CASA – A Framework for Dynamically Adaptive Applications

Doctoral Thesis

for the degree of a
Doctor of Informatics

at the Faculty of Economics,
Business Administration and
Information Technology
of the
University of Zurich

by
Arun Mukhija
from
India

Accepted on the recommendation of

Prof. Dr. Martin Glinz
Prof. Dr. Gustavo Alonso

December 2007

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, December 5, 2007*

The Vice Dean of the Academic Program in Informatics: Prof. Dr. Gerhard Schwabe

*Date of the graduation

To my parents,
for teaching me to think independently.

Acknowledgments

This work would never have come to fruition without the invaluable help, support and guidance provided by many people. Although words can never express my gratitude to them, I would simply like to acknowledge them here for their contributions to my research and to my life as a researcher.

First and foremost, Thanks to my advisor, Prof. Martin Glinz. He has been truly a *Doktorvater* (a German word for Doctoral advisor, which literally means Doctor-Father) to me, giving me the freedom to explore new ideas, and, at the same time, providing his able guidance and motivation to keep me focussed on such an interesting research topic. His analytically deep questions and attention to details have helped me greatly in identifying critical research issues, and in finding solutions to both our satisfaction. He has also been a source of inspiration by way of his thorough and flawless working style. I would like to thank him especially for his understanding and encouragement, which proved very beneficial in creating a productive and pleasant working atmosphere. He has been an ideal advisor that I could ask for.

Thanks to Prof. Gustavo Alonso for agreeing to be my co-advisor, and for providing insightful comments and suggestions that helped to improve the quality of this dissertation.

Thanks to NCCR-MICS (National Center of Competence in Research on Mobile Information and Communication Systems, a center supported by the Swiss National Science Foundation) for partially funding my work, and for providing an excellent platform for disseminating results and getting feedback at different stages of my work through its various scientific conferences and workshops.

Thanks to Rolf Hintermann, Tobias Reinhard and Adrian Gygax for their help with the implementation and evaluation of the prototype system.

Thanks to the members of IfI (Institut für Informatik), and in particular to the members of our research group, for providing an intellectually-stimulating and fun-

filled environment – both within and outside of work. Some names need a special mention here: Christian, Martin, Nancy, Norbert, Samuel, Silvio, Stefan H., Tobias, Uta and Yong – Thanks folks for making my time in Switzerland such a special and memorable one!

Thanks to my family and friends for their support. Unfortunately not all of them can be listed here, but special Thanks to my brother Rakesh and his family for the happy times spent together which helped me re-energize myself to face new challenges in research; in particular, Thanks to Rakesh for his constant encouragement.

Lastly, I dedicate this dissertation to my parents. I will forever remain indebted to them for their love and sacrifices.

It is not the strongest of the species that survives,
nor the most intelligent,
but the one most responsive to change.

Charles Darwin

Abstract

More and more software applications are deployed in dynamic computing environments. These environments are characterized by frequent and unpredictable changes in the availability of resources to software applications, as well as changes in the contextual information of interest to the applications. Some of these changes might present an opportunity for an application to improve its performance or to provide a more relevant functionality, while others might pose a threat to the continued execution of the application. In either case, a software application should be able to adapt its behavior dynamically in response to runtime changes in its execution environment.

This dissertation describes the CASA (Contract-based Adaptive Software Architecture) framework, which enables the development and operation of dynamically adaptive applications. The CASA framework integrates a number of different adaptation mechanisms with an aim to comprehensively meet the adaptation needs of software applications executing in dynamic environments. These adaptation mechanisms can collectively change any part of an application's configuration at runtime, including changing the application components, aspects, attributes and lower-level services. The design of the CASA framework is based on the software engineering principle of separation of concerns. The separation of concerns allows implementing the adaptation concerns as a reusable and shareable runtime adaptation system, while the business concerns of an application are implemented as a part of the application code. The adaptation policy of every application is defined in a so-called application contract in the CASA framework. The application contract is defined using an XML-based specification language, and allows changing the adaptation policy at runtime. The ability to carry out runtime changes in the adaptation policy is useful for customizing the adaptation policy according to a user's current needs and preferences, as well as for evolving the adaptation policy to include new adaptation capabilities.

Zusammenfassung

Mehr und mehr kommen Software Applikationen in dynamischen Umgebungen zum Einsatz. Solche Umgebungen lassen sich durch häufige und unvorhersehbare Änderungen charakterisieren, welche die Ressourcenverfügbarkeit für Software Applikationen beeinflussen. Zusätzlich können sich kontextabhängige Informationen ändern, die für Applikationen von Bedeutung sind. Einige dieser Änderungen können einer Applikation die Möglichkeit bieten, ihre Leistung zu verbessern oder weitere, passendere Funktionalität anzubieten. Andere Änderungen hingegen können den weiteren Ablauf der Applikation gefährden. So oder so, Software sollte in der Lage sein, ihr Verhalten dynamisch anzupassen, um auf Änderungen während der Laufzeit angemessen zu reagieren.

Diese Dissertation beschreibt das CASA (Contract-based Adaptive Software Architecture) Framework, welches die Entwicklung und Ausführung von dynamisch adaptiven Applikationen ermöglicht. Das CASA Framework integriert eine Reihe von verschiedenen Adaptionsmechanismen mit dem Ziel, den Anforderungen von Software Applikationen, die in dynamischen Umgebungen ablaufen, umfassend nachzukommen. Diese Adaptionsmechanismen können zusammen jeden Teil der Konfiguration einer Applikation zur Laufzeit ändern, insbesondere Komponenten, Aspekte, Attribute und systemnahe Dienste. Das Design des CASA Frameworks basiert auf dem softwaretechnischen Prinzip der ≪Separation of Concerns≫. Dies erlaubt es, die Adaptionsmechanismen als ein wiederverwendbares und gemeinsam nutzbares System zu implementieren, während die eigentliche Funktionalität im Applikationscode implementiert ist. Im CASA Framework wird die Adaptionsstrategie jeder Applikation in einem sogenannten ≪Application Contract≫ festgelegt. Ein solcher ≪Application Contract≫ wird mittels einer XML-basierten Spezifikationssprache definiert. Dadurch lassen sich Strategien zur Laufzeit ändern und anpassen. Die Möglichkeit, die Strategie während der Laufzeit zu ändern, ist zum einen sehr nützlich, um die Strategie an die jeweiligen Bedürfnisse und Präferenzen eines

Benutzers anzupassen, und zum anderen, um eine Evolution der Strategie zu ermöglichen.

Contents

List of Figures

List of Tables

List of Abbreviations

AAS	Aspects Adaptation System
CAS	Components Adaptation System
CASA	Contract-based Adaptive Software Architecture
CES	Contract Enforcement System
CM	Context Monitor
CRS	CASA Runtime System
EmCoS	Emergency Coordination System
RCF	Resource Contention Factor
RM	Resource Manager
SC	Service Coordinator
SRC	Software Resource Coordinator

Chapter 1

Introduction

1.1 Need for dynamic adaptation of applications

Software applications executing in today's dynamic computing environments, such as mobile and wireless computing environments, experience frequent and usually unpredictable changes in their execution environment. Changes in the execution environment can be in the form of (1) changes in *resource availability* (e.g. bandwidth, battery power, connectivity etc.), or (2) changes in *contextual information* (e.g. user's location, activity, identity of nearby objects or persons etc.).

Contextual information here refers to (purely) the *information* about the context of an application that may influence the service provided by the application (such as locational information, temporal information, atmospherical information etc.), while resource availability refers to the *physical infrastructure* available to the application for providing this service (such as communication resources, data resources, computing resources etc.).

The inherent *flexibility* and *value* provided by dynamic computing environments have resulted in their widespread growth over the last few years. Moreover, the amount of effort being invested currently (by both academia and industry) for the realization and widespread deployment of *pervasive* and *ubiquitous* computing environments guarantees that dynamic computing environments will continue to grow rapidly for some more time to come.

With the growth of dynamic computing environments, many new and innovative applications are being conceived and developed for these environments. However, in order to be able to cultivate the benefits offered by dynamic computing environments,

the applications need to successfully face the challenge posed by runtime changes in the execution environment.

A change in the execution environment can present an *opportunity* or a *threat* for a running application. For instance, a change in the execution environment in the form of a loss of certain resources required by an application may present a threat, as this may force the application to run at a degraded performance or functionality. On the other hand, a change in the execution environment in the form of a change in contextual information may present an opportunity for an application to provide a more relevant context-dependent service.

In either case, an application should be able to adapt its behavior dynamically (i.e. at runtime, without requiring to stop and restart the application) in order to cope with the changes in its execution environment [MG03, MG04, MG05a].

1.2 State of the art, or how dynamically adaptive applications are developed currently

Adapting an application in response to a change in the execution environment requires ability to: (1) detect a change in the execution environment, (2) take appropriate adaptation decisions in response to the change, and (3) carry out adaptation actions corresponding to the adaptation decisions taken.

As discussed before, a change in the execution environment can be in the form of a change in resource availability or a change in contextual information.

Unreliable availability of resources is now a well-accepted fact for applications executing in dynamic computing environments. Accordingly, several approaches have been proposed for adapting an application in response to a change in the resource availability.

A change in contextual information, on the other hand, has only recently been recognized as a significant part of an application's execution environment that may be exploited by the application to provide a context-dependent behavior. Though some progress has been made in the area of context-aware computing, the focus so far has been mostly on the ways of extracting the contextual information from the execution environment. The actual interpretation and analysis of the contextual information, as well as the resulting adaptation of the application behavior have been largely assumed to be the responsibility of the application itself.

In the most primitive form of application adaptation, as followed in the earliest adaptation approaches, the adaptation mechanism is hard-coded within an application. There are two main disadvantages of hard-coding the adaptation mechanism within an application. First, it increases the complexity involved in application development because of intertwining of the adaptation concerns with the business concerns of the application. And second, the adaptation policy of the application cannot be modified at runtime because of the hard-coding of the adaptation mechanism (apart from any adaptation tuning capabilities provided by the application itself).

Over the last few years, significant progress has been made in the field of middleware technologies for facilitating execution of software applications deployed in dynamic and distributed computing environments. The basic idea behind middleware technologies is to insulate an application from the complexities of the underlying computing environment by providing a layer of software, called middleware, between the application and its computing environment. Middleware is responsible for facilitating interactions of the application with its computing environment. More specifically, middleware is equipped to provide various lower-level services required by an application for its interactions with the computing environment (such as communication services, security services etc.), thereby freeing the application from dealing with the complexities of the computing environment (such as heterogeneity and dynamicity of the environment).

