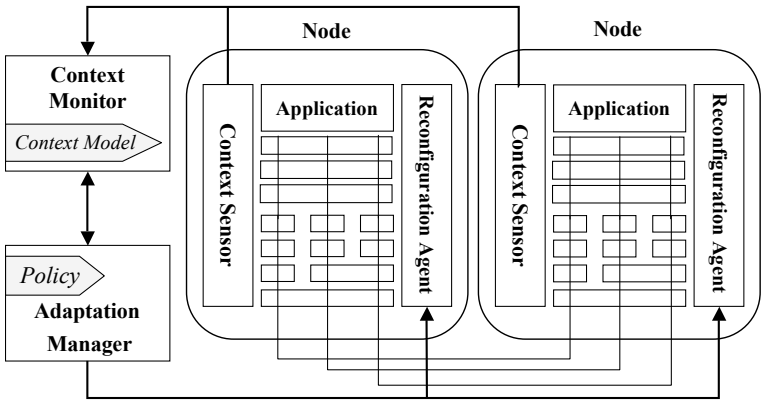

Figure 2: The proposed architecture for adaptive distributed applications

the necessary mechanisms for automatic reconfiguration, such as context sensing, context monitoring, decision making, and adaptation management. For this purpose, we have developed *RAppia*, a framework including components such as a *Context Monitor*, an *Adaptation Manager*, a *Context Sensor* and a *Reconfiguration Agent*. These are general components in the sense they can be used in the construction of a wide variety of applications by adopting the architecture depicted in Figure 2. The tailoring of the framework to the specific application is achieved mainly through the development of the appropriated adaptation policy and context model.

Adaptation Management. In the proposed architecture, the application has two types of components involved in the adaptation management — local reconfiguration agents and a central adaptation manager. Adaptation is controlled by the manager, taking reconfiguration decisions according to the adaptation policy. The adaptation manager is also responsible for guiding the nodes during the adaptation process, either preparing them for reconfiguration, coordinating, or ordering specific reconfigurations. At each node, the reconfiguration agent is responsible for performing the necessary reconfigurations, as ordered by the manager. These reconfigurations comprise the reconfiguration of services compositions, through addition, removal, and exchange of services as well as the fine-tuning of services parameters.

Context Monitoring. Context information comprises all relevant information whose evolution can trigger adaptation. This context information may have several sources: hardware, software, execution environment, user's preferences, among others [3]. Depending on the context information source, more than one type of sensor may be needed, in other cases a single generic sensor may handle all context information. For example, to keep information on the error rates of different services, a single generic sensor can be used. This sensor collects information from all the target services through a request and answer approach. On the other hand, keeping information on the available bandwidth at specific intervals, and CPU usage would require different specific sensors.

In the proposed architecture, the application has two types of components involved in the gathering, management and dissemination of contextual information — local sensors and a central context monitor. The information is captured by the local sensors and concentrated in the context monitor. This component not only keeps all information, but also interprets it according to the context model, detecting changes, that are communicated to the adaptation manager.

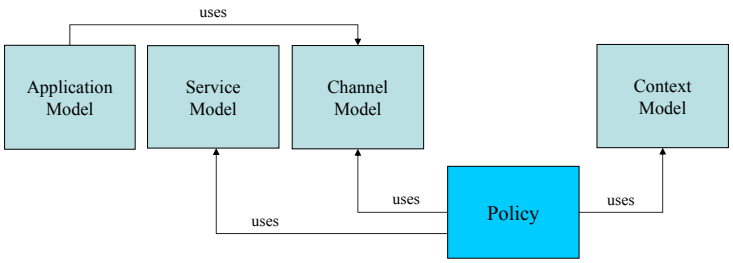

Figure 3: The elements involved in the definition of an application's adaptation logic

3 Defining Dynamic Reconfiguration

In our approach, the adaptable behaviour of an application results from the dynamic reconfiguration of the service composition associated to the channels used by the core layer. To support the definition of the runtime adaptation of those service compositions at a high-level of abstraction, we have developed primitives for the specification of high-level adaptation policies and four types of models — *service models*, *channel models*, *context models* and *application models*. Among other purposes, those models describe the elements that can be used in the adaptation policies. They allow to specify when and how the service composition associated with each channel has to be reconfigured in terms of a logical view of those service compositions and channels.

As depicted in Figure 3, an adaptation policy uses a service model, a channel model and a context model. The service model describes the services that are available in the service layer and, hence, can be used in service compositions. The channel model describes the channels whose service compositions are the target of adaptation. The context model describes the context information required to define the situations in which adaptation is needed. Adaptation policies, models and their relationships are described in the next sections.

