A Framework to Support Multiple Reconfiguration Strategies

Liliana Rosa
University of Lisbon
lrosa@lasige.di.fc.ul.pt

Luís Rodrigues
IST/INESC-ID
ler@ist.utl.pt

Antónia Lopes
University of Lisbon
mal@di.fc.ul.pt

ABSTRACT

Self-management is a key feature of autonomic systems. This often demands the dynamic reconfiguration of a distributed application. An important issue in the reconfiguration process is the strategy that is used to coordinate the multiple participants involved in the reconfiguration. This paper addresses the problem of providing support for multiple reconfiguration strategies in autonomic systems that are designed as self-reconfigurable service compositions. We decompose existing strategies in two separate aspects — an orchestration protocol and a local reconfiguration procedure. This separation allowed us to design a set of generic pluggable components that can be integrated in concrete service compositions, in order to support different strategies. The strategy selection is performed according to the semantics of each reconfiguration. To illustrate our approach, we have implemented an instance of these pluggable components for the *RAppia* composition framework.

Categories and Subject Descriptors

D.2.m [**Software Engineering**]: Miscellaneous

General Terms

Management, Dynamic Reconfiguration, Adaptation Strategy

1. INTRODUCTION

Modular applications can be built from the composition of many fine-grain services. This approach not only allows the reuse of individual components in different applications but also facilitates application adaptation to dynamic changes in the user requirements, or in its operational envelope. The support for dynamic adaptation is a key requirement to build flexible autonomic systems, able to self-manage via runtime reconfiguration.

The need for reconfiguration may emerge from different sources, such as the user preferences or the execution context. From this results that some aspects of reconfiguration are related to the user, while others are tied to the execution context, among others. Therefore, it is important to have a separation between adaptation and functional concerns [10] based on monitored changes in user preferences, execution context, hardware, etc. Thus, monitoring and adaptation can be greatly simplified if the issue of supporting multiple reconfigurations is addressed at the middleware level.

In a distributed application, the dynamic reconfiguration of participants may require coordination. For instance, if an application reconfigures itself to use ciphered communication instead of plain text communication, all participants must instantiate the required cipher components. We denote by *reconfiguration strategy* the set of coordination rounds and the local reconfiguration procedure required to perform a reconfiguration.

Reconfiguration strategies play a key role in the development of self-reconfigurable systems. Unfortunately, they may be an important source of overhead in the system operation. Also, the amount of coordination required highly depends on the type of reconfiguration being performed. For instance, in the previous example, one may be required to stop the plain text traffic flow and install new communication components before restarting the (now ciphered) communication. On the other hand, other classes of reconfigurations may be less demanding in terms of coordination. For instance, the activation of an auditing component that intercepts and logs certain interactions is a reconfiguration procedure that can be applied locally to each node, without coordination with the remaining nodes. In fact, many reconfiguration strategies have been proposed in different contexts [9, 23, 3, 21, 19].

We are interested in creating a framework that allows the developer of an autonomic system to use predefined generic components to instantiate the most appropriate strategy to each particular reconfiguration, without having to (re)implement a customized solution for each case. With this goal in mind, the paper makes the following contributions:

i) It proposes the decomposition of existing reconfiguration strategies into two separate aspects — an orchestration protocol and a local reconfiguration procedure. Furthermore, it identifies a number of relevant patterns for each of these aspects that can be combined to obtain multiple strategies.

ii) Based on the decomposition above, it identifies a set of generic pluggable components that support the execution of the identified strategies.

iii) It describes the implementation of pluggable com-

AUTONOMICS 2007, 28-30 October 2007, Rome, Italy
Copyright © 2007 ICST 978-963-9799-09-7
DOI 10.4108/ICST.AUTONOMICS2007.2113

ponents in a concrete composition framework, namely the *RAppia* system.

The rest of the paper is structured as follows. Section 2 describes the generic architecture in the context of which our work fits. Section 3 presents our proposal of defining strategies in terms of orchestration protocols and local reconfiguration procedures, and also identifies several reconfiguration strategies defined in this manner. Section 4 proposes a number of generic pluggable components that can be embedded in the composition framework to support different strategies. Moreover, we describe how these components have been implemented in the *RAppia* framework. Related work is addressed in Section 5 and the paper is concluded in Section 6.

2. SYSTEM MODEL

Reconfiguration strategies are typically part of a broader architecture to support autonomic behavior. In this section, we briefly overview the system model where our work fits.

As illustrated in Figure 1, we assume that the system is composed by a set of nodes that participate in a distributed multi-party application. The application itself is obtained by the composition of multiple services, more concretely, the application is constructed by using one or more stacks of services. Some of these services are responsible for the coordination and communication aspects of a distributed application (for instance, agreement and mutual exclusion), while other services are responsible for interface aspects, etc. We assume that the application autonomic behavior is achieved through the dynamic adaptation of its stacks of services.

An application can be adapted through the combined execution of different kinds of actions: service parameter adaptation; exchange implementation of one, or more services in place; and change the service stack configuration.

The reconfiguration process is controlled by an external *adaptation manager*. The manager is a generic component with an application-specific adaptation *policy* [17], and a set of *reconfiguration strategies*. The manager controls the *reconfiguration agents*, present in each node. These agents are responsible to apply locally reconfiguration actions, executing several commands received from the manager, such as adding services, among many others.