In many ways, middleware seems to be a logical place for providing adaptation capabilities to an application. Since the middleware comes into direct contact with the computing environment of an application, it is in a favorable position to monitor the execution environment of the application and detect any runtime changes therein. Moreover, many times, the adaptation actions involve adapting the lower-level services controlled by the middleware itself. A common example of the adaptation of lower-level services is changing the degree of data compression (i.e. adapting the lower-level data compression service) in response to a change in the communication bandwidth available.

A middleware-based adaptation system further helps in separating the adaptation concerns of an application from its business concerns (as the adaptation responsibility is delegated here to the middleware), thus reducing the complexity involved in application development.

In view of the abovementioned reasons, several middleware-based adaptation approaches have been proposed in the last few years, such as [Sch94, ZBS97, PSA+03] etc. In the earlier middleware-based adaptation approaches, the adaptation decisions were taken by the middleware itself by following some standard pre-defined adaptation rules encoded within the middleware. That is, in such approaches, the application had little control over the adaptation decisions. However, some of the latter middleware-based adaptation approaches realized that the application's participation is crucial in taking adaptation decisions because of different adaptation needs of different applications. This resulted in the concept of application-aware adaptation [NSN+97], where the application has a control over the adaptation decisions to be taken.

Although middleware-based adaptation approaches offer several advantages and are indeed quite successful in meeting the adaptation requirements of many applications, their scope is rather limited. More specifically, middleware-based adaptation approaches are largely restricted to adapting the lower-level services used by an application. Whereas, sometimes adaptation of lower-level services alone may not be sufficient, and a change in the application code itself may be required as a means of application adaptation.[1]

Continuing with our above example of adapting the data compression service, for a small drop in the communication bandwidth, a corresponding increase in the degree of data compression may be sufficient. However, for a large drop in the communication bandwidth, increasing the degree of data compression alone may not be sufficient, and a change in the application code may be required in order to reduce the data throughput of the application. Similarly, in response to a change in contextual information, usually a corresponding change in the application's functionality is required which may necessitate a change in the application code.

In the last few years, a number of approaches have been proposed for enabling runtime software evolution [HG98, MPG+00, Dmi01]. These approaches have been targeted towards evolving software applications which are required to be always available, i.e. where shutting down an application temporarily for the evolution is not preferred. Runtime software evolution naturally involves changing the application code at runtime. Though the original focus of these approaches has been software

[1]The relationship between the application code (consisting of application components and aspects) and the lower-level services used by an application is discussed in Section 1.3.

evolution, the concepts developed as a part of these approaches are nevertheless useful for application adaptation as well.

Modern software applications are developed as a composition of software components, where the components collaborate among themselves in order to accomplish the required task of the application. In a component-based application development, the components encapsulate their implementation details, interact with each other only through their well-defined interfaces, and generally follow the principle of separation of concerns [Szy98]. These characteristics of components make them a natural unit of adaptation for adapting the application code. That is, adaptation of the application code can be conveniently carried out by recomposing (adding / removing) the components at runtime, as in some of the recent approaches for runtime software evolution [DL03, RP03, LPH04].

Another field closely related to runtime adaptation of applications is dynamic AOP (aspect-oriented programming) [KMM+98, Bol99]. AOP [KLM+97] has been strongly advocated in the last few years as a means for separating the crosscutting concerns (implemented as aspects) of an application from its core concerns (implemented as conventional components), and thereby reducing the application's complexity. Examples of crosscutting concerns are access control, persistence management, transaction management etc. More recently, some approaches have been proposed for weaving and unweaving of aspects into / from an application at runtime. These approaches are commonly referred as dynamic AOP approaches. Some of the popular dynamic AOP systems include PROSE [PGA02, PAG03, NA05], JAC [PSDF01], TRAP/J [SMCS04] etc.

Since the adaptation of an application in response to a change in its execution environment may involve a change in the crosscutting concerns rather than the core concerns, dynamic AOP approaches provide a useful means of adapting an application in certain situations. As an example, in response to a change in an application's location from a low-risk area to a high-risk area, the corresponding access control aspect of the application may need to be adapted. Similarly, in response to a loss of connection to a remote data storage, the corresponding persistence management aspect of the application may need to be adapted.

1.3 Need for a framework-based approach

Different approaches for application adaptation, as discussed in Section 1.2, are quite successful in meeting the specific adaptation requirements of their target applications. That is, these approaches are capable of adapting a software application in response to specific changes in its execution environment. However, none of these adaptation approaches is individually capable of meeting a wide range of adaptation needs of software applications executing in dynamic environments.

In particular, the same application may benefit from one adaptation approach for certain specific changes in its execution environment, but may require another adaptation approach for some other kinds of changes in the execution environment that could not be served by the first adaptation approach. For example, in response to a runtime change in the resources available to an application, a corresponding adaptation of the lower-level services used by the application may be sufficient for the purpose. Whereas, in response to a change in the contextual information of interest to the application, a corresponding change in the component composition of the application may be required in order to realize a change in the application's functionality. Similarly, recall our earlier example where for a small drop in communication bandwidth a change in the lower-level data compression service is required, while for a large drop in bandwidth a change in the application code is required to reduce the throughput of the application.

Though the adaptation approaches discussed in Section 1.2 are not individually capable of meeting a wide range of adaptation needs of software applications, these adaptation approaches can in fact complement each other in order to meet those adaptation needs.

There is a need to integrate different adaptation mechanisms provided by the different approaches into a single comprehensive framework in order to benefit from the complementary nature of these approaches. This way, the resulting framework will have the capability to meet the adaptation needs of a broad and diverse set of applications executing in dynamic environments.

We classify different mechanisms for application adaptation according to the level where the adaptation takes place, as follows:

- dynamic recomposition of application components
- dynamic weaving and unweaving of aspects

- dynamic change of application attributes
- dynamic change of lower-level services

Figure 1.1 illustrates the relationship between the application code and the lower-level services used by an application for its execution.

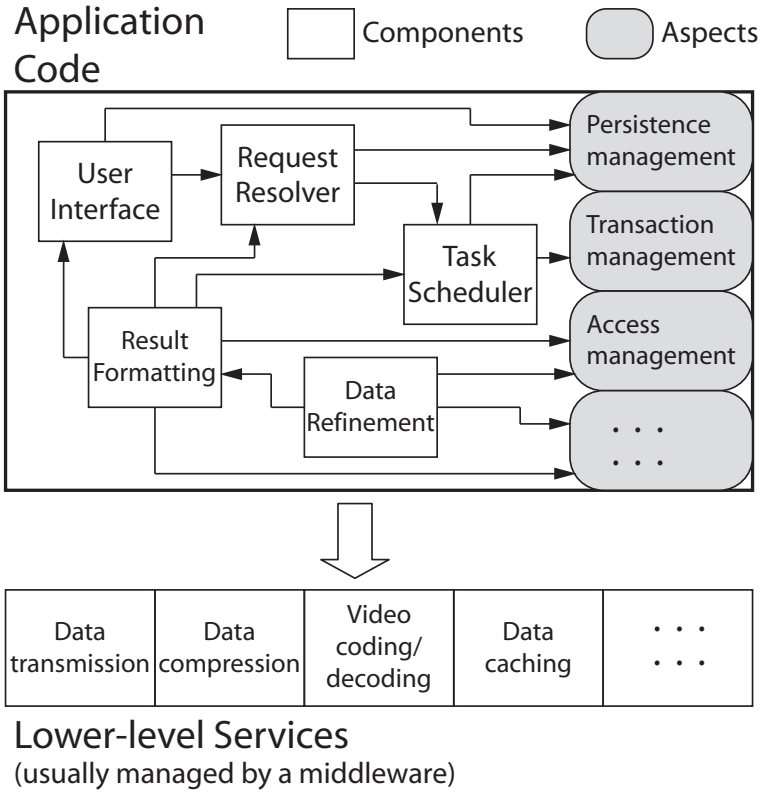


Figure 1.1: Constituents of an application

The upper part of Figure 1.1 shows the application code consisting of application components and aspects. The directions of arrows within the application code indicate the dependencies among components, and the dependencies of components on aspects. The lower part of Figure 1.1 shows the lower-level services used by the application code. The different constituents of an application shown in the above figure work together to accomplish the required task of the application. The organization of an application into components, aspects and lower-level services helps in reducing

the complexity involved in developing and maintaining software applications.

Lower-level services are not application-specific, and can be shared among different applications deployed on the same platform. Some of these services might be specialized to a few application domains, such as video coding-decoding service, while others might be more widely applicable, such as data transmission service. Since these services are mostly generic and easily shared among different applications, these are provided by the deployment platform. These services are very often managed by an underlying middleware system which is a part of the deployment platform.

Even though some or many of the application components and aspects can be reusable, these are generally not shared among different applications. Accordingly, the application components and aspects are parts of the specific application code (i.e. these are not provided as a part of the deployment platform). As discussed earlier, application components implement the core concerns, and the aspects implement the crosscutting concerns of an application.

The third mechanism listed above is used for adapting an application by simply changing certain attributes of the application, without adding or removing any of its components or aspects, or changing any lower-level services. Changing an application attribute here refers to changing the value of a certain variable that is part of the application code. This variable can be encoded within one of the application components or in a configuration file. It is assumed here that changing the value of this variable at runtime results in the adaptation of application behavior.

In this dissertation, we consider an application configuration to consist of application components, aspects, attributes and lower-level services.

In addition to the mechanisms listed above, another form of adaptation can be carried out directly at the resource level. For example, in response to a loss of a certain resource used by an application, an adaptive middleware may locate an alternative resource providing the same type of service as the lost resource, and allocate this alternative resource to the application. Since this form of adaptation is carried out transparently without resulting in a change in the application configuration, it is not a part of the proposed application adaptation framework. However, this form of adaptation at the resource level is obviously quite useful, and should be provided if possible.

1.4 Human metaphor to application adaptation

In everyday life, we human beings adapt our behavior all the time according to our current surroundings and environment. It is interesting to observe the analogies between the adaptation of human behavior and application behavior, in response to changes in their respective environments. As we will notice from our discussion below, there are indeed several similarities between the two, and the natural approach of human adaptation can help in understanding the requirements for developing an appropriate approach for application adaptation.

To understand the adaptive behavior of humans, let us look at a typical day in the life of John.

As John goes about his daily life, the contextual information of interest to John keeps changing almost consistently. Examples of contextual information are location of John, people in his surroundings, time of day, weather conditions etc. John collects data about the current contextual information of interest through his senses to see, hear, touch, smell etc. The data collected by various sensors of John is passed to his mind that interprets and analyzes this data, and – based on the ontological information already stored in the mind – defines the current contextual information of interest to John.

Based on the current contextual information, John decides an appropriate behavior. John may typically have a number of alternative behaviors to choose from, corresponding to the given contextual information. The behavior ultimately chosen by John will likely depend upon the resources currently available to John. Practically, the alternative behaviors corresponding to the given contextual information may have some kind of preferential ordering within John's mind, and each of these alternative behaviors may have their own specific resource requirements. Depending on the resources currently available, John will select the behavior which is highest in this preferential ordering and for which sufficient resources are available. Examples of resources here can be anything that John may use, such as a car, phone, map, directory service, pizza delivery service etc.