3.1 Service Model

A service model describes the services that are available for composition in terms of a hierarchy of types reflecting the functionality provided by those services. As usual, this notion of sub-typing subsumes the *is a* relationship. Moreover, all the characteristics of the supertype also apply to the elements of the subtype. Service types can be concrete, designating a specific service for which an implementation is available, or abstract, representing simply the characteristics of a group of other service types. Naturally, the service type hierarchy can have multiple levels. Figure 4 depicts part of the service type hierarchy for the messaging application. In this model, *LineAggregationService* is an example of a concrete service whereas *AggregationService* is an abstract one.

The service type hierarchy supports the specification of policy rules in an abstract and flexible manner. For instance, it is possible to specify a rule that applies to any service of a given abstract type, without concern for which concrete service is being used in the composition at a particular point in time. This is particularly important in a adaptive system, where the concrete service being used may change as a result of a reconfiguration. When an application uses multiple service compositions simultaneously, the type architecture also allows to specify reconfiguration rules that apply to all services of a given type, without requiring a specific enumeration of these services.

In addition to the type hierarchy, the service model also describes, for each service type, the configuration parameters and which context information can be provided. It is

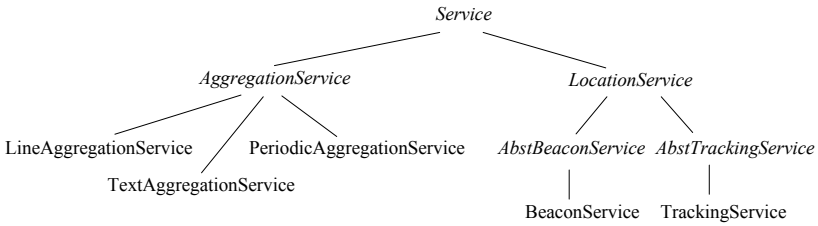

Figure 4: Part of the service hierarchy for the messaging application

considered that a service can provide context information in two different forms: through queries and traps. The first consists in maintaining context information in its local state that can be queried (such as, e.g., the average number of interface messages aggregated in a single transport message). The second consists in having the service raising an alarm event, a trap, when some exception condition occurs (e.g., when the network connectivity is lost).

The full description of services in service models has the following structure:

```
{abstract} service serviceType is
 subtypeOf
  [serviceType]*
 parameters
   [parameterSignature]*
 queries
   [querySignature]*
 traps
   [trapSignature]*
```

For instance, the periodic aggregation service used in in the messaging application could be described as follows.

```
service PeriodicAggregationService is
 subtypeOf
  AggregationService
 parameters
  period:long
```

3.2 Channel Model

As discussed in Section 2, in our approach, the connection between the application and service layers relies on the notion of channel. In other words, channels are the unique mechanism available for the exchange of information between the two layers. At run-time, a channel is associated with a stack of service instances (or just service stacks, for short) that process the information sent by the application and produce and deliver information to the application. As illustrated in Figure 1, typically, at the bottom of the stack there is an interface to some operating system level service. For instance, a stack of services may send and receive messages using a socket interface, or save data in the persistent store using the file system interface.

An application may use multiple channels simultaneously, each one for a different purpose, i.e., a different quality of service. For instance, the messaging application has two different channels. One is used for text, and drawing messages, while the other is used for exchange of information related to location of the participants of the chat. In order to support the specification of the scope of the reconfiguration actions in a flexible way, channels are described in a channel model also in terms of a hierarchy of types defining a subtype relationship. To some extend, the channel types in this hierarchy reflect the quality of service that is expected from their instances.

Figure 5 depicts the channel model developed for the messaging application. For instance, this model allows to express that, in a given node, when the remaining lifetime of the battery

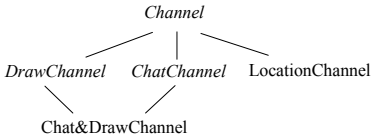

Figure 5: The channel hierarchy for the messaging application

is below a certain threshold, all the channel instances of type *ChatChannel* or *DrawChannel* must have the *PeriodicAggregationService*. Although the application now only uses a channel instance of type *Chat&DrawChannel*, this rule could be reused if later on we decide to change the application in order to send text and draw message through different channels.