In each node there are also *context sensors*. These sensors acquire any relevant context information, such as user preferences or available bandwidth [2]. Sensors can be generic

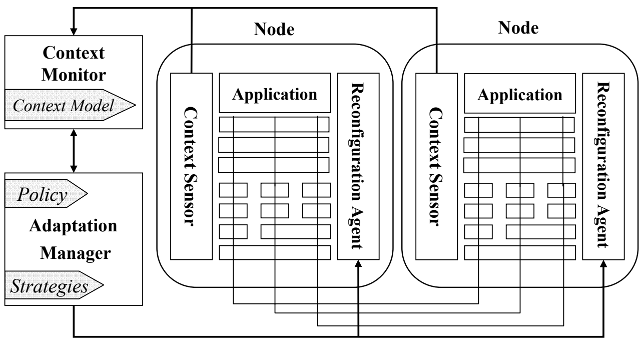

Figure 1: System Model

or specific. The former can capture information made available by any of the services in the service stacks, while the later is oriented to specific information, which may require specific processing. The sensed information is delivered to a *context monitor* that manages and interprets it in terms of high-level concepts. The processed information is made available to the adaptation manager.

When the adaptation manager receives relevant context information, it starts the policy evaluation. This evaluation determines if any reconfiguration is required. If so, the manager plays an algorithm to choose the appropriated *reconfiguration strategy*. The strategy dictates mainly the coordination and local reconfiguration concerns to be applied. Based on that, the manager issues several commands to the reconfiguration agents of the affected nodes. This paper focus on a set of generic reconfiguration strategies, and in the architecture that allows the most adequate strategy to be applied in each case.

3. RECONFIGURATION STRATEGIES

Reconfiguration strategies define how a reconfiguration action is applied through nodes and how reconfiguration is locally achieved in each node. In the context of the system model presented in Section 2, a reconfiguration strategy can be defined in terms of an *orchestration* protocol and a *local reconfiguration* technique. The first defines coordination among nodes, while the second describes how reconfiguration is applied locally. The combination of different orchestrations and local reconfigurations yields a large set of viable strategies, that can be applied in several distinct contexts.

Different reconfiguration actions typically have different coordination requirements. For instance, whereas the reserved memory space for a caching service can be changed in multiple nodes without requiring inter-node coordination, the reconfiguration of the communication services used by the nodes (by replacing TCP by UDP, for instance), needs to be performed in a coordinated fashion, as communication would be impossible if each node is using a different communication service.

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

Note also that the local reconfiguration technique is, to some extent, independent of coordination adopted. The need of reaching a quiescent state is also independent of coordination requirements. Quiescence is often required in situations that demand a strong coordination between participants of the distributed application (s.a. the exchange of the communication service), but can also be required in a situation that does not need such coordination. For instance, when a service implementation service is replaced by another compatible implementation (e.g. to install a bug-fix on-the-fly), it may not be necessary to coordinate the reconfiguration in all nodes. However, in each node, it may be necessary to put the service in a quiescent state to replace it.

Therefore, we propose to define a reconfiguration strategy

k	Orchestration
0	Uncoordinated
1	Single Sync
2	Double Sync

Table 1: Orchestrations

in terms of different combinations of distinct orchestrations and local reconfiguration techniques, as described in detail in the next paragraphs.

3.1 Orchestrations

An orchestration defines how nodes coordinate to perform the distributed reconfiguration. Each orchestration depends on the exchange of messages between the adaptation manager and the local reconfiguration agents. The exchange of messages allows the manager to communicate which reconfiguration needs to be performed and how, including if coordination is required. The coordination between the nodes involved in the reconfiguration works very much as a synchronization barrier, where none of the agents proceeds with further reconfiguration steps until every other agent is ready. An orchestration can have zero, one, or multiple synchronization points. A *k-orchestration* will enclose k synchronization points. Each synchronization point is preceded by an exchange of messages between the manager and involved agents. Each exchange consists of the transmission of a *start-round(k)* message by the manager to all agents and the collection of *round-done(k)* messages sent back from the agents to the manager.

From our experience, we have only identified three meaningful orchestrations, with $k = 0$, $k = 1$, and $k = 2$. Although so far we did not find any practical example of *k-orchestrations* with values of k higher than 2, this does not mean that they do not exist, thus we kept the generic formulation. For easier reference, we have named the relevant *k-orchestrations* as depicted in Table 1. Different sequences of local reconfiguration steps can be performed at each round of the these orchestrations, thus resulting in different reconfiguration strategies.

3.2 Local Reconfiguration Techniques

A local reconfiguration technique is concerned with the manner the reconfiguration is performed locally, namely the local actions that have to be performed to achieve the envisaged reconfiguration of the local services stacks.

There are several approaches to local reconfiguration, each offering different trade-offs. These trade-offs are related with the main issues a reconfiguration presents, such as delays, or resource consumption. The delays refer to message flow interruption, due to changes in the services composition. The delay caused by this interruption depends on the target services, which may have to stop processing messages. Some techniques cause a higher delay. For instance, this is the case of the approach adopted in [23] which implies the interruption of the execution, at the expense of a buffer that accumulates any messages produced by the application. Other techniques aim at reducing the delay to a minimum, at the expense of elaborated methods depending on specific components. These components can be switches [3, 16] or multiplexers [9]. However, all these solutions are service-specific, only applicable to the exchange of services, and just one service at a time. The delay introduced is related either with the switching, or the specific component addition/removal. The current state of the art does not offer strategies that do not introduce delays, since some sort of interruption always exist, either as a result of the addition of components, switching, or forwarding issues.