As an example, consider the current contextual information of interest to John be: "location = home" and "time = 45 minutes to attend a meeting at office". Now, let the alternative behaviors possible for John corresponding to the above contextual information be: (1) to reach his office by car, (2) to reach his office by public transport, (3) call up his office to cancel the meeting. Assuming that the

meeting is important for John, the alternative (3) will be the least preferred by John. Among the first two alternatives, John may prefer alternative (1) over (2), i.e. to go to office using his car rather than using public transport, as John may find going by car to be more comfortable and time-saving. Alternative (1) requires that a car in functional condition is available to John. However, if John realizes that his car has a flat tire since last evening, he may have to select alternative (2), even though it is actually less preferred by John than alternative (1). If a little later, John realizes that there is a strike of all public transport in the city, he will be left with no other alternative than (3), i.e. to call up his office and cancel the meeting, though it is actually the least preferred alternative.

The above example shows that, in general, the current contextual information presents the human with a limited number of alternative behaviors to choose from. The ultimate choice of a behavior is dependent upon the human's preference as well as the current resource availability.

Over time, the contextual information of interest to John as well as the resources available to John may change. Such changes in the environment are quite common in the daily life of John, and John needs to adapt his behavior accordingly in response to these changes.

The adaptation of John's behavior can take place at various levels – in particular at the level of the subconscious mind or the conscious mind of John.

For example, suppose John is talking to Mary over the phone. Upon realizing that Mary is not able to hear John properly due to some disturbance in the phone line, John automatically raises the volume of his voice. Rising of the volume of John's voice is carried out automatically by the subconscious mind of John. In a similar manner, the subconscious mind of John does several instant reactions in response to certain changes in the environment. For example, if John tastes a very bitter substance in his food, he might spit it out instantly. Spitting the very bitter substance here is controlled by the subconscious mind of John.

Naturally, many of the other changes in John's environment require adaptation of John's behavior directly controlled by the conscious mind of John. For example, on meeting a potential customer, John might behave in a certain way to persuade the customer to buy a product that John is trying to sell. This persuasive behavior of John is controlled by the conscious mind of John.

The above example shows that for certain types of changes in the environment

an adaptation at the level of subconscious mind is sufficient, while some other types of changes demand an adaptation at the level of conscious mind.

Now let us compare the above metaphor to software applications. Even in software applications, the current contextual information of an application presents the application with a choice of alternative behaviors, while the final choice of a behavior is dependent upon the user's preference as well as the current resource availability.

Similar to the adaptation of human behavior at different levels of human mind, the adaptation of software applications also takes place at different levels. The adaptation at the level of subconscious mind of humans is similar to the adaptation of lower-level services carried out at the level of middleware for adapting application behavior. Whereas, the adaptation at the level of conscious mind is similar to the adaptation of application's components, aspects or attributes carried out at the level of application code.

Humans are naturally more powerful than software applications. In particular, humans may have abilities to predict any changes in the environment in advance, or even control the environment to some extent. Software applications normally do not have such abilities. Therefore, we have assumed here that an application can neither predict any changes in its execution environment in advance, nor have any direct control over the environment.

1.5 Contributions of the dissertation

In this dissertation, we present the CASA (Contract-based Adaptive Software Architecture) framework for enabling dynamic adaptation of software applications in response to changes in their execution environment.

The key features of the CASA framework are:

Comprehensive: The CASA framework integrates a number of different adaptation mechanisms identified in Section 1.3. This way the CASA framework is able to provide support for adaptation at various levels of an application, and thereby comprehensively meet a wide range of adaptation needs of software applications executing in dynamic environments.

Runtime support: The CASA framework provides a runtime system for handling adaptation concerns, i.e. for monitoring the execution environment, taking

adaptation decisions, and carrying out adaptation actions. The runtime system considerably facilitates development of new adaptive applications by reducing the effort required of the application developer.

Modifiable adaptation policy: The CASA framework follows a contract-based adaptation policy, wherein the adaptation policy is defined in a so-called application contract. The application contract is external to the application, and is defined using an XML-based specification language. The contract-based adaptation policy allows modifying the adaptation policy at runtime. A runtime modification of the adaptation policy is useful for customizing the adaptation policy according to an individual user's needs and preferences, as well as for evolving the adaptation policy e.g. to prepare an application for handling any changes in the execution environment which were not foreseen at the time of application development.

Separation of concerns: The adaptation concerns of an application are separated from its business concerns in the CASA framework. The runtime system provided by the CASA framework and the contract-based adaptation policy jointly facilitate this separation. The separation of concerns, in turn, helps in reducing the complexity involved in the development and maintenance of dynamically adaptive applications.

In developing the CASA framework, our objective has been to build on the advances already made in the field of dynamic adaptation of software applications. In particular, there have been a number of adaptation systems developed for adapting the lower-level services used by an application at the middleware-level. Any of these middleware-based adaptation systems can be integrated with the CASA framework, subject to satisfying certain minimum requirements concerning their integration (Chapter 4). Similarly, for dynamic weaving and unweaving of crosscutting aspects, the CASA framework integrates a system called PROSE [NA05], which is a flexible and efficient system developed for this purpose (Chapter 4).

However, for dynamic recomposition of application components, none of the existing systems were found to be suitable for integration with the CASA framework in terms of the flexibility offered by these systems. Therefore, we have developed our own approach for dynamic recomposition of components in the CASA framework (Chapter 5).

Several of the existing middleware-based adaptation systems incorporate capa-

bilities for monitoring the availability of resources. Therefore, we have delegated the task of monitoring resource availability to the underlying middleware-based adaptation system integrated with the CASA framework (Chapter 3).

As we mentioned before, recognition of the contextual information as a useful part of an application's execution environment is a relatively recent development. Accordingly, some approaches for monitoring contextual information have been developed, and more progress is underway. In the CASA framework, we have used an abstract context monitor, which relies on externally developed mechanisms for monitoring contextual information (Chapter 3).

We should point out here that our focus in this work has been more on the ways of adapting an application's behavior dynamically, rather than on the ways of monitoring the execution environment (resource availability and contextual information).

1.6 Example applications to benefit from the CASA framework

Below we give a few example applications that may benefit from the dynamic adaptation capability provided by the CASA framework, followed by a short discussion on the capabilities of the CASA framework. These example applications form just a small subset of the target applications for our work. There can of course be innumerable other kinds of applications that may benefit from the dynamic adaptation capability provided by the CASA framework.

Tourist-Guide: Consider a tourist taking a walk through a new city. The tourist is guided by a *Tourist-Guide* application running on a mobile handheld device. While traveling, as the tourist comes in the vicinity of a touristic point-of-interest, the *Tourist-Guide* application provides a service relevant to that point-of-interest. The quality of the service provided by the *Tourist-Guide* application ultimately depends on the resources currently available to the application. A dynamic change in the point-of-interest here constitutes a change in the contextual information of interest to the tourist, and demands an appropriate adaptation of the service provided by the *Tourist-Guide* application. The resources available to the *Tourist-Guide* application may also change at runtime, and demand a corresponding adaptation of the quality of service. For example, if the contextual information of interest to the tourist changes

from *shopping mall* to *museum*, the *Tourist-Guide* application provides relevant information about the museum in place of the information about the availability of the items from the tourist's shopping wish-list in the shopping mall. The quality of information about the museum provided by the *Tourist-Guide* application in turn depends on the communication bandwidth available between the tourist's handheld device and the information server of the museum. If the bandwidth available is high, a multimedia-rich brochure of the museum is fetched and displayed by the application, but if the bandwidth is low or drops during the service, only a simple text-based brochure of the museum can be fetched and displayed by the application.

Collaborative-Working: Consider a hypothetical collaborative working scenario, where a number of participants are collaborating on a common mission. The participants may be geographically distributed, and some of them may even be mobile at any given time. For the collaboration to work, each participant runs an instance of the *Collaborative-Working* application, which includes a video display showing other participants, a shared drawing space and a discussion board. Some of the resources available to a participant, such as communication bandwidth and battery power, are likely to vary over time because of the mobility and other constraints. Similarly, the contextual information related to a participant (in a meeting, at home etc.) is likely to vary over time. The runtime changes in resources and contextual information demand an appropriate and non-disruptive adaptation of the *Collaborative-Working* application. For example, in response to a small drop in the bandwidth available to a participant, the quality of the video display may be reduced accordingly. Whereas for a large drop in the bandwidth, any video content may be removed altogether. A change in the contextual information may also influence the application behavior, e.g. only important updates may be sent while the participant is in a meeting.

Personal-Assistant: A *Personal-Assistant* application is used to assist a user in her daily business, and can be customized by the user to meet her individual needs and preferences. For example, if the user needs to attend a meeting in some time (as informed by the user's electronic agenda which acts as a source of contextual information here), the *Personal-Assistant* application assists the user in finding the fastest route to the place of meeting by collecting information about the current location of the user, traffic congestion situation on various alternative routes etc. While in the meeting, the *Personal-Assistant* application retrieves past notes related to each

of the participants present at the meeting. If the meeting ends around noon, the *Personal-Assistant* application provides information about the restaurants nearby, and may even suggest one for lunch according to the user's likings and stored diet requirements. The quality of the service provided by the *Personal-Assistant* application can vary depending on the resources currently available to the application. For example, in deciding the fastest route to a given destination, if the application has access to the information server of the traffic department, then it can provide an accurate result based on the precise information about the traffic conditions on alternative routes. However, if the connection to the traffic information server is broken, then the *Personal-Assistant* application may form an ad-hoc network with other similar applications running on the devices in the vicinity, and provide a result based on the information shared between these applications. The result in the second case can be of inferior quality (less accurate) compared to the one in the first case, because of an important resource (traffic information server) being unavailable in the second case.

Discussion: The CASA framework is able to adapt any of the constituents of an application configuration, i.e. application components, aspects, attributes and lower-level services. For adding any new application components (which implement the core functionality provided by an application) and aspects (which implement the crosscutting functionality of the application), the executable codes of these components and aspects need to be provided. The locations of these components and aspects are specified in the application contract, which defines the adaptation policy of the application. The runtime replacement of components and aspects is carried out using the adaptation mechanisms provided in the CASA framework. For adapting the application attributes (which can influence the application behavior to some extent), the concerned application needs to implement appropriate callback methods that can be invoked by CASA at runtime. The adaptation capability achievable by changing the application attributes is pre-defined, and is usually restricted to minor tuning of the application behavior. Adaptation of the lower-level services involves activating or deactivating certain services, or changing the parameters of some running services. The lower-level services are usually controlled by an underlying middleware system, while CASA interacts with this system by invoking certain operations for carrying out the required changes in the lower-level services.