3.3 Application Model

The application model is quite simple and mainly describes which channels are used by the application. If the application captures user-defined preferences and these preferences need to be passed to the service layer, the application model also defines how this context information is provided and where. Changes in the relevant user-defined preferences are modeled as context traps, raised by the application, and sent through the channels where they are relevant. The description of an application in the application model has the following structure:

```
[use channel channelName:channelType
 traps
   [trapSignature]* ]*
```

Our messaging application, which uses two different channels, is described by the application model presented below. Whereas the user's preferences concerning the broadcasting of its location are passed to the service layer through two traps delivered in the *Location* channel, the mode of delivery of messages is sent through the *Draw&Location* channel. The user's preferences on the location channel indicate if the user wants to make its location private or public and if it is (or not) interested in receiving location information from the other participants.

```
use channel Chat&Draw:Chat&DrawChannel
 traps
   responsiveMode
   powerSaveMode
use channel Location:LocationChannel
 traps
   changePrivacy(isPublic:bool)
   changeTracking(on:bool)
```

3.4 Context Model

The description of context information relies on two types of mechanisms: *observables* and *events*. Observables are part of the context information kept in the state of the context monitor, while events are indications of asynchronous changes in the context. Events can carry extra information, as the identity of the node that raised the event.

Observables and events can be defined as *imported* or *exported*. Imported observables/events refer to context information provided to the context monitor by sensors. Exported observables/events refer to context information that is provided by the context monitor to the adaptation manager and, hence, they can be used in the adaptation policy. Typically, exported information is obtained by interpreting, combining and/or constraining imported information from different sources.

Imported Context Information. This context information is often obtained from the services that comprise the services compositions, through queries and traps. Imported information is the basic context information that feeds the context monitor: it is stored for later analysis and is used to generate the exported information. Observables and events are described as follows:

```
imported observable returnType accessName([parameter]*)
  [periodically: number]
imported event eventName
  [attributeName: returnType]*
```

Observables have a return value and may also have one or more parameters. The observable may be defined to be captured periodically with a certain sampling time (for instance, through the query of a sensor that has that information). Events may have attributes, carrying different types of information. In the messaging application example, it is possible to import the events that signal the user's preferences (such as location privacy). It is also possible to import observables that maintain state regarding the system operation, such as an observable that indicates the number of participants in the application (this observable is maintained by the ChatRoom service).

```
imported event changePrivacy
  id: nodeId
  isPublic: bool
imported observable int numberOfParticipants()
```

Exported Context Information. This information is produced from the imported context information correlation through a number of calculations. The exported information can be described in the following manner:

```
exported observable returnType accessName([parameter]*)
  expressionOfReturnType
exported event eventName
  [attributeName: type]*
  when [condition]
  with [attributeName = expressionOfType]*
```

The definition of an exported observable includes an expression describing how its value is calculated from other observables, and/or events. The definition of an exported event includes a *when* clause that allows to express what is the condition, expressed in terms of imported events or changes in other observables, that once evaluated to true triggers the publishing of the event. Through the clause *with* we can express the values of attributes of this event. These values can, for instance, be inherited from imported events or calculated using observables. The following example illustrates the definition of some exported events included in the context model for our messaging example.

```
exported event publicLoc
   id: nodeId
   when changePrivacy && changePrivacy.isPublic
   with id = changePrivacy.id
exported event privateLoc
   id: nodeId
   when changePrivacy && !changePrivacy.isPublic
   with id = changePrivacy.id
```

The logical architecture of the service compositions associated with channels is an important information that in most of the applications need to be taken into account while specifying adaptation. For this reason, this information is considered to be also part of the context information. In this way, in addition to application-specific exported events and observables, every context model includes some built-in exported observables. For instance, it includes **exported observable** *bool hasService(ServiceType,ChannelType,nodeId)*, that allows to query if a certain service type is present in the service composition of a channel type in a given node.

3.5 Adaptation Policy

An adaptation policy defines *when* adaptation should be performed, *how* the application should be adapted, and *what* to adapt, i.e., which are the targets of the adaptation. In our case, this description is achieved by a set of rules specified using a policy specification language [4, 5]. Each rule follows an event-condition-action (ECA) style, specifying the events that trigger the rule; the conditions that must apply to activate the rule; and the reconfiguration actions to be applied. All the elements that are needed to specify these rules, are defined by the previous models. More concretely, in each rule:

- the conditions that trigger the adaptation are expressed in terms of the context information exported by the context monitor. Policies use the elements defined in the context model to refer context changes.
- how the system is adapted is expressed in terms of a number of actions that can be performed on the current service compositions. The reconfiguration actions available are: tuning parameters that change the behaviour of a service and add, remove, or replace an ordered set of services by another one.
- the target of adaptation expresses which services, channels, and nodes of the distributed application should be affected by the reconfiguration actions, in what is called *action scope*. Policies use the elements defined in service and channel models to specify this scope.