In this paper, we describe three local reconfiguration techniques: *Direct*, *Stop-and-go*, and *Prepare-and-go*. These techniques cover the three essential manners to perform a local reconfiguration in the composition of services. However, many variations and optimizations can be derived in several techniques that are out of the scope of this paper.

The *Direct* technique can be applied to all reconfiguration actions. A value can be set, services added, removed, or exchanged directly. However, no guarantee is given about time of occurrence, neither regarding the message flow. On the other hand, the *Stop-and-go* approach forces a quiescent state in all the services on the affected service stack, assuring that reconfiguration occurs without hanging messages and that services will not produce new messages while reconfiguration is taking place. Moreover, the reconfiguration start and end are controlled by the agent. As the *Direct* technique, it can be applied to all reconfigurations, but introducing a much higher delay. The *Prepare-and-go* technique also offers control on reconfiguration occurrence, and assures that no messages are lost or mis-handled. However, when compared to the latter, the introduced delay is much smaller, at the expense of extending the reconfiguration period, and using an auxiliary composition of services. Nevertheless, this technique only applies to the exchange of services. The three techniques are described in detail:

Direct: This technique is applied when a directive is received from the manager, that includes all the required information. The reconfiguration is executed promptly, without any preparation. The reconfiguration agent waits until the affected services finish handling the current messages to remove or exchange them. The adding of a service or tunning of a parameter are executed immediately.

Stop-and-go: This technique requires a preparation phase, in which all services from the composition are forced to achieve a quiescent state, and stopped. Only after that, the reconfiguration is applied. At the end, the services are awakened. The new services start and the remaining ones resume the execution. During the stop and start no messages are handled and the execution is frozen.

Prepare-and-go: This technique also requires a preparation phase. The preparation relies on adding special components, and creating an auxiliary channel with the new services. Both old and new services are maintained, and the special components are responsible for message forwarding between the two compositions. When it is safe to remove the old services, only the new ones are kept, which are added to the original composition. Afterwards, both the special components, and the auxiliary composition are removed. To add and remove the special components, the *Direct* technique is applied. Thus, the delay depends on that addition and removal.

A local reconfiguration technique may have further concerns related with the services that are being exchanged. Among these concerns, there is the need of management of the information that must survive reconfiguration. For instance, when replacing a service implementation, the service may have important information, such as state information, as the identification of peers, or service static information, as a timeout value. However, handling those concerns is only possible when a technique assumes a preparation phase, previous to reconfiguration, what does not happen in the *Direct* technique. Thus, the handling of service information is only suitable to *Stop-and-go* and *Prepare-and-go* techniques, and relies in the following directives:

StoreInfo This command captures any relevant information from the service, that is not related with the service state, and also loads it.

StoreState This command captures the internal state of a service and also loads it. This requires that the service achieves a quiescent state previously.

The following section describes how these directives and techniques, combined with different orchestrations, give rise to distinct strategies.

3.3 Strategies

As previously said, each strategy is a combination of an orchestration and a local reconfiguration technique. The several orchestrations, combined with the different techniques plus information handling commands allows the definition of many strategies. In this section we only described the strategies we found more relevant, without discussing their many variations. The strategies are identified as relevant either from the literature analysis, or from our own experience in developing adaptive systems. These strategies, summarized in Table 2, are the following:

3.3.1 Flash

This is the most lightweight strategy. Every node is reconfigured without concerns about other nodes, so no coordination is used. Furthermore, service information is not handled, thus it is neither captured, nor loaded. Although few assurances are given by this strategy, it can be used to apply any reconfiguration action, at its own risk.

Protocol: The strategy uses the *Uncoordinated* orchestration. Each reconfiguration agent executes the command immediately as soon as it receives the *start-round(0)* message. The reconfiguration success is signaled with a *round-done(0)* message.

Delay cost: It is the time of performing the reconfiguration action. If x is the duration of the reconfiguration, the cost is x.

Examples: Many services have configurable parameters that can be changed without any global or local synchronization. For instance, in a service that caches data items, the amount of space reserved for caching may be changed locally, independently at each node, without stopping the service. Other examples include changing timeout values, and adding/removing logging services.

3.3.2 Interrupt

The purpose of this strategy is to reconfigure a service in a globally safe state. For this purpose, the service is stopped at all involved nodes before the reconfiguration takes place. When the services have been stopped at every node, reconfiguration is performed. Finally, the services are only restarted when all nodes have finished the local reconfiguration actions. This strategy ensures that the reconfiguration will occur at the same time in all nodes, and that no messages generated previously to the reconfiguration will be handled after. All reconfiguration actions can be applied using this strategy.

Protocol: The strategy uses the *Double Sync* orchestration. The *start-round(0)* message indicates which preparatory actions need to be performed by each reconfiguration agent, such as the *Capture* command, if a service is going to be replaced by another service. Each agent starts the preparation immediately by first creating the new services (if any are to be created). Afterwards, it activates the buffering, and forces quiescence. After quiescence is obtained the service information can be captured. When all nodes have completed these tasks, they reply to the manager with a *round-done(0)* message. After the first synchronization point, the *start-round(1)* message is sent, thriggering the reconfiguration. After reconfiguration, if is an exchange of service's implementations, the previously captured information is loaded in the new implementation. The reconfiguration completion is signaled by sending the *round-done(1)* message to the manager. Finally, when all nodes have performed the reconfiguration, the services can be (re-)started by sending a *start-round(2)* message to all affected nodes. The success is signaled with the final *round-done(2)* message.