Depending on the adaptation requirements of a given application, any combination of the above adaptation mechanisms can be used. For instance, a change in contextual information usually affects an application's functionality and therefore demands an appropriate adaptation of the application components or aspects, while it rarely requires an adaptation of the lower-level services. A change in resources, on the other hand, is usually well responded by an adaptation of the lower-level services or application attributes, though in some cases it may also require an adaptation of the application's functionality achieved by changing the application components or aspects.

In our earlier example of a *Collaborative Working* application, a small drop in the communication bandwidth (resource) is likely to require a corresponding change in the lower-level video coding/decoding service for reducing the frame-rate or resolution of video, without requiring any changes in the application components, aspects or attributes. Whereas, a change in contextual information from office to meeting room can be best served by replacing a *filter* component of the application, such that only important updates are forwarded to the user rather than delivering all messages.

On the other hand, in our example of a *Personal Assistant* application, even a change in a resource (connection to a traffic information server) requires a corresponding change in an application component (the route planning component). A change in contextual information in all of the example applications discussed above requires a corresponding adaptation of the application components or aspects.

1.7 Organization of the dissertation

The rest of the dissertation is organized as follows.

Chapter 2 presents the overall design and working of the CASA framework, including details of application contract specifications and service negotiations among distributed applications. Chapter 3 discusses techniques for monitoring the execution environment (resource availability and contextual information) of an application, and provides details of the Resource Manager and the Context Monitor used in the CASA framework. Chapter 4 discusses adaptation of lower-level services used by an application at the middleware level, and adaptation of crosscutting aspects of an application. Chapter 5 provides details of the dynamic components recomposition

approach followed in the CASA framework. Chapter 6 describes the implementation of a prototype system based on the CASA framework, and discusses the performance evaluation of the prototype system. Chapter 7 gives an overview of other approaches related to our work on dynamic adaptation of applications. Finally, Chapter 8 presents concluding discussion, and outlines directions for future work.

Chapter 2

The CASA Framework

This chapter is organized as follows. Section 2.1 presents the main design goals of the CASA framework. Section 2.2 gives an overview of the CASA framework. Section 2.3 presents the design of the CASA framework. Section 2.4 presents application contract specifications. Section 2.5 describes the overall working of the CASA framework. Section 2.6 presents details of service negotiations among distributed applications. Finally, Section 2.7 concludes this chapter with a discussion on the capabilities and limitations of the CASA framework, and how to use the framework in practice.

2.1 Design goals of the CASA framework

The design of the CASA framework has been guided by the following design goals:

Separation of Concerns: The design of the CASA framework is based on the fundamental principle of *separation of concerns* [Dij82]. Following this principle, the adaptation concerns of an application are separated from its business concerns. The separation between the adaptation concerns and business concerns of an application results in considerably reducing the complexity involved in developing and maintaining dynamically adaptive applications. This separation is achieved through the following two characteristics of the CASA framework.

Independent and reusable adaptation infrastructure: The adaptation infrastructure in CASA consists of the entities responsible for monitoring the execution environment

of running applications, and adapting these applications. This adaptation infrastructure is independent of the business logic of the application. The entities constituting the adaptation infrastructure are also loosely coupled among themselves, i.e. any of these entities may evolve independent of the others.

The independent adaptation infrastructure in CASA implies that it can be reused and shared among applications. This results in significantly reducing the complexity involved in developing dynamically adaptive applications, as the application developer is able to better focus on the business logic of the application, without having to worry about implementing the adaptation mechanisms for the application.

Contract-based adaptation policy: In CASA, the adaptation policy of every application is defined in a so-called application contract. The application contract is external to the application, and is specified in an application-independent format. The contract-based adaptation policy further enables separating the adaptation concerns of an application from its business concerns, and also plays a facilitating role in the development of an independent and reusable adaptation infrastructure mentioned above.

Comprehensive Adaptation Solution: In order to satisfy a wide range of adaptation requirements of the software applications executing in dynamic computing environments, the CASA framework provides a comprehensive adaptation solution. This is achieved by: (a) being able to monitor changes in different parameters of an application's execution environment, and (b) supporting a number of adaptation mechanisms for adapting different parameters of an application's configuration.

In particular, the CASA framework supports the following four adaptation mechanisms (as identified in Chapter 1):

- dynamic recomposition of application components
- dynamic weaving and unweaving of aspects
- dynamic change of application attributes
- dynamic change of lower-level services

The above adaptation mechanisms can collectively adapt any parameter of an application's configuration, and thus help in providing a comprehensive adaptation solution.

Modifiable Adaptation Policy: As stated earlier, CASA follows a contract-based adaptation policy. In addition to facilitating the separation between adaptation concerns and business concerns, a major contribution of the contract-based adaptation policy is in supporting runtime modifications of the policy. Runtime modifications of the adaptation policy are possible as the application contract defining the adaptation policy is external to the application, and uses an XML-based specification language for defining the adaptation policy.

Runtime modifications of the adaptation policy of an application help in customizing the adaptation policy according to the user's needs and preferences. Customization of the adaptation policy is important as each individual user of an application may have different requirements or preferences concerning the way an application should behave and adapt in response to changes in the execution environment, and these user requirements may even change dynamically. Runtime modifications of the adaptation policy also mean that the application developer is able to evolve the adaptation policy of the application at runtime. This may be required, for example, to equip an application to deal with certain changes in the application's execution environment that were not foreseen at the time of application development, or to add new adaptation capabilities to the application and thereby change the way it adapts to certain execution environment conditions.

2.2 Overview of the CASA framework

In CASA, the execution environment E of an application is defined by the set of resources available to the application and the contextual information relevant to the application. That is,

$$E = \{Availability\ of\ Resources,\ Contextual\ Information\}$$

As discussed in Chapter 1, an application configuration G consists of the components, aspects, attributes, and lower-level services constituting the application. That is,

$$G = \{Components,\ Aspects,\ Attributes,\ Lower\text{-}level\ Services\}$$

The application contract defines the mappings between all possible states of the execution environment of an application and the corresponding application configurations. That is,

$$Application\ Contract = \{(E_1, G_1), (E_2, G_2), \ldots, (E_n, G_n)\}$$

The adaptation process in CASA consists of the following three steps: (1) determining a change in the execution environment, say from E_i to E_j, (2) consulting the application contract to determine the configuration corresponding to E_j, i.e. G_j, (3) changing the application configuration from G_i to G_j.

A change in the execution environment of an application means a change in the availability of resources and/or a change in the contextual information relevant to the application. A change in the application configuration in response to a change in the execution environment implies changing one or more constituents of the configuration (i.e. components, aspects, attributes, and lower-level services). An application configuration can be represented as,

$$G_i = \{C_i, A_i, T_i, L_i\}$$

where C_i is the set of components constituting G_i,

$$C_i = \{C_{i_1}, C_{i_2}, \ldots, C_{i_p}\}$$

A_i is the set of aspects constituting G_i,

$$A_i = \{A_{i_1}, A_{i_2}, \ldots, A_{i_q}\}$$

T_i is the set of application attributes constituting G_i,

$$T_i = \{T_{i_1}, T_{i_2}, \ldots, T_{i_r}\}$$

and L_i is the set of lower-level services constituting G_i

$$L_i = \{L_{i_1}, L_{i_2}, \ldots, L_{i_s}\}.$$

Changing the component composition of an application from C_i to C_j (as a result of the change in configuration from G_i to G_j) implies (1) adding any components that are in the set C_j but not in C_i, and (2) removing any components that are in C_i but not in C_j. A change in the aspects composition is carried out similarly. A change in application attributes from T_i to T_j implies resetting the attribute values, such that the values specified in T_j are assigned to the corresponding attributes replacing

the previous values assigned by T_i. A lower-level service specified in L_i or L_j denotes the corresponding service and any parameters characterizing the service. A change in lower-level services from L_i to L_j implies that (1) if a service exists in both L_i and L_j, then resetting the parameters of this service as specified in L_j, (2) if a service exists in L_j but not in L_i, then activating this service with the specified parameters, and (3) if a service exists in L_i but not in L_j, then deactivating this service.

The components, aspects, attributes, and lower-level services that remain the same across all configurations are not specified in the application contract, and are instead activated during the application initialization.

Every computing node hosting dynamically adaptive applications is required to run an instance of the CASA Runtime System (CRS). The CRS has two responsibilities: firstly, it monitors the execution environment on behalf of the running applications. Secondly, in case of significant changes in the execution environment, the CRS carries out the adaptation of the affected applications.

The adaptation process in CASA discussed above is carried out by the CRS as follows (refer Figure 2.1):

1. *Detect a change in the execution environment:* The CRS continuously monitors the execution environment of running applications, and detects any significant changes in the execution environment that may warrant an adaptation of one or more running applications.

2. *Consult adaptation policy:* Once the CRS detects a significant change in the execution environment, it consults the adaptation policies of the affected applications (specified in the respective application contracts) in order to decide if there is a need to take certain adaptation decisions.

3. *Carry out adaptation actions:* If certain adaptation decisions are taken in the previous step, then the CRS carries out adaptation actions on the concerned applications corresponding to the adaptation decisions taken. The adaptation actions involve changing the configurations of the concerned applications.

In the next section, we discuss the detailed design of the CASA framework, and the functions of individual entities constituting the framework.

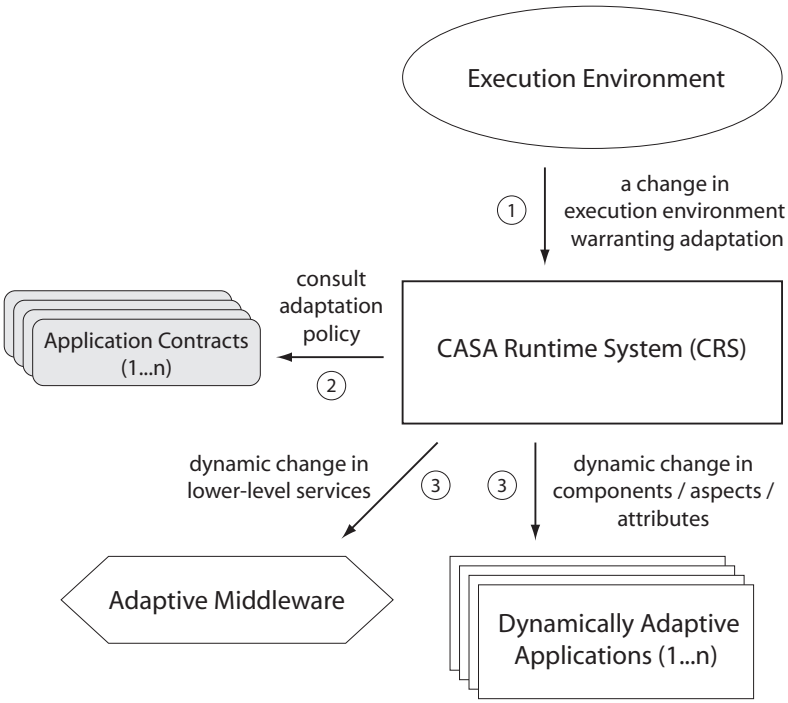


Figure 2.1: Adaptation process in the CASA framework

2.3 Design of the CASA framework

Before discussing details of the design of CASA framework, we should emphasize that our focus in this work has been more on the mechanisms for adapting applications, rather than on the mechanisms for monitoring the execution environment. Therefore, we present only an abstract design of the mechanisms used for monitoring the execution environment (contextual information and resources), leaving the details and refinements of these mechanisms as a future work.