From a global perspective, the policy is a set of rules that define all circumstances that require adaptation. Each rule has the following general syntax:

```
When triggerCondition
[With stateCondition]
Do {reconfigurationAction
 [Where nodeScope]
 [For serviceScope][Apply compositionScope]}+
```

The *triggerCondition* is an exported event, and specifies when the rule is triggered. The *stateCondition* is a function of one, or more exported observables that specifies the conditions that need to be satisfied in order for the rule to be applied. Each *reconfigurationAction* has a scope composed of a *node scope* (defining the target nodes), a *service scope* (determining the target services using the types defined in the service model), and a *channel scope* (describing the target channels using the hierarchy on the channel model). The scopes are optional and, by default, an action is considered to target all nodes/services/compositions. In the messaging example, the adaptation policy includes the following rule:

```
When publicLoc
With !hasService(AbstBeaconService,LocationChannel,publicLoc.id)
Do addServices([BeaconService],above)
 Where publicLoc.id
 For ChatRoomService
 Apply LocationChannel and Chat&DrawChannel
```

The rule states that, when a trigger *publicLoc* occurs, and the node has not the beacon service already active, the beacon service should be added to the location channel, immediately above the ChatRoom service. For more details about the policy specification language please refer to [6].

4 Support for Dynamic Reconfiguration

In this section we describe the runtime support for dynamic reconfiguration of services compositions that has been designed and implemented in the context of our work. In the proposed architecture there are two distinct types of components. On the one hand,

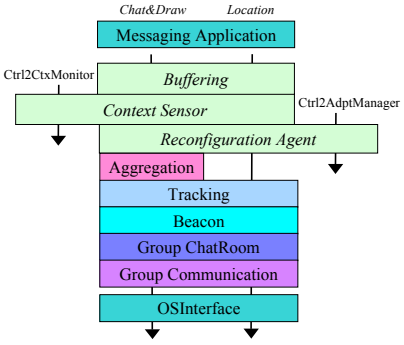

Figure 6: A messaging application with dynamic reconfiguration support

there are centralized components — the context monitor and the adaptation manager — that operate decoupled from the target adaptive application. On the other hand, there are local components that execute in the same nodes and are coupled to the distributed application. These local components are realized by auxiliary services that are embedded in the application service composition, augmenting it with mechanisms for context sensing and adaptation management. The set of auxiliary services, besides the local generic context sensors and reconfiguration agents presented before, also includes a buffering service, whose role is to maintain the service adaptation transparent to the core application layer.

We have presented in Figure 1 an example of the service compositions for our messaging application. Figure 6 shows the real constitution of these compositions, integrating the auxiliary services that support the dynamic reconfiguration. Notice that there are two more additional channels — *Ctrt2CtxMonitor* and *Ctrt2AdptManager*. These are control channels that establish the communication the between the new services and the context monitor/adaptation manager.

As a proof of concept, we have implemented a version of all components of the architecture using *RAppia* [7], a service composition and execution framework. This framework was selected because it has a number of features particularly well suited for performing reconfiguration at runtime, and also because it was developed "in house". The reader should notice however that, although most of the content of this section is based on *RAppia*, similar implementations could have been derived for other frameworks such as Cactus [8], and Ensemble [9].

RAppia is a framework that promotes the modular implementation of service compositions. Services communicate using events. The event flow is associated with service stacks, which are vertical compositions of service instances and are associated with channels. Often, at the bottom of the stack, there is a service that interfaces the operating system, for instance, to save some information on a persistent store, or to send a message to other nodes.

A service stack may have elements that are shared with other stacks. This is useful, e.g., to synchronize events among channels. In the following sections, we describe in more detail the components we have developed using *RAppia*. They define a general infrastructure that is reusable across different applications.

4.1 Pluggable Components

As mentioned before, we have developed three auxiliary services that can be added to a service composition realizing mechanisms for context sensing and adaptation management. In particular, we have developed a buffering service, a context sensor service, and a local reconfiguration agent service. These services need to be added to the composition that

supports each (reconfigurable) channel. Note that the sensor service uses a control channel to communicate with the context monitor. Similarly, the local reconfiguration agent also uses a control channel to communicate with the adaptation manager.

Buffering Service. The purpose of the buffering service is to (temporarily) buffer the events produced by the application and sent to a channel composition that is being reconfigured. These events are delivered to the service composition as soon as the reconfiguration is concluded. In this way, the application is not required to stop operating while the reconfiguration proceeds. When the service composition is not being reconfigured, i.e., in a steady state, the buffering service only forwards the events from the application. Every service stack has an instance of the buffering service in the top. This instance is shared by all stacks belonging to the same application.

Sensor Service. Sensors work as mediators, collecting local context information and sending this information back to the central context monitor. The information collected by the sensor is described in the "*imported*" section of the context model. Quite often, this information is provided by the existing services themselves. In this case, this is captured in the service model (it consists of the queries of a service and also the traps generated by that service). The application itself is also a source of context information that can be captured by the sensor, according to what is described in the application model.