Delay cost: It is the time when the quiescence is forced until the service composition restart. If x is time to perform the reconfiguration (adding/removing/exchanging/setting), y is the time required to put the composition on a quiescent state, z is the time necessary to capture, or load information, and w is the time taken until completion, i.e., between *round-done(0)* and *start-round(1)* or between *round-done(1)* and *start-round(2)*. The total cost is $x + y + 2w[+2z]$.

Examples: Many services have to be added/removed/exchanged in a coordinated way, otherwise messages will be lost, or mishandled. For example, this is the case of the addition or removal of group communication services from service compositions. Messages that are generated during the use of such a service have to be delivered to it, otherwise they will be inappropriately handled. Other examples include reconfiguration actions affecting ordering, and cyphering services.

3.3.3 Non-Interrupt

The purpose of this strategy is to support services exchanges in several nodes without stopping any service execution. For this purpose, two instances of a multiplexer service are added to the service composition and an auxiliary channel is created with the new services (see Figure 2). In this way it is possible to switch from old services implementation to the new one, without disturbing all the services in the composition. During reconfiguration, events are forwarded either through the main or auxiliary channels, according to a forwarding algorithm. This ensures that services message processing is not interrupted during the reconfiguration. The end of the local reconfiguration process is detected by the multiplexer service, which signals it to the reconfigurator.

Protocol: The strategy uses the *Single Sync* orchestration. The *start-round(0)* message carries all the reconfiguration information, including which service-state needs to be captured. The reconfigurator acts promptly when receives the message and adds the special component using the *Direct* technique. If any service state is required, it is retrieved at this point. Afterwards, the reconfigurator creates the

Strategy	Orchestration	Preparation/Recovery	L. R. T.
Flash	Uncoordinated	-	Direct
Interrupt	Double Sync	[StoreState/Info]	Stop-and-go
Non-Interrupt	Single Sync	[StoreInfo]	Prepare-and-go

Table 2: Strategies

auxiliary channel with the new service implementations. If there was any information captured, it is loaded. At the completion of these tasks, the reconfigurator signals end of preparation by sending a *round-done(0)* message and continues message processing. When all nodes have finished the preparation, the manager issues a *start-round(1)* message. At the arrival of this message, the reconfiguration agent activates service tags on events, and the special component to forward messages. Only when the special component detects that old services can be removed, it signals the agent, which removes the old services and sends a *round-done(1)* message to the manager.

Delay cost: It is the time of adding the special components, and removing them, at the end. The reconfiguration itself can take as many time as necessary, but it does not affect the final delay. If k is the time to add/remove the component, total cost is $2k$.

Examples: Some reconfigurations require coordination but the longer the execution is interrupted, the worst. This is the case of the exchange of total order [3, 16], that cannot afford to interrupt execution for a comparably longer period of time. Other examples include highly dependable services with real-time constraints, such as a timely failure detector.

3.4 Choosing a Reconfiguration Strategy

Given that we have several reconfiguration strategies available, it is necessary to have mechanisms to choose the most suited to each potential reconfiguration situation on the system under development. First of all, the strategy must be applicable to the specific reconfiguration action and target services. Moreover, it must also be cost efficient. The strategy selection process starts with searching the applicable set of strategies. Each service defines a binary table that describes which strategies can be used for a particular reconfiguration action. This table does not describe which are the best suited strategies, but all the strategies that can be applied, even if they offer better guaranties than necessary. By crossing this action-strategy information from all the reconfiguration target services, the applicable strategies set is built.

The next step in strategy selection, relies in a strategy ordering. This ordering can be cost oriented, based in resource consumption, or introduced delay, among others. Based on this ordering, the lowest cost strategy is chosen to be applied to the reconfiguration action. For example, we can order the strategies described in this section, based on the delay introduced, as follows: *Flash* < *Non-Interrupt* < *Interrupt*. Applying the selected strategy is achieved by using a set of pluggable components that are described in the next section.

4. GENERIC PLUGGABLE COMPONENTS

The decomposition of reconfiguration strategies in an orchestration and local reconfiguration components, as well as the definition of protocols for adaptation control between the adaptation manager and the adaptation agents, opens the door to develop generic pluggable components that can be added to a service composition to support multiple strategies in the same framework. In this section, we present such a set of generic pluggable components that can be added to an application built as a composition of services in order to facilitate autonomous behavior. These components are either permanent or temporary. The permanent components are always present in the composition, providing support for the adaptation of the application services in the stack. These components are the following:

- *Context sensors* able to extract from each node the information required to trigger reconfiguration policies.
- A *buffering service* that hides from selected application services (for instance, the user interface) the fact that some of the lower level services are being reconfigured.
- A channel *reconfigurator service* able to implement the protocols defined in Section 3.1 and used by the different orchestrations to invoke services required by the local reconfiguration techniques.

The temporary components are specific to a local reconfiguration technique, being added before the reconfiguration action, and removed at its completion. These components are called *multiplexers*. A *multiplexer* service is one of the special components described in Section 3.2, used in the *Prepare-and-go* technique. Figure 2 illustrates how these services are organized in the stack. As a proof of concept, these components have been implemented using a particular service composition framework: *RAppia* [18]. Both the main features of this framework and the detailed description of the developed pluggable components are reported in the following sections.