2.3.1 Design overview

Figure 2.2 depicts the CASA framework. The entities within the dotted area represent the CASA Runtime System (CRS). Below we first briefly describe the responsibilities of each of these entities, and then discuss the roles of these entities in monitoring the execution environment and adapting applications.

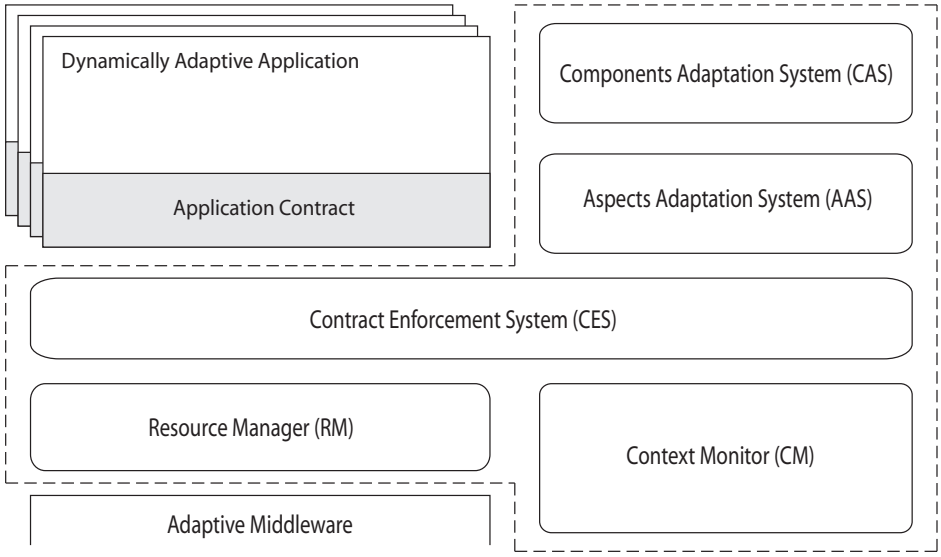


Figure 2.2: Design of the CASA framework

Contract Enforcement System (CES): The CES is responsible for coordinating the working of all other entities of the CRS.

Resource Manager (RM): The RM is responsible for allocating resources to applications in accordance with the relative priorities of these applications. The RM relies on an external resource monitoring service for information about the current availability of local and network resources. Such a resource monitoring service can be embedded within the underlying adaptive middleware.

Context Monitor (CM): The CM is responsible for monitoring the contextual information relevant to applications, and detecting any runtime changes in the contextual information.

Components Adaptation System (CAS): The CAS is responsible for recomposing application components at runtime, as and when instructed by the CES.

Aspects Adaptation System (AAS): Similar to the CAS, the AAS is responsible for dynamically changing the aspects composition of an application, as and when instructed by the CES. The AAS is integrated with the PROSE system [NA05],

which actually carries out dynamic weaving and unweaving of aspects into / from an application.

Adaptive Middleware: Though an adaptive middleware is technically not a part of the CRS, the CRS relies on the following services provided by the underlying adaptive middleware for its operations:

- The adaptive middleware is responsible for dynamically adapting the lower-level services used by an application, as and when instructed by the CES.
- The adaptive middleware can be optionally responsible for monitoring the current availability of local and network resources, and updating the RM about any runtime changes in the availability of these resources.

2.3.2 Monitoring the execution environment

As discussed earlier, the execution environment of an application can be divided into: *contextual information* and *resources*. The term contextual information here refers to (purely) the *information* about the context of an application that may influence the service provided by the application (such as locational information, temporal information, atmospherical information etc.), while the term resources refers to the *physical infrastructure* available to the application for providing this service (such as communication resources, data resources, computing resources etc.).

Monitoring contextual information: The Context Monitor (CM) in the CASA framework is responsible for monitoring contextual information. Examples of contextual information are user's location, activity, identity of nearby objects or persons etc.

Monitoring the contextual information relevant to an application consists of the following steps:

- acquiring the data related to contextual information,
- structuring the acquired data based on an application domain-specific ontology, and
- deducting the final knowledge, i.e. the contextual information relevant to the application, from this data.

A variety of context sensors are used for acquiring the data related to contextual information. The context sensors can range from being highly sophisticated (e.g. to determine the identity of a person, sophisticated sensors are employed) to very simple (e.g. to determine the current location, a GPS monitoring device is sufficient). The data acquired by various context sensors is structured according to the application domain-specific ontology. Once the data is formally structured, the current contextual information relevant to the application can be deducted from this data in a straightforward manner.

As can be seen from above, the application domain-specific ontology plays a very important role in reasoning about the current contextual information. The ontology defines different contextual parameters and their possible values in OWL (Ontology Web Language) [OWL07]. OWL is based on RDF (which in turn is based on XML), and provides rich expression capabilities for defining an ontology.

It is very likely that different application domains might require different implementations of the CM, as no single CM can be presumed to be able to monitor all the different kinds of contextual parameters across different application domains. This is because of a wide range of context sensing devices required for monitoring different contextual parameters.

The above implies that an application needs to make sure that the CASA framework it is going to use has the right CM installed, before using the framework. This should not be a major problem as there would only be a limited number of application domains for dynamically adaptive applications. Moreover, a given node is likely to cater to the applications from the same application domain, and therefore would have the right configuration of the CASA framework installed for that domain. In our work, we propose only an abstract design of the CM, which can be customized for the individual application domains.

Further details on monitoring contextual information are given in Chapter 3.

Monitoring resources: We categorize resources into: hardware resources and software resources. Hardware resources include both local resources (such as memory, CPU, battery) as well as network resources (such as communication bandwidth). Software resources can be local or remote.

For monitoring hardware resources (both local and network), several resource monitoring services have been developed that operate at the platform (operating

system, network) level, where resources can be monitored efficiently. Examples of such resource monitoring services are Remos [LMK+03], Dproc [APK+03] etc. Therefore, for monitoring local and network hardware resources, CASA relies on an external resource monitoring service. Several adaptive middleware systems have built-in hardware resource monitoring services that can be made available to CASA. For the discussion in this dissertation, we assume that CASA is integrated with an adaptive middleware system called Odyssey [NSN+97], which can provide resource monitoring service. The Resource Manager (RM) in CASA is responsible for interacting with the underlying resource monitoring service.

For monitoring any software resources required by an application, the RM includes a Software Resource Coordinator (SRC) entity. The SRC may use a variety of service discovery infrastructures for discovering software services required by an application.[1] For the discussion in this dissertation, we assume that software services required by an application are implemented as Web services, and use XML-based Web service standards like UDDI (for service registry), WSDL (for describing service interface) and SOAP (for message wrapping) [ACKM04]. However, automated discovery and selection of software resources requires semantic information, which is not provided in the plain WSDL specifications. For semantic specification, we rely on OWL-S [OS07], which is an OWL-based ontology for the semantic specification of Web services.

An OWL-S specification of a service has three parts: profile, process model and grounding. *Profile* describes the capability of a service complete with information about its input messages, output messages, preconditions and effects. *Process model* describes in detail what the service does in terms of atomic processes, and composite processes that are formed by combining other atomic and/or composite processes. And, *grounding* describes details of how to access the service, including details of communication protocols, serialization techniques etc. The grounding information usually maps the service-access related information to the corresponding WSDL description of the service. In this sense, OWL-S can be viewed as building on top of

[1] It should be noted that once the required software resources are discovered, their references are passed to the application. It is up to the application on how it interacts with these resources. However, an application may or may not delegate the software resource discovery to CASA. For example, if the software resource discovery involves complex negotiations and establishing a SLA (service level agreement), then the application will do the discovery and selection on its own rather than delegating it to CASA.

WSDL, i.e. providing additional semantic information that cannot be provided using WSDL. For more details on OWL-S specifications, please refer to [OS07]. A service requestor needs to provide only an OWL-S profile, while a service provider should provide at least the grounding information in addition to the profile description.

More details on monitoring and managing resources are given in Chapter 3.

2.3.3 Adapting applications

Application adaptation can be realized using one or more of the following adaptation mechanisms supported by CASA, depending on the adaptation needs of a specific application.

Dynamic change of lower-level services: As discussed in Chapter 1, lower-level services are generic services provided by the deployment platform and usually managed by a middleware system. Software applications rely on these services for their execution. Since these services are generic and can be shared among different applications, these are not implemented as a part of the application code. Examples of such services are data encryption, compression, transmission, caching, video coding/decoding etc.

An application adaptation through a dynamic change of lower-level services is typically required in response to a change in the availability of resources, though it can also be required in response to a change in contextual information.

Common examples of such an adaptation exhibit some form of resource vs. resource tradeoff or resource vs. quality of service tradeoff. An example of a resource vs. resource tradeoff is to compress the data being transmitted over a communication channel by invoking a lower-level lossless data compression service, in response to a drop in the bandwidth available on the channel. Compressing the data will result in increased CPU consumption, but at the same time it will result in saving precious bandwidth.[2] Note that the quality of service (here, the quality of data) is not affected by this adaptation. An example of a resource vs. quality of service tradeoff is to reduce the frame-rate or resolution of the video being transmitted over a communication channel by adapting the lower-level video coding/decoding service,

[2]In fact, compressing the data also requires increased battery consumption. However, at the same time, there is a saving in the battery consumption during transmission of the compressed data as compared to the uncompressed data.

in response to a drop in the available bandwidth.

Several adaptive middleware systems have been developed that are capable of adapting an application by dynamically changing the lower-level services used by the application. Some of these middleware systems are reflective in nature, i.e. support external regulation of their adaptation strategy. Any of the reflection-based adaptive middleware can be integrated with CASA for carrying out dynamic adaptation of lower-level services. CASA can even be integrated with more than one adaptive middleware system offering different adaptation capabilities.

The Contract Enforcement System (CES) in CASA is responsible for interacting with the underlying adaptive middleware system. The reflective nature of the adaptive middleware system implies that the CES can instruct the middleware to carry out the required changes in lower-level services. This includes activating or deactivating some services, or changing certain parameters of some of the running services. More details on dynamically adapting lower-level services are given in Chapter 4.