A sensor can be generic, or specific, depending on the context information that it targets. A generic sensor is one that is prepared to deal with all common information: queries and traps. In contrast, a specific sensor is one that depends on the service been monitored. In both cases, the instance of the sensor is placed just below the buffering service and is also shared with the other service stacks of the same application. In *RAppia*, we have implemented a generic sensor service. This service is ready to intercept all traps produced by other services in the same service stack and forward them to the context monitor. It can also be instructed by the context monitor to capture the value of queries of selected services and send them back. The sensor can be configured to obtain this information either on demand or periodically.

Reconfiguration Agent Service. The local reconfiguration agent service is responsible for executing local reconfiguration actions as instructed by the adaptation manager. It is a generic service that is prepared to interpret a pre-defined set of commands that are issued by the adaptation manager, such as putting a service in a quiescent state, capture the state, start/stop, or add/remove a service from a channel, among others. The exact sequence of commands that has to be executed is of the responsibility of the manager, and it makes part of a reconfiguration strategy.

4.2 Context Monitor

The main purpose of the context monitor is to collect, and process the information that is produced by the local sensors that run in the different nodes of the distributed application, and make the resulting context-information available to the adaptation manager.

As shown in Figure 2, the context monitor needs to be plugged to a context model defining the information that needs to be imported and exported and the relation between them. Although we envision the automatic generation of an implementation of the context model from its definition, currently this element has to be programmed explicitly and loaded into the context monitor as a plug-in.

Typically, most of the imported information is provided by the services in use at the service layer. In this case, the generic sensor may be used to collect this information. It is the responsibility of the context monitor to configure the local context sensors according to what is defined by the context model, namely configuring context sensors such that they intercept the relevant traps and query the relevant control variables. If specific sensors are used, they are implemented as services that are embedded in the service stack of the

control channel *Ctrl2CtxMonitor*, possibly shared with other stacks. Through this control channel, the context monitor gets sensed information from those sensors using the same kind of interface adopted in the generic sensor.

The values of the imported observables are stored in an internal database of the context monitor. Using the imported context information, the context monitor is responsible for producing the exported context observables and events. This task is carried by the implementation of the context model available.

4.3 Adaptation Manager

The adaptation manager is responsible for the adaptation policy evaluation and for coordinating global reconfigurations. The execution flow of the adaptation manager is the following. Unless stimulated, the manager remains in the idle state. The manager is activated when an event is triggered by the context monitor. This trigger the evaluation of the policy. As a result, two situations may occur: no reconfiguration is necessary, or a reconfiguration needs to be performed.

A reconfiguration is described as a set of primitive actions to be performed on the service compositions and a set of targets (nodes, channels, and individual services). The role of the adaptation manager is to select which are the required steps to execute these actions and which form of coordination is required among the nodes. The manager also has to decide if some preparatory actions are required to perform the reconfiguration, e.g., if a given service needs to be put in a quiescent state. The exact sequence of preparation, coordination, and reconfiguration actions is called a reconfiguration strategy.

Policy Evaluation Engine. The adaptation policy is compiled and stored, in coded form, in a policy database. When the context monitor triggers an exported event, all rules that compose the policy are evaluated to check which ones are fired by the event. For each fired rule, the associated condition is tested, by invoking a query operation on the context monitor, to check if the rule may be activated. Finally, all actions from activated rules are collected and passed to a component of the adaptation manager that chooses the strategy to be used.

Reconfiguration Strategies. When the adaptation manager needs to carry out a reconfiguration, it proceeds by sending commands to the reconfiguration agents at each involved node, following a specific strategy. These directives establish the coordination of participants in the communication through a particular *orchestration* and also define how the reconfiguration must proceed locally, at each node. The orchestration defines how the nodes coordinate to perform the distributed reconfiguration. Each orchestration requires the execution of a different number of communication steps among the adaptation manager and the local reconfiguration agents. The latter aspect, which we designated by *local reconfiguration mode*, is concerned with: (1) placing the local service 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. Similarly, there are different ways of proceeding with the local reconfiguration of a service composition, with different costs and applicability constraints. 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 that require a strong coordination of the participants of the distributed application, s.a. the exchange of the communication service, but can also be required in a situation without coordination needs, s.a. replacing the aggregation service in the messaging application. In this case, each affected node may reach the quiescent state, and perform the reconfiguration locally, without coordinating with the remaining nodes. In our work, we have identified a collection of useful strategies. Some

Strategy	Orchestration	Step	Local Reconfig. Mode
Flash	Uncoordinated	Step 1	Reconfigure
Local Quiescence	Uncoordinated	Step 1	Quiescence, Reconfigure
Stop-and-go	Coordinated	Step 1	Quiescence
		Step 2	[CaptureState,] Reconfigure [, LoadState]
		Step 3	Start [, Remove]

Table 1: Strategies

of them are depicted in Table 1 and shortly described below. A detailed description, and discussion of a complete list of strategies is outside the scope of this paper, and can be found elsewhere [10].