4.1 The *RAppia* Framework

We have selected the *RAppia* framework to build the proof of concept of our architecture because it has been developed “in house”, but also because it has a number of features particularly well suited for our goals. First of all, it promotes the design of applications as compositions of independent services. Being defined as compositions, they can be adapted through the reconfiguration of this composition, namely through the addition, removal, or exchange of services being used. Secondly, it supports channel configuration and activation via XML specifications. This eases the adaptation manager design, as new configurations can be sent to the reconfiguration agents using XML strings. Also, it includes a library of multiple services, which allowed us to build early prototypes for adaptation with minimal coding effort. Services have been developed to build multiuser object-oriented environments [22], distributed real-time games [13], collaborative mobile applications [12], and database replication application [15]. Lastly, but not less important, it supplies the necessary runtime reconfiguration support, by adding or removing services to a stack of services.

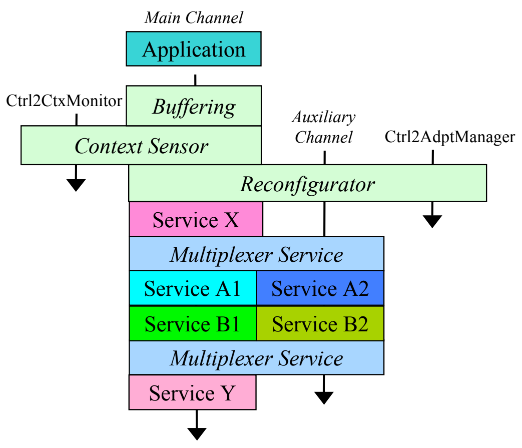

Figure 2: Pluggable Components

Note that it is not the purpose of our paper to discuss the merits and weaknesses of the *RAppia* platform. Therefore, the framework description only provides enough detail to allow the reader to understand how the pluggable components that support multiple strategies can, in fact, be built in a generic manner, using *RAppia* as a particular instantiation. We believe that similar implementations could have been developed for other frameworks.

RAppia is a framework that supports the implementation and execution of modular applications built as compositions of independent services. A channel is an instantiation of the services of a composition. The same service can be shared by several channels, thus being the same instance. The channel abstraction allows to hide any details regarding the compositions configuration, its change, or the reconfiguration process. The services communicate through events that can flow either in *up* or *down* direction in the channel. Each service describes which events are produced, accepted, or required to its execution. From this results that an event does not have to be subscribed by all the services in a composition. Thus, each event has a route and is only delivered to the services that belong to it.

This framework enforces a methodology for programming application services. Each service is coded as a set of event handlers. An event handler receives an event, adds/removes some header to/from the message associated with the event, and forwards the event in the same channel. Note that some services may temporarily store events in the session state, for instance, to forward in the correct order messages that are received out of order, or to assemble a message from different segments. Furthermore, our reconfiguration schemes require the service implementation to support the following functionalities:

i) Provide the context information that might be relevant to express adaptation policies;

ii) Accept events that allow the relevant configuration parameters to be changed during runtime, for instance, by allowing timeout values to be adjusted dynamically;

iii) Have the ability to achieve a quiescent state when requested;

iv) Have the ability to store and load the service state across different “incarnations” of a service instance, when a channel is reconfigured.

4.2 Buffering Service

The buffering service purpose is to hide the reconfiguration of services from the application, i.e. the non adaptable application part. In particular, it hides the fact that the communication needs to be temporarily interrupted to reconfigure a channel, by delaying any events. This is achieved by having the buffering service to store all messages produced while the channel is being reconfigured. The buffering service is added to all channels that are used by the top application service and is controlled by the reconfigurator service.

We have implemented a generic buffering service that can be used whenever reconfiguration has to be hidden from the application. This buffering service is positioned right under the application and either forwards messages, if inactive, or delays them for the necessary time. This service is vital for any major interruption, as happens in the *Stop-and-go* technique.

4.3 Sensor Service

This service captures context information locally at each node, and communicates it through a dedicated channel to the context monitor. Sensors can capture information from services in the composition by a request-answer approach to the services; accept any asynchronous events launch by services, or by the top application service (for example, denoting user preferences); or capture specific information. The first two cases can be handled by a generic sensor usable in any scenario, while the last demands a specialized sensor.

The generic sensor service is shared by all monitored channels. The sensor issues a request for an information and expects an answer with the updated value. This information can also be captured in a periodic manner. The sensor can capture any information, as long as it is configured to do so, or receives a query from the context monitor. Moreover, the sensor also receives any traps generated either by services, or the application, forwarding it to the context monitor.

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

To build the described generic sensor, services must be ready to process the request event *GetValue*, which indicates the information name, and reply with a *ReplyValue* event containing the information. Moreover, the service has to be prepared to generate a *TrapIndication* event asynchronously, which notifies a change in a specific condition. All services have to declare the information that can be read, and the notifications they are able to generate.