Many of the adaptive middleware systems also have a built-in resource monitoring service for monitoring local and network resources. Thus, an adaptive middleware system can serve the dual purpose of monitoring local and network resources, in addition to dynamically changing lower-level services.

However, an application adaptation through a dynamic change of lower-level services is applicable only in limited scenarios – limited kinds of applications, execution environment conditions, and the amount of variations in the execution environment. In order to deal with the other scenarios, one or more of the following adaptation mechanisms may be used.

Dynamic weaving and unweaving of aspects: AOP (aspect-oriented programming) [KLM+97] enables separating the crosscutting functionality of an application from its core functionality. Many times a change in the execution environment requires a corresponding change in the crosscutting functionality of an application, without really affecting the core functionality of the application. Thus, the ability to dynamically weave and unweave aspects into / from an application, generally referred as dynamic AOP, presents a powerful adaptation mechanism.

As an example, consider an application using an access control aspect. If the node hosting the application moves from a low-risk environment to a high-risk environment, the access control aspect of the application should be adapted accordingly

to enforce a stronger security behavior (i.e. by replacing the current aspect with a new aspect that implements a more restrictive access control policy). As another example, consider an application using a persistence management aspect that relies on a remote data storage. If the connection to the remote data storage is lost, the persistence management aspect of the application should be adapted accordingly (e.g. by replacing the current aspect with a new aspect that relies on the local storage only).

For dynamic weaving and unweaving of aspects, CASA relies on a dynamic AOP system called PROSE [NA05], which is a flexible and efficient Java-based system developed for this purpose.

The Aspects Adaptation System (AAS) in CASA is responsible for interacting with PROSE. The AAS can pass the appropriate aspect details (name and location of the aspect file) to PROSE at runtime for dynamically weaving the aspect into an application. The aspect file contains the information about the join-points (the execution points where the aspect needs to be weaved) as well as the actual aspect code to be executed at these points. PROSE is able to intercept the join-points in a running Java application, and invoke the corresponding aspect code at these points. At the end of the execution of the aspect code, the control is returned to the next execution point in the application. Similarly, the AAS can instruct PROSE to unweave an already weaved aspect. More details on dynamically changing application aspects in CASA are given in Chapter 4.

However, if a change in the core functionality of an application is required in response to a change in the execution environment, rather than in the crosscutting one, then the following adaptation mechanisms may be used.

Dynamic recomposition of application components: Modern software applications are composed of components, where each component implements a subtask of the application. In a component-based application development, the components encapsulate their implementation details and interact with each other only through their well-defined interfaces.

This makes it possible and convenient to dynamically change the core functionality of an application through dynamic recomposition of application components. A dynamic recomposition of application components involves adding, removing or replacing the components at runtime. A change in the core functionality is most

likely required in response to a change in contextual information, but may also be required for significant variations in resource availability [MG05b].

For example, if the contextual information related to a *Tourist-Guide* application changes from *shopping mall* to *open-air cinema*, the application needs to provide relevant information about the weather conditions and show-timings, in place of the information about the availability of the items from the tourist's shopping wish-list in the shopping mall. This kind of a change in the core functionality can be accomplished through dynamic recomposition of application components. Needless to say that the possibilities for application adaptation through dynamic recomposition of application components are enormous.

In the last few years, a number of approaches have been proposed for dynamic recomposition of application components. However, these approaches are not able to adapt the currently running components, without discarding their current execution. That is, these approaches require that either a component is not running at the time of adaptation, or its current execution is discarded as a result of adaptation. Moreover, many of these approaches are restricted to using specific programming languages or platforms. More discussion on the related approaches for dynamic recomposition of components is provided in Chapter 7. Due to the lack of flexibility offered by the currently available approaches, we have developed our own approach for dynamic recomposition of components in CASA.

The Components Adaptation System (CAS) in CASA is responsible for dynamic recomposition of application components. That is, the CAS can add, remove or replace the application components dynamically. The CAS takes care to ensure that the consistency of the application is not compromised as a result of dynamic recomposition. Ensuring the consistency involves, among other things, transferring the state of an outgoing component to its successor (in case of a dynamic replacement), and maintaining the integrity of the interactions ongoing at the time of dynamic recomposition. The components recomposition approach developed in CASA is generic enough to be used with any modern object-oriented programming language. More detail on this approach are given in Chapter 5.

However, sometimes only a dynamic change of certain attributes of an application may be required, rather than a dynamic recomposition of application components. For a dynamic change of application attributes, the following adaptation mechanism may be used.

Dynamic change of application attributes: An application attribute here refers to a variable that is part of the application code. This variable can be encoded within an application component or in a configuration file. A runtime change of an application attribute, therefore, implies changing the value of the corresponding variable. A dynamic change of attribute can effect a change in the application behavior, thereby allowing the application to adapt to a change in the execution environment.

Examples of application adaptation through a dynamic change of application attributes include changing the value of a certain timeout period, frequency of data transmission, size of each transmission, or some other threshold parameters affecting an application's behavior in response to a change in the execution environment. It is assumed here that the above parameters are encoded as variables in the application code, whose values can be changed at runtime. Changing the value of a variable results in a change in the application behavior. For instance, reducing the frequency of data transmission by changing the value of the corresponding variable is likely to result in reducing the throughput of the application. A reduction in the throughput allows the application to successfully adapt to a drop in the availability of communication bandwidth.

For a dynamic change of application attributes, the concerned application needs to provide appropriate callback methods that can be called by the CES at runtime. That is, the ultimate responsibility of changing the application attributes lies with the application itself. This allows the application to decide the appropriate timing and order of changing these attributes.

In addition to the above adaptation mechanisms, certain resource-level adaptations may be carried out transparently by the underlying adaptive middleware or the SRC (Software Resource Coordinator). Such adaptations usually involve dynamically selecting an alternative resource among those available, in response to a loss of certain resource. For example, if the availability of a remote resource being used by an application drops, the SRC may dynamically switch to an alternative resource providing the same type of service but with better availability. Or, if the bandwidth on a certain communication channel drops, the adaptive middleware may switch the data transmission to another communication channel with better bandwidth (assuming that alternative communication channels are available for the transmission of data).

The overall working of the CASA framework is described in Section 2.5. In the

next section, the specification of an application contract is discussed.

2.4 Application contract specifications

The adaptation policy of every application is defined in a so-called application contract. The application contract is external to the application, and is specified using an XML-based language. The contract-based adaptation policy facilitates easy modification, extension and customization of the adaptation policy at runtime, besides playing a key role in separating the adaptation concerns of an application from its business concerns.

An excerpt of the application contract of a *Tourist-Guide* application is shown in Figure 2.3. The complete XML schema for application contracts is given in Appendix A.

An application contract is divided into <`context`> elements, where each <`context`> element represents a state of contextual information of interest to the application. The parameters characterizing this state are specified within the <`params`> element, which is a sub-element of <`context`> element. As discussed in the next section, the <`context`> elements are ordered within the application contract according to their user-perceived preference.

The <`params`> element has an attribute called `ontology`, which contains a reference to the corresponding OWL ontology for the given application domain. In addition, the <`params`> element contains one or more <`par`> elements, with each <`par`> element corresponding to a distinct context parameter. Every <`par`> element contains a `name` attribute specifying the name of the context parameter, an optional `unit` attribute specifying the unit of measurement, a `value` attribute (for non-numeric parameter values) specifying the corresponding value of the parameter, `minv` and `maxv` attributes (for numeric parameter values) specifying the lower and upper bounds of the corresponding range of parameter values, and an `enum` attribute (for enumerated values of a parameter, separated by commas) specifying the corresponding enumeration of values. For the non-numeric parameter values specified using a `value` attribute, an exact match is required. For the numeric parameter values, the monitored value must be within the range specified by the `minv` and `maxv` attributes. And for the values specified using an `enum` attribute, the monitored value must match at least one of the enumerated values, in order for the corresponding

```
<app-contract name="Tourist-Guide">
    <context id="1">
        <params ontology="tourist.owl">
            <par name="vicinity" value="museum"/>
        </params>
        <config id="1">
            <resources>
                <hw name="bandwidth" unit="kbps" mpv="200" lpv="50"
                    reference="homeserver"/>
                ...
                <sw name="museum-info" reference="/sw-folder/mi.owl"/>
                ...
            </resources>
            <components>
                <binding handle="HC" boundto="/class-folder/CdefA"/>
                ...
            </components>
            <aspects>
                <aspect name="access-control" reference="/aspect-folder/AC"/>
                ...
            </aspects>
            <callbacks>
                <call method="upcall-1">
                    <arg value="5" type="int"/>
                </call>
                ...
            </callbacks>
            <llservices>
                <lls manager="Odyssey" name="video" operation="tsop">
                    <arg value="/VideoWarden/QT_Movie" type="string"/>
                    <arg value="QT_SwitchTracks" type="string"/>
                    <arg value="5" type="int"/>
                    <arg value="2" type="int"/>
                </lls>
                ...
            </llservices>
        </config>
        <config id="2">
            ...
            ...
        </config>
        ...
        ...
    </context>
    ...
    ...
</app-contract>
```

Figure 2.3: Application Contract

parameter to be considered valid in the given state of execution environment.

The names of context parameters are standard and unique for the corresponding application domain. The parameter names and their possible values are specified in the corresponding application domain-specific ontology. Examples of the parameter names for the *Tourist-Guide* application are `vicinity` (referring to a place of interest nearby), `time` (referring to the time of day) etc.

There may be a default <`context`> element in an application contract that is valid when none of the other <`context`> elements is valid. This default <`context`> element is listed at the end of the application contract, and does not contain a <`params`> element.

Each <`context`> element further contains a list of alternative configurations of the application, which are suitable for activation in the corresponding state of contextual information. Please note that there must be at least one configuration listed within a <`context`> element.

The alternative configurations within a <`context`> element are listed in an ordering that reflects their user-perceived preference. These configurations vary in their resource requirements, and most likely in their behavior (functionality / performance). Each of the alternative configurations is specified within a <`config`> element, which is a sub-element of <`context`> element.

A <`config`> element, representing a configuration, specifies the resource requirements of the configuration (in <`resources`> sub-element), the components and aspects constituting the configuration (in <`components`> and <`aspects`> sub-elements), the callback methods to be called for changing the application attributes for the configuration (in <`callbacks`> sub-element), and the lower-level services participating in the configuration (in <`llservices`> sub-element).

The <`resources`> element may contain any number of <`hw`> and <`sw`> elements. A <`hw`> element represents a hardware resource, and a <`sw`> element represents a software service that is part of the corresponding application configuration.