Flash This strategy is the most light-weight strategy one can devise. It requires no coordination among nodes and no local preparation to perform the reconfiguration actions. The strategy uses the *Uncoordinated Orchestration*. The *step1* message carries the **Reconfigure** command. Each agent executes the command immediately as soon as it receives the *step1* message and replies to the manager.

Local Quiescence This strategy is similar to the previous one, with the exception that forces a safe state in the target services. Thus, the *step1* message carries both a **Quiescence**, and **Reconfigure** commands. Each agent forces the quiescent state before reconfiguring, as soon as the message is received. In the end, it replies to the manager.

Stop-and-go This strategy can be used to reconfigure a service or to replace one implementation of a service by another implementation. The strategy uses *Coordination*. The *step1* message carries the **Stop** command. When all nodes have stopped the service, the *step2* message is sent. The reconfiguration is performed at this point. When changing an implementation, the state from the previous implementation is transferred to the new implementation. Finally, when these actions have been performed at all nodes, the service can be (re-)started. This is triggered by the *step3* message.

Strategies together with their cost define a order. For instance, for the strategies presented above we have: *flash* $<$ *local quiescence* $<$ *stop-and-go*. Choosing the strategy to achieve a specific reconfiguration action r, depends on the set of strategies that can be used with the services that are the target of r, and on the overall strategy cost. Each service defines, for each reconfiguration action, which strategies can be applied. By combining that information from all the target services of r, it is possible to discover the applicable strategies for r. The strategy with lower cost is chosen.

5 Messaging Application

To illustrate and validate our approach we have designed and implemented a simple adaptive messaging application. In the previous sections, we have already described the main functionality and components of this application. In this section we describe with more detail why and how the application can be adapted.

We recall that our messaging application relies on the following services: *ChatRoom* service, *Beacon* service, *Tracking* service, and *Aggregation* service. Furthermore, two different implementations of the *ChatRoom* service, and three implementations of the *Aggregation* service are available. The reader can refer to Section 2 for a brief description of these services.

Even with a small set of services, many different compositions may be instantiated. The role of adaptation consists in using the most efficient composition, satisfying the user

requirements, by taking into consideration the operational envelope. In the next paragraphs, we address the following issues in the design of our adaptive messaging application: the cost of executing each of these services, the policies that control adaptation, and the strategies used for reconfiguration.

5.1 Service Cost

When a service belongs to a composition, not only it has a cost for the benefits offered, but also it introduces a delay in the overall processing. Adaptation has to carefully balance the benefits, and costs of having a service, optimizing efficiency. In this example, several services are involved in the adaptation of the application. The *Aggregation* service, as described in Section 2, reduces the number of packets sent in the network at the expense of delaying their delivery. The *Beacon* service places an additional header, with the updated user location, in every text, and drawing messages exchanged by the application. Thus, it increases the message size, and slightly delays their handling. The *Tracking* service recovers location information from every received message. To do so, it has to parse messages to extract the location headers. This delays the handling, a cost that should be avoided unless locations are advertised and the user is interested in them. The *ChatRoom* service purpose it to abstract the rest of the application from the transport protocol used to support inter-process communication. There are two distinct implementations of the Chat Room. One implementation supports only two participants and relies on point-to-point channels (in practice, a TCP connection) to support the communication between the participants. The other implementation supports any number of participants and relies on group communication. The latter implementation offers a richer support, allowing membership management, and failure detection, at the expense of a higher delay in the handling of messages, as well as an increased number of execution cycles and resource consumption.

5.2 Policy

As noted before, the role of adaptation is to use the most efficient service composition at each moment. From the previous description it should be clear that the most efficient composition, from the resource consumption point of view, is a service composition using the point-to-point implementation of the *ChatRoom*, the *TextAggregation* service to reduce network traffic, and no *Beacon* or *Tracking* active. The policy will attempt to use this configuration as long as the user preferences and the operational conditions allow. There are two types of causes that may trigger adaptation: changes in the user's preferences and changes in the operational envelope. We describe each of these changes below.

Changes in the User's Preferences. The user can specify the desired energy saving tradeoff and is also responsible to state if its location information should remain private or if she is interested in received the location of the remaining participants. Each of these preferences may force a given service to be added or removed from the service configuration.