4.4 Multiplexer Service

This service is added to a composition of services in pairs. They work as a container for the old and new services, thus, shared by the main target channel, and the auxiliary channel created for reconfiguration purposes. The disposition of this service is depicted in Figure 2. The top multiplexer service intercepts events originated or forwarded by the top services, while the lower multiplexer from lower services. When an event is intercepted, the multiplexer decides to deliver it either to old or new services, forwarding it to the original, or auxiliary channel. This decision is based on event information, namely the event origin. The multiplexer depends on the reconfigurator to activate tagging in both old and new services, so that this information is available. It is also the multiplexer service that decides when the reconfiguration is over. This depends on the target services, and information collected by the multiplexer service. When the decision is taken, the reconfigurator service is signaled.

The multiplexer service can be used with different target services, consisting in a generic feature, with service-specific configurable features. The multiplexer generic part is related to forwarding support and event subscription. The configurable part regards the forwarding and the end of the reconfiguration detection algorithm (which varies from service to service). The forwarding approach is reusable and determines how an event is redirected between the channels. For example, in [16], the event is sent to both old and new services (both channels); while on [9], an event handled by old services is delivered to the original channel, while by new services, to the auxiliary channel. The *end algorithm* determines when it is safe to remove the old services.

4.5 Reconfigurator Service

The *Reconfigurator Service* is responsible for the dynamic reconfiguration of the local service composition. This service is shared by all channels, and communicates with the adaptation manager through a dedicated channel. The service answers to commands received from the adaptation manager. The commands available to be sent, in the different rounds that make orchestrations, are the following:

- StoreState/Info: this command determines the capture of information, state or non-state, and the loading in the end of the reconfiguration;
- Direct: this command dictates the execution of the *Direct* local reconfiguration technique;
- Prepare-and-go: this command indicates to start the *Prepare-and-go* technique;
- Stop-and-go: this command indicates to start the *Stop-and-go* technique;
- Go: this command indicates the start of the reconfiguration;
- Resume: this command indicates the finalization of the reconfiguration process, either by (re-)starting service execution, or by removing unnecessary services.

One or several commands can be issued in a single message. The reconfigurator service has knowledge on how to perform the local reconfiguration techniques, capture/load information, resume, and when to reply to the adaptation manager. This reduces the exchange of control information, and speeds the reconfiguration process. The commands used in the different steps by each strategy are listed as follows:

Direct The *start-round(0)* message contains the *Direct* command.

Non-Interrupt The *start-round(0)* message includes the *Prepare-and-go* command, as well as a *StoreInfo* command. The reconfigurator proceeds promptly by first adding the multiplexer service, and creating the auxiliary channel. After, it triggers a *GetInfo* event that crosses the channel in the up direction, from the bottom most service to the reconfigurator. The *SetInfo* event is forwarded via the auxiliary channel to the new services, with the retrieved service state. This event crosses the channel in the up direction. Only when the reconfigurator receives this event, it sends a *round-done(0)* to the manager. The *start-round(1)* message includes the *Go* and *Resume* commands. In response, the reconfigurator activates the multiplexer service. When the multiplexer signals the end of the reconfiguration, both the old services and the multiplexer are removed. A *round-done(1)* message is sent to the manager.

Interrupt The *start-round(0)* message contains the *Stop-and-go* command, as well as a *StoreState/Info* command. The reconfigurator proceeds immediately by activating the buffering service, and sending a *Stop* event in the down direction to the channel. This event forces a quiescent state in all services. Afterwards, any state or non-state information is captured by using a *GetInfo* or *GetState* event in the up direction. The reconfigurator sends a *round-done(0)* message. The after received *start-round(1)* message carries the *Go* command, which trigger the reconfiguration. At the end of reconfiguration, the reconfigurator sends a *SetInfo*, or *SetState* event, to load the captured information. Then, a *round-done(1)* message is sent to the manager. Finally, the *start-round(2)* message containing the *Resume* command is sent. The reconfigurator (re-)starts the service composition, and deactivates the buffering service. A *round-done(2)* message is sent.

4.6 Example

We used the pluggable components described in the previous sections for developing an adaptive messaging application. This application, besides text messages, supports cooperative drawing. Moreover, it can publish the location of a user and show the locations of other users. The application was designed as being composed by several services: chat interface, aggregation, location, chat room, and communication services. The chat interface service allows a user to type messages, draw, and see the locations of other users. Furthermore, it allows to choose location privacy settings. The drawing produces a series of points that can be either independent or connected (if they define a line). To reduce the number of messages carrying draw information, the aggregation service collects several points and aggregates them in a single message. There are two implementations of the *aggregation service* available: *simple aggregation service*, and *all aggregation service*. The first aggregates points that are part of the same line, allowing a gradual transmission of the drawing. The later aggregates all points until a text message is produced, grouping all points in a single message. The *location service* publishes the user location if the user location is currently defined to be public. Moreover, it collects locations of other users. The *chat room service* purpose is to abstract the communication service used from the rest of the application services. The communication service relays the text and draw information to other users. Communication can be either point-to-point or group communication. The group communication service is, in fact, a set of several services that offer membership management and failure detection support for a group of users.