Every <`hw`> element contains a `name` attribute specifying the name of resource, an optional `unit` attribute specifying the unit of measurement, a `mpv` attribute specifying the most preferred value, a `lpv` attribute specifying the least preferred value of the corresponding resource, and an optional `reference` attribute specifying the reference item (e.g. in case of network bandwidth, the `reference` attribute will contain

the referred destination). We have assumed here that all hardware resources are numerically quantifiable (however, the schema of application contract allows non-numeric values to be specified in a `value` attribute, and an enumeration of values to be specified in an `enum` attribute). The range of numbers between `mpv` and `lpv` are assumed to be linearly distributed. Both `mpv` and `lpv` attributes might have special values denoted as `max` or `min`. A `max` value specifies the maximum possible value for the resource, and a `min` value specifies the minimum possible value for the resource under the given execution environment conditions. That is, these two special values denote relative values for a resource rather than absolute ones. A <`hw`> element can have an optional attribute called `essential` with a possible value of `yes` or `no` (default is `yes`), specifying whether this resource is essential for the application configuration or not.

The resource names for hardware resources are standard and unique, i.e. every hardware resource can be uniquely identified by a standard resource name. Resource requirements for hardware resources can also be specified at a high level of abstraction such as in terms of throughput and packet size, instead of directly specifying these in terms of actual resources such as bandwidth. If the resource requirements are specified at a high level, then the RM needs to convert these into actual concrete resource values to be allocated.

A <`sw`> element has a `name` attribute specifying the name of the software service and a `reference` attribute containing a reference to the OWL-S profile of the service. The name of the service is used for internal reference only, while the description of the required service is provided in the OWL-S profile referred by the `reference` attribute. The OWL-S profile referred in the <`sw`> element is used by the SRC (Software Resource Coordinator) for discovering the required matching services. As will be discussed later in Chapter 3, the SRC relies on UDDI enhanced with OWL-S for service discovery [SPS04], and the OWL-S API matchmaker for service matchmaking [API07]. A <`sw`> element can also have an optional attribute called `essential`, to specify whether this resource is essential for the application configuration or not.

The <`components`> element contains a list of adaptable application components, i.e. those components that may differ from one configuration to another. The non-adaptable components, which remain the same across all configurations, are not specified in the <`components`> element. In the same way, the <`aspects`> element

contains a list of adaptable aspects. Every component and aspect is specified along with its namespace location, as this is required by the CRS (CASA Runtime System) for activating a configuration. Similarly, the <callbacks> element contains a list of application methods to be called, and the <llservices> element contains a list of lower-level services corresponding to the configuration. Further details of <components>, <aspects> and <llservices> elements are given in the following chapters, along with the descriptions of their respective adaptation mechanisms.

A <callbacks> element contains one or more <call> elements, with each <call> element corresponding to a distinct application method to be called. A <call> element contains a method attribute specifying the method to be called, and a number of <arg> sub-elements specifying the values and types of the arguments to be passed with the method call. There are no return values associated with such method calls.

All the elements specifying the constituents of a configuration (i.e. the <components>, <aspects>, <callbacks> and <llservices> elements) are optional. That is, any of these elements may or may not appear in a configuration specification, depending on the adaptation requirements of the corresponding application. For example, if an application has no adaptable components, but only adaptable aspects, attributes and lower-level services, then the <components> element will not appear in the specification of an application configuration.

Similarly, if an application needs to respond only to the changes in contextual information, but not to the changes in resource availability, then the <resources> element can be omitted from the configuration specification (e.g. if all the resources required by an application are guaranteed to be available sufficiently). In this case, every <context> element may contain only a single configuration suited to the corresponding state of contextual information, as the availability of the constituents of a configuration (components, aspects etc.) is usually guaranteed.

On the other hand, if an application needs to respond only to the changes in resource availability, but not to the changes in contextual information, then there will be a single default <context> element in the application contract. That is, the application contract will contain a simple listing of the alternative configurations of the application (<config> elements), ordered according to their user-perceived preference, within a single <context> element.

Depending on the current state of the execution environment (contextual infor-

mation and resources), the appropriate configuration from the application contract is selected and activated by the CRS, as explained in the next section.

2.5 Working of the CASA framework

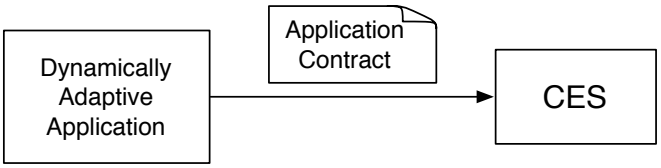
2.5.1 Initial activation of an application

In the beginning, when an application wants to start its execution, it registers itself with the CRS. As a part of the registration, the application contract of the application is sent to the CES (Contract Enforcement System) (Figure 2.4a).

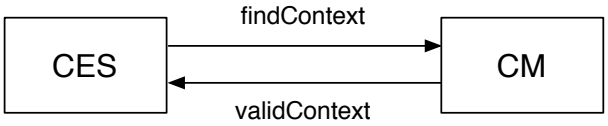
If the application contract contains a <context> element, other than the default <context> element (i.e. the one without a <params> element), the CES invokes the CM (Context Monitor) passing a reference to the application contract (Figure 2.4b). The CM monitors the values for contextual parameters defined in the application contract, in order to determine the current contextual information relevant to the application. The contextual information monitored by the CM is matched with the <params> elements specified in the application contract, for deciding the currently valid <context> element. This is done by matching the monitored values of contextual parameters with the corresponding values specified within the <params> element of every <context> element in the application contract.

It is possible that more than one <context> element specified in the application contract is eligible to be valid simultaneously. For example, one <context> element may be valid if the user is currently in her office-building, and the other may be valid if the user is currently in her office-room. So, if the user is currently in her office-room, then both the above <context> elements are eligible to be valid at the same time. In this case, the ordering of the <context> elements in the application contract is important for deciding the currently valid <context> element, as this ordering reflects the relative preference of the <context> elements. That is, the CM identifies the highest listed valid <context> element as the currently valid <context> element. Practically, the CM starts searching for the currently valid <context> element from the top of the application contract, and the search ends as soon as a valid <context> element is found.

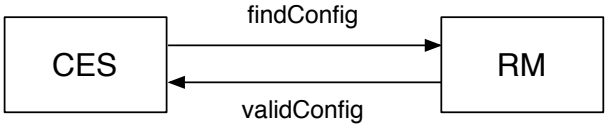
If there are certain contextual parameters in the application contract that cannot be monitored by the CM, it raises an exception to the CES communicating the parameters that cannot be monitored. The CES in turn communicates these



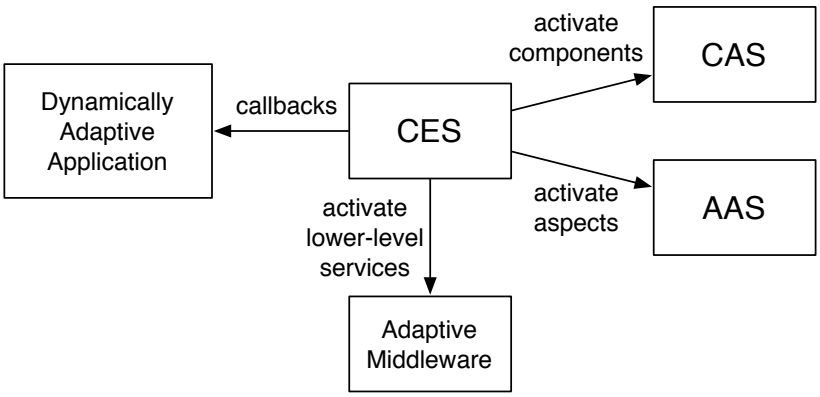
a. Application contract sent to the CES



b. Determining the valid context



c. Determining the valid configuration



d. Activating the selected configuration

Figure 2.4: Working of the CASA framework

parameters to the concerned application. The application may either ignore these parameters (i.e. remove these parameters from the application contract and resubmit the contract to the CES), or choose not to execute as the current configuration of the CASA framework is not compatible with the application's requirements.

Similarly, it is possible that none of the <context> elements specified in the application contract is currently valid (assuming that no default <context> element is specified in the contract). In this case, the application cannot begin its execution, and the CES communicates this failure to the application.

Once the current contextual information relevant to the application is successfully determined, the CM communicates the identifier of the currently valid <context> element to the CES (Figure 2.4b).

Recall that every <context> element in the application contract contains a list of alternative configurations, represented by <config> elements, which are suitable for the corresponding state of contextual information defined by the <context> element. If the <config> elements listed within the currently valid <context> element contain <resources> elements, then the CES invokes the RM (Resource Manager) passing a reference to the application contract and the identifier of the <context> element (Figure 2.4c). As with the ordering of the <context> elements, the <config> elements are also preferentially ordered within every <context> element in the application contract.

The RM is responsible for allocating resources to the application according to the current availability of resources. Since the configurations are listed in a preferential ordering within the <context> element, the RM tries to allocate resources for the first configuration listed in the <context> element. If there are not sufficient resources for the first configuration then it tries for the second configuration, and so on. The RM relies on the underlying resource monitoring service and the SRC (Software Resource Coordinator) for the discovery of required resources (as discussed earlier in Section 2.3). The resource allocation phase ends as soon as resources for a configuration can be successfully allocated, given the current availability of resources. This configuration is then selected for activation.

In case of multiple applications competing for the same resources, the RM takes into account the relative priorities (as defined by the user) as well as the adaptation possibilities of the applications for allocating resources. The details of the resource allocation algorithm followed by the RM are given in Chapter 3.

If none of the configurations listed in the current <context> element can be allocated the required resources (due to the unavailability of resources), the RM communicates the failure to the CES. The CES reinvokes the CM, asking it to determine all currently valid <context> elements. The list of all currently valid <context> elements returned by the CM is then forwarded to the RM by the CES. This list is ordered in the same order as the original order in the application contract, i.e. in the user-perceived order of preference. The RM carries out the resource allocation procedure for every <context> element in the list, until a configuration from one of the <context> elements can be successfully allocated its required resources.

In practice, the chances of successful allocation of one of the configurations from the most preferred <context> element are likely to be high. On the other hand, the chances of an application having more than one valid <context> element at the same time are likely to be low. Therefore, a re-run of context determination and resource allocation phase, as discussed in the above paragraph, is likely to be very rare.

However, if no configuration of the application can be allocated its required resources at the end of the complete resource allocation phase, then the application, obviously, cannot begin its execution. This failure is communicated by the CES to the application.

Once the resources for a configuration are allocated successfully, the RM communicates the selected configuration to the CES (Figure 2.4c). The CES in turn communicates the selected configuration and context to the application for its confirmation.

Once the application sends its confirmation, the CES instructs the CAS (Components Adaptation System), the AAS (Aspects Adaptation System) and the underlying adaptive middleware to activate the components, aspects and lower-level services corresponding to the selected configuration respectively (Figure 2.4d). The CES also issues any callbacks specified for this configuration. Recall that the components, aspects, lower-level services and callbacks corresponding to the selected configuration are specified within the <config> element describing the configuration. The application can now begin its execution.