When the location is public, the beacon service needs to be added to the service composition that supports the location channel. When the location is private, this service is no longer necessary nor desirable. The use of the *Tracking* service depends on a mix of local and global information. To start with, there is no need to waste resource with the this service if the local user is not interested in receiving location updates. However, even if the user has expressed interest in such updates, it is only worth to activate this service if there exists at least one participant that has made its location public. Unless both conditions are met, the *Tracking* service remains excluded from the service composition.

Regarding energy saving, the user can opt between higher responsiveness (where more energy is used) or a power-save option. These two options, respectively, either removes, or adds the *Aggregation* service to the composition. This service allows to decrease the number

of packets sent to the network, thus reducing the resource consumption, used bandwidth, and energy.

Changes in the Operational Envelope. Our example also illustrates adaptations that are not triggered by user's preferences, but triggered by changes in the environment, namely changes in the number of users of the application. Given that a simple point-to-point connection is less resource consuming that a group communication stack, the point-to-point implementation of the *ChatRoom* service is used as long as there are only two users in the application. When a third user connects, the system automatically adapts to use group communication. Furthermore, different implementations of the *Aggregation* service are available, as described in Section 2. The choice of the most appropriate implementation also depends on the operational envelope (available bandwidth, usage pattern, etc). Due to lack of space we do not discuss further these alternatives.

5.3 Strategies

The policies described above require the following reconfigurations: adding, or removing the *Beacon*, *Tracking*, or *Aggregation* services; replacing the point-to-point *ChatRooom* implementation by the group communication implementation, and vice-versa. We now discuss the coordination and preparation requirements for each of these reconfigurations and, in consequence, which strategy needs to be applied in each case.

The *Beacon* service addition or removal is a reconfiguration action that can be performed locally without coordination with the remaining nodes. Furthermore, the *Beacon* service is a stateless service, so there is no need to force it to reach a quiescent state before removal. Thus the *flash* strategy, which is the cheapest of all strategies, can be applied to perform both the addition and removal of this service. The same reasoning applies to the *Tracking* service.

The *Aggregation* service addition or removal is also a local decision, that requires no coordination with other nodes. However, contrary to the *Beacon* and *Tracking* services, the *Aggregation* service is state-full: it collects different messages produced by the application to merge them in a single message. Thus, before removing the *Aggregation* service it is necessary to force it to reach a quiescent state, by flushing its internal queue of messages. Therefore, the *local quiescence* strategy needs to be used.

Finally, replacing the *ChatRoom* is a reconfiguration that needs to be globally coordinated, given that two nodes cannot communicate if one is using the point-to-point implementation and the other the group communication implementation. Furthermore, before removing any of these implementation, they need to be put in a quiescent state to ensure that messages in transit are not lost. Finally, their state needs to be capture and reloaded, such that the identity of the active peers is not lost during the reconfiguration. For these reasons, the expensive *stop-and-go* strategy is used.

To illustrate the importance of using the weaker strategy, we depict in Table 2 the time values it takes to execute the different strategies in our prototype. All values were measured using a network of either 2, or 3 participants (depending on the reconfiguration), executing in Pentium IV / 2.8GHz machines with 1 Gb of memory. The machines are connected through a 100 Mbps Ethernet switch.

As depicted in Table 2, the *stop-and-go* strategy (two synchronization steps), through the exchange of *ChatRoom* services, is much slower that the remaining strategies. In a messaging application, the stop-and-go strategy reconfiguration time is not a limiting factor, given the relatively low response time of the user (evidence collected for massive on-line multi-player games show that even higher latencies are tolerated[11]). However, in many other cases, faster reconfiguration times are required. Due to this reason we are currently researching different reconfiguration strategies.

	Beacon	**Tracking**	**Aggregation**	**ChatRoom**
Add Time (ms)	31,1	31,3	31,1	n.a.
Remove Time (ms)	28,6	28,5	30,3	n.a.
Change Time (ms)	n.a.	n.a.	36,4	253,3