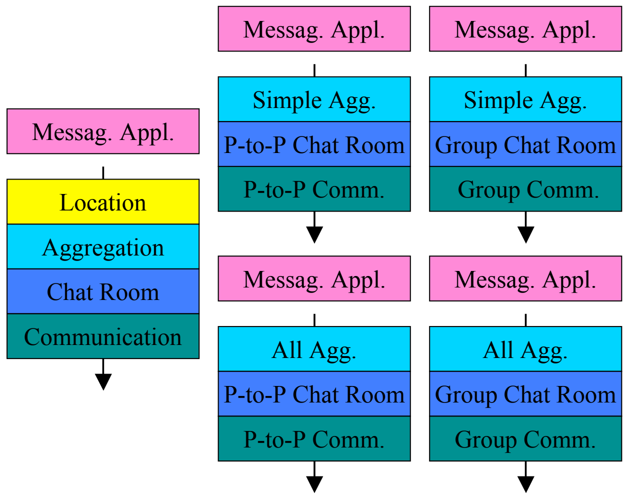

Figure 3: Possible compositions

Using the services described above, there are several possible compositions (see Figure 3). The goal is to use the most efficient composition that satisfies the user requirements, by taking into consideration the operational envelope conditions. Several important adaptation actions were identified: adding/removing the location service, changing the communication service, and changing the aggregation service. The first adaptation answers to changes in the user's preferences. When the user defines his/her location as public or that he/she wants to see other users' locations, the location service is added. The service is removed when the user changes his profile back to private location and shows no interest in other users' locations. The second adaptation allows a richer communication support at the expense of a higher delay in message transmission. Thus, the number of participants determines the exchange of communication service in use. The higher is the number, the better is the rewarding of using the group communication service.

By adding the pluggable components previously described in the service stacks of our application is possible to achieve the adaptation goals just described. The last adaptation allows to reduce the number of packets sent in the network at the expense of delaying their delivery. The final service composition, showing the pluggable components that support adaptation, is illustrated in Figure 4.

The location service does not keep any messages, nor has state information. Moreover, it does not have to be present in all nodes and, hence, it can be added or removed from a local service composition do not requiring coordination with the other nodes. As a consequence of these characteristics, the *Direct* strategy should be used in these reconfiguration actions as it has the lowest delay cost: x. Table 3 illustrates the delay introduced in adding and removing this service in the experiments we have conducted. All values were measured using a network of participants (the number depending

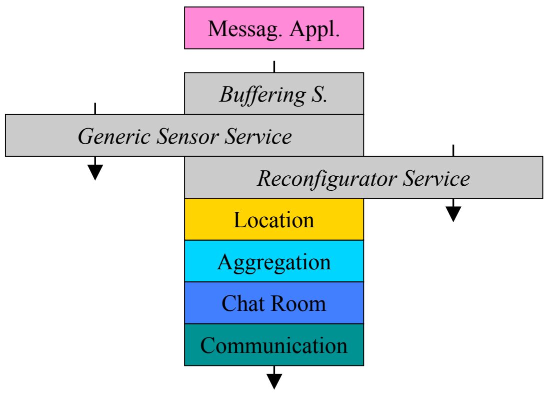

Figure 4: Final composition

Direct strategy

Action	Delay (in seconds)
Adding	0.0239
Removing	0.0236

Interrupt strategy

# participants	Delay (in seconds)
3	0.2476
4	0.3431
5	0.4256
6	0.5195
7	0.6132

Non-interrupt strategy

Action	Delay (in seconds)
Adding	0.0416
Removing	0.0409

Table 3: Strategies' induced delays

on the reconfiguration), executing in Pentium IV / 2.8GHz machines with 1 Gb of memory. These machines were connected through a 100Mbps Ethernet switch.

The communication service adaptation is far more complex than the location service adaptation, since these services cannot afford to lose messages. Moreover, the group communication service has several features demanding that a quiescent state is achieved before reconfiguration. In addition, the different communication services are not compatible. Therefore, to exchange the communication services, coordination of all the involved nodes is required. This exchange does not require any state or other information handling.

With these requirements, the *Interrupt* strategy is the most appropriated. The delay cost is $x + y + 2w$ and the different costs for a rising number of coordinated nodes is depicted by Table 3. Further results show that the reconfiguration time x, and the time to achieve a quiescent state y remains closely the same, however, the waiting time between steps, w, increases with the number of participants. This means that the coordination time rises with the number of participants.

The remaining adaptation is the exchange of aggregation services. This exchange has a low weight on the composition of services. However, these services keep drawing information. This information has to be processed before being exchanged and, hence, the *Non-Interrupt* strategy can be used in this case. The delay cost is $2k$, being k the necessary time to add/remove the multiplexer service. Average values for these costs are presented in Table 3.

5. RELATED WORK

Currently, dynamic reconfiguration is approached in two distinct manners. One regards reconfiguration as part of the functional concerns, while the other clearly separates them. Using the first approach, we find several platforms. One is the well known Odyssey platform [14], where both the operating system and the application are responsible for the adaptation: the first monitoring changes and implementing resource allocation decisions, while the second decides the best allocation of resources. Others, as Bayou [4] concentrate the adaptation responsibility in the operating system. Also, in the first approach, we can find network level protocols [20, 8] that improve performance according to their goals, by tunning specific parameters during runtime. This reconfiguration relies on protocol's performance self-evaluation. The second approach is common on component-based architectures [1, 5], and service composition frameworks [3, 18], that choose to separate the functional and reconfiguration concerns, improving flexibility and maintenance easiness. The latter is applied in several areas [7, 6].

As previously stated, this separation of concerns approach is commonly used mainly in two areas: component-based and service composition frameworks. The first targets component interaction: adding, removing, and changing interactions between components; the second aims at network level protocols: exchanging algorithms, and fine-tunning of parameters.