The process of getting confirmation from the application before activating the selected configuration is usually straightforward, as the application is expected to agree to the decision of the CES. This is because the decision of the CES is based on

the adaptation policy specified by the application itself in its application contract. However, the process of getting confirmation from the application, before activating the configuration, allows the application to perform initialization operations. This might include, for example, informing the user that the corresponding service will be available soon. In some rare cases, the application might even have to refuse the configuration because of any failure of its initialization operations. For instance, the user might opt not to execute the service at the last moment. In this case, the resources already allocated to the application are freed, and the activation of the selected configuration is aborted.

2.5.2 Runtime changes in the execution environment

Runtime changes in the execution environment can be in the form of changes in contextual information or resource availability. If there is a runtime change in the contextual information relevant to the application, such that either the currently active <context> element is no longer valid or one of the <context> elements listed higher than the currently active <context> element becomes valid, then this change is detected by the CM. The CM communicates the new <context> element to the CES. The CES invokes the RM (assuming that the configurations listed within the new <context> element contain <resources> elements) passing the identifier of the new <context> element. The RM allocates resources for a new configuration from this <context> element, following the same procedure as during the initial allocation, and communicates the new selected configuration to the CES.

Similarly, if there is a runtime change in the availability of resources, but not a change in the contextual information, then the RM is informed about the change by the underlying resource monitoring service or the SRC. If the new resource availability makes it impossible for the application to continue with its current configuration, then the RM allocates resources for a new configuration from the current <context> element, based on the changed availability of resources. Similarly, if the new resource availability allows a new configuration listed higher than the currently active configuration within the current <context> element to be activated, then the RM allocates resources for this configuration. The resource allocation algorithm followed by the RM is discussed in Chapter 3. The RM communicates the new selected configuration to the CES.

Any failure to resource allocation (i.e. no configuration can be selected due to

unavailability of resources) or context determination (i.e. no context specified in the application contract is currently valid) is communicated by the CES to the application. Such a failure implies that the application cannot continue with its execution. The application needs to implement appropriate failure handling mechanisms for dealing with such a failure.

Once a new configuration is selected by the RM and communicated to the CES, the CES communicates the change in configuration to the application for its approval. Once the application confirms the change, the CES instructs appropriate entities of the CASA framework for carrying out a change from the old configuration to the new configuration, and issues any callbacks as required. A change in configuration may imply a change in application components, aspects, lower-level services and/or application attributes. Further details of a dynamic change in an application's configuration are discussed in the following chapters.

However, if the application does not accept the change in configuration (less likely because the change is decided in accordance with the adaptation policy of the application; but still possible due to a last-minute change in external factors such as user's preference), then it may not be able to continue with its execution. In this case, the application needs to carry out finalization operations, and the resources allocated to the application are freed.

2.5.3 Dynamic changes in adaptation policy

Customization of adaptation policy: As seen from the above description of the working of CASA, the ordering of <context> elements within an application contract, as well as the ordering of <config> elements within a <context> element, influence the adaptation policy of an application to a large extent.

The ordering of these elements can be changed by the user at runtime. In addition to changing the order, the user can also remove certain <context> or <config> elements at runtime. This way the user is able to customize the adaptation policy of an application according to her needs or preferences.

For example, in the *Tourist-Guide* application, if the user is not interested in the *museum* context, but only interested in the *shopping mall* and *open-air cinema* contexts, then she may remove the *museum* context from the application contract.

For customizing the adaptation policy, a graphical user interface for the application contract should be provided by the application developer, which ex-

plains the significance of the various <context> and <config> elements in user-understandable terms. That is, instead of displaying the list of parameters characterizing a <context> element, it displays what the particular state of contextual information means for the user. And instead of displaying the detailed constituents of a <config> element, it displays the appropriate functionality and performance characteristics associated with the corresponding configuration.

Evolution of adaptation policy: For the evolution of adaptation policy, any of the elements in the application contract (e.g. <context> or <config> elements, or any of their sub-elements) can be modified at runtime. Similarly, new <context> or <config> elements can be added to an application contract (which were not foreseen at the time of application development), and any obsolete <context> or <config> elements can be removed from an application contract at runtime.

2.6 Service negotiations

In the design of the CASA framework, we have assumed that applications are completely autonomous, and have no a-priori knowledge of other applications that might be available for interactions. Different autonomous applications may discover each other only at runtime, and decide to interact in an ad-hoc manner. In this way, the interacting applications may form a distributed software system at runtime, and collaborate with each other for a common task.

In view of the autonomous nature of applications, each application has its own individual adaptation policy defined in its own application contract. That is, there is no centralized adaptation policy for the whole distributed software system, as each application is considered to be independent in defining its own policy and requirements.

However, in a distributed software system, peer applications may need to be consulted before an adaptation decision is taken by an application, as the interests of different interacting applications can be interdependent. This implies that different peer applications participating in a distributed software system should be able to carry out service negotiations among themselves. CASA provides support for service negotiations among peer applications.

For carrying out service negotiations, the concerned application needs to implement a Service Coordinator (SC) component. In addition, the application developer

needs to provide a service description for every alternative configuration defined in the application contract. A service description specifies the service capability of the corresponding configuration. The service description of a configuration is defined in a separate document called the Service Description Document, which is accessible to the SC.

The Service Description Document is defined using OWL-S [OS07]. Even though the application may not be accessible as a Web service (i.e. it need not communicate using SOAP, or be advertised on a UDDI registry etc.), the OWL-S specification language provides a standard format for describing a service, which is useful for exchanging service descriptions among peer applications.

In the following, we describe the changes in the working of CASA framework that are implied by the inclusion of service negotiations. We discuss simple point-to-point service negotiations in Section 2.6.1, and complex negotiations in Section 2.6.2.

2.6.1 Simple negotiations

When service negotiations are involved, the resource allocation phase does not end as soon as a match between the resource requirements of a preferred configuration and the current availability of resources is found. Rather, the RM identifies all the application configurations that can be activated in the current availability of resources. The RM communicates the identified configurations to the CES. The CES passes the list of identified configurations to the SC.

If there was no need for a resource allocation phase (i.e. alternative configurations did not specify any resources requirements in the application contract), then the CES simply communicates all the alternative configurations that can be activated in the current execution environment to the SC.

The SC sends the Service Description Documents associated with each of the identified configurations to the peer applications that are affected by the change, i.e. the peer applications that were using the service provided by the adapting application (refer Figure 2.5). The peer applications are then required to rank these configurations based on the service descriptions, and send the ranked list back to the SC. The SC then selects the most appropriate configuration for activation, based on the rankings given by the peer applications, and communicates its decision to the peer applications and to the CES.

An application might assign different weights to different peer applications, so

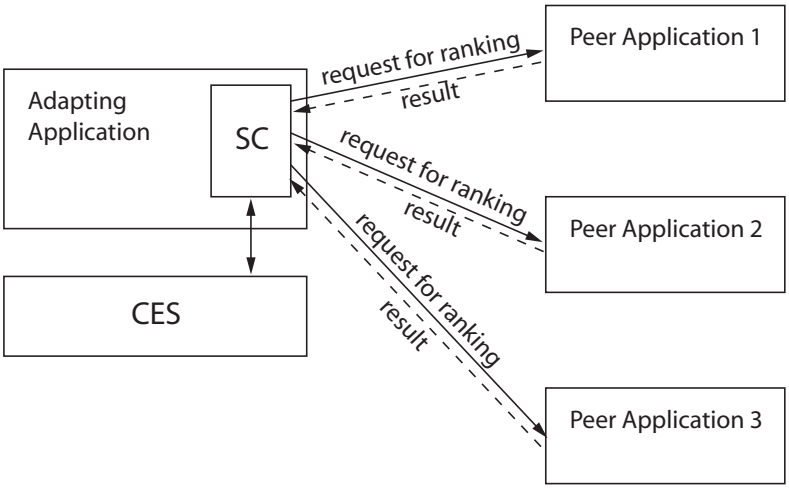


Figure 2.5: Simple point-to-point service negotiations

that the rankings by these peer applications are treated accordingly for the final selection of a configuration. Any ties for the top-ranked configuration are resolved by selecting the configuration listed highest in the application contract.

The peer applications need not be developed according to the CASA framework, but they must provide a component for receiving a list of alternative configurations and ranking them. If a peer application also needs to be adapted due to a change in the configuration of the above application, then the ranking is decided according to the relative preferences of the corresponding adaptations of the peer application itself for each of these alternative configurations.

If a peer application is developed according to the CASA framework, then this peer application might be using the service provided by the above adapting application as a required software resource discovered by the SRC (irrespective of whether the discovery was made using UDDI enhanced with OWL-S, or by any other means). Though the above situation is likely, it is not necessary because an application may not delegate all its software resource requirements to the CASA framework. For instance, if the selection of a software service requires complex negotiations (possibly involving a human user) then the application may discover and select the service providers itself, rather than delegating this job to the underlying CASA framework.

However, if the above adapting application is being used as a software resource that was discovered by the SRC of the peer application, then the list of alternative

configurations of the above application is forwarded to the SRC. The SRC passes this list to the RM of the peer application, which then evaluates the corresponding adaptation actions. Inclusion of different alternative configurations from the list results in different resource availabilities from the RM perspective. Based on the relative preferences of these adaptation decisions, i.e. relative preferences of the resulting configurations of the peer application, the RM ranks the list and sends it to the SRC. The SRC sends the ranked list back to the SC of the above adapting application.

As an example where service negotiations may be required, consider an application transmitting high-quality multimedia (video + audio) to a number of clients. In response to a drop in the available bandwidth, the application may have an option either to switch to a configuration that reduces the quality of audio but maintains the high quality of video, or to a configuration that reduces the quality of video but keeps the quality of audio high. Before actually changing its configuration, the application needs to carry out service negotiations with the clients. The clients' decision, on the other hand, may be governed by the actual content of the transmission. For instance, if the transmission is that of a soccer match, the clients may choose to reduce the quality of audio while maintaining the high quality of video. Whereas, if the transmission is that of a musical concert, the clients may choose to reduce the quality of video without disturbing the quality of audio.

However, service negotiations are not essential for all kinds of applications that participate in distributed software systems. For instance, if in the above example the application transmits sports events only, then it may have the default adaptation behavior of switching to the first kind of configuration, without requiring any service negotiations with its clients.

At the time of registering with the CRS, an application needs to indicate whether service negotiations are required, and pass a reference of its SC component accordingly.

2.6.2 Complex negotiations

Section 2.6.1 describes simple service negotiations involving point-to-point negotiations between the adapting application and its peer applications. In most of the cases occurring in practice, simple service negotiations will be sufficient. However, the above