Table 2: Reconfigurations

6 Related work

Separation of computation from adaptation concerns is common on component-based architectures [12, 13] as well as in service composition frameworks [8, 7]. This approach has proved to improve flexibility and maintainability and, hence, it has been applied in several areas [14, 15]. In component-based architectures, adaptation is typically achieved through the addition, removal, and exchange of system components or the interactions between those components, while in composition frameworks, mainly dedicated to the composition of network level protocols, adaptation is achieved typically through the exchange of algorithms and the fine-tuning of protocol parameters. The work described in this paper is a tentative of combining the two types of approaches. The proposed approach targets services in general (i.e., network services but also application-specific services), offering not only fine-tuning of services, but also the support for the addition, removal and exchange of services at runtime. Support for adaptation has been addressed in the context of several frameworks, namely Ensemble [16] and Cactus [17]. Each framework offers a different approach. Ensemble is a protocol composition framework, that relies in vertical protocol stacks to offer a service. Runtime reconfiguration is achieved by switching algorithms. The switch relies in a coordinator-based orchestration, and a stop-and-go local reconfiguration mode, thus using a single strategy for switching protocols. Cactus is a service composition framework, whose dynamic reconfiguration relies in switching micro-protocols (whose composition results in a service). Moreover, reconfiguration can also be achieved by parameters tuning. The framework offers monitoring, and agreement features to support automatic dynamic reconfiguration. Since the system's global reconfiguration is expected, Cactus offers a single reconfiguration strategy based on inter-host global orchestration, and non-stop local reconfiguration mode. Our approach targets all types of services while the described frameworks target only network services. Moreover, in our case, several adaptation strategies can be used, new ones developed, thus optimizing the performance of the reconfiguration process. The other frameworks lack any support for new strategies and do not offer a diverse set of strategies. Complementary approaches to self-adaptation include several architecture-based adaptation frameworks, most notably the Rainbow framework [18]. This framework, which incorporates mechanisms to monitor and adapt systems to surrounding changes, mainly aims at providing a reusable infrastructure, with specialization mechanisms to fill in any particular needs. Being an approach that aims to have a broaden applicability, the set of tailorable parts that need to be customized or even developed from scratch still requires some effort. In contrast, our approach relies in a more tight architecture, with a strong structure, but demanding less effort to address specific application's needs.

7 Conclusions and Future Work

This paper presents a middleware framework that supports the definition, implementation, and execution of reconfigurable service compositions. These compositions are adapted by adding, removing, or exchanging services, and also by tuning the service's parameters. Adaptation is specified through high level policies, and conducted based on generic adaptation strategies, applicable to any adaptation whose requirements are fulfilled. Furthermore, we

put forward an approach to the construction of adaptive distributed applications in this framework. The approach relies in a set of generic pluggable components that can be added to a service composition, to support adaptation in a transparent manner to the application.

There are two directions in which we are currently extending our work. From the model perspective, we would like to enrich our models with additional information, in order to increase the amount of reconfiguration code that can be automatically generated from these models. From the runtime support perspective, we are currently building a library with a broader set of strategies, such that reconfiguration can be performed with minimal interference in the regular operation of the distributed application.

Acknowledgments

This work was partially funded by FCT project MICAS (POSI/EIA/60692/2004) through POSI and FEDER.

References

[1] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan (2004)

[2] McCarthy, D., Dayal, U.: The architecture of an active database management system. In: SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (1989) 215–224

[3] Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Technical report, Dartmouth College, Hanover, NH, USA (2000)

[4] Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. In: POLICY '01: Proceedings of the International Workshop on Policies for Distributed Systems and Networks, Springer-Verlag (2001)

[5] Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: WOSS '02: Proceedings of the first workshop on Self-healing systems, New York, NY, USA, ACM Press (2002) 33–38

[6] 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

[7] Rosa, L., Rodrigues, L., Lopes, A.: Appia to R-Appia: Refactoring a protocol composition framework for dynamic reconfiguration. DI/FCUL TR 07–4, Department of Informatics, University of Lisbon (2007)

[8] Hiltunen, M.A., Schlichting, R.D., Ugarte, C.A., Wong, G.T.: Survivability through customization and adaptability: The cactus approach. discex **01** (2000) 0294

[9] 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

[10] Rosa, L., Lopes, A., Rodrigues, L.: A framework to support multiple reconfiguration strategies. In: Autonomics'07: Proceedings of the International Conference on

Autonomic Computing and Communication Systems, Washington, DC, USA, IEEE Computer Society (2007) to appear

[11] Sheldon, N., Girard, E., Borg, S., Claypool, M., Agu, E.: The effect of latency on user performance in warcraft iii. In: NetGames'03: Proceedings of the 2nd workshop on network and system support for games, New York, NY, USA, ACM Press (2003) 3–14

[12] Batista, T., Rodriguez, N.: Dynamic reconfiguration of component-based applications. In: PDSE '00: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, Washington, DC, USA, IEEE Computer Society (2000) 32–40

[13] 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

[14] Kramer, J., Magee, J.: Analysing dynamic change in software architectures: A case study. In: CDS '98: Proceedings of the International Conference on Configurable Distributed Systems, Washington, DC, USA, IEEE Computer Society (1998) 91

[15] Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Communications ACM **45**(6) (2002) 33–38

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

[17] 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

[18] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer **37**(10) (2004) 46–54