The work described in this paper pursues a separation of concerns approach to dynamic reconfiguration, targeting services in general, both network, and application level; offering not only fine-tunning of services, but also support to add, remove, or exchange services during runtime.

In composition frameworks both approaches are followed. From the several frameworks [11, 3, 24, 23] among many others, only Ensemble [23] and Cactus [3] regard dynamic reconfiguration. They offer different reconfiguration sophistication. Ensemble relies in a vertical protocol composition to offer a service, and runtime reconfiguration is achieved by switching algorithms. The switch relies in a coordinator-based orchestration, and a stop-and-go local reconfiguration technique, 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 tunning. The framework offers monitoring and agreement features to support automatic dynamic reconfiguration. Since, system's global reconfiguration is expected, Cactus offers a single reconfiguration strategy based on inter-host global orchestration, and non-stop local reconfiguration technique.

This paper addresses a sophisticated dynamic reconfiguration, in which several reconfiguration strategies are available, and is given support to use any of them, as well as develop new strategies. This allows to choose the best strategy for each reconfiguration, improving performance, and addressing different needs.

6. CONCLUSION AND FUTURE WORK

This paper addresses a fundamental problem in the construction of distributed autonomic systems: using the most adequate strategy to apply a required reconfiguration. Several strategies are described in this work, some of them were already proposed in the literature. The collection of strategies described in this work do not pretend to cover all possible ones, but to describe the more significant combinations of orchestration and local reconfiguration technique.

Up to this date, this is the first architecture that allows to use multiple strategies in the same framework, via the implementation of a set of generic pluggable components. These components can be added to an application built as a composition of services to fit adaptation needs. To achieve this goal we have proposed the decomposition of existing strategies into orchestration and local reconfiguration aspects. For each aspect were defined protocols and interfaces that allowed us to implement a generic reconfigurator component. As a proof of concept we have implemented these components in the *RAppia* framework, proving that many aspects of the reconfiguration strategies can be handled with a generic approach with all its advantages.

Moreover, being a distributed system, further work is required in the area of fault tolerance, namely the components behavior when crashes or faults occur. It is important to address these issues to ensure that the system will not be indefinitely locked in reconfiguration when some fault occurs.

7. ACKNOWLEDGMENTS

The authors are grateful to the anonymous referees for their comments on earlier version of the paper. This work was partially funded by FCT project MICAS (POSI/EIA/60692/2004) through POSI and FEDER.

8. REFERENCES

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

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

[3] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting. Constructing adaptive software in distributed systems. In *ICDCS '01: Proc. of the The 21st International Conference on Distributed Computing Systems*, page 635, Washington, DC, USA, 2001. IEEE.

[4] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B. Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

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

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

[7] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *CDS '98: Proceedings of the International Conference on*

Configurable Distributed Systems, page 91, Washington, DC, USA, 1998. IEEE Computer Society.

[8] Y. Kwon, Y. Fang, and H. Latchman. Improving transport layer performance by using a novel mac protocol with fast collision resolution in wireless lans. In *MSWiM '02: Proc. of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*. ACM Press, 2002.

[9] X. Liu, R. van Renesse, M. Bickford, C. Kreitz, and R. Constable. Protocol switching: Exploiting meta-properties. *ICDCSW '00: International Conference on Distributed Computing Systems Workshops*, 00:0037, 2001.

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

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

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

[13] M. J. Monteiro, J. Pereira, and L. Rodrigues. Integration of flight simulator 2002 with an epidemic multicast protocol. In *International Workshop on Large-Scale Group Communication,(in conjuction with The 22nd Symposium on Reliable Distributed Systems)*, 2003.

[14] B. D. Noble and M. Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mob. Netw. Appl.*, 4(4):245–254, 1999.

[15] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, , and P. Vicente. Strong replication in the globdata middleware. In *Proceedings of the Workshop on Dependable Middleware-Based Systems*, pages G96–G104, Washington D.C., USA, June 2002. IEEE. (Suplemental Volume of the 2002 Dependable Systems and Networks Conference, DSN 2002).

[16] L. Rodrigues, J. Mocito, and N. Carvalho. From spontaneous total order to uniform total order: different degrees of optimistic delivery. In *Proceedings of the 21st ACM Symposium on Applied Computing (SAC'06)*, Dijon, France, Apr. 2006. ACM.

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

[18] L. Rosa, L. Rodrigues, and A. Lopes. Appia to rappia: Refactoring a protocol composition framework for dynamic reconfiguration. DI/FCUL TR 07–4, Department of Informatics, University of Lisbon, March 2007.

[19] K. Rothermel and T. Helbig. An adaptive stream synchronization protocol. In *NOSSDAV '95: Proc. of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, London, UK, 1995. Springer-Verlag.

[20] P. Sudame and B. R. Badrinath. On providing support for protocol adaptation in mobile wireless networks. *Mob. Netw. Appl.*, 6(1):43–55, 2001.

[21] B. Swaminathan and K. J. Goldman. Dynamic reconfiguration with i/o abstraction. In *SPDP '95: Proceedings of the 7th IEEE Symposium on Parallel and Distributeed Processing*, page 496, Washington, DC, USA, 1995. IEEE Computer Society.

[22] S. Teixeira, P. Vicente, A. Pinto, H. Miranda, L. Rodrigues, and A. Martins, J. Rito-Silva. Configuring the communication middleware to support multi-user object-oriented environments. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, pages 965–980, Irvine (CA), USA, Oct. 2002.

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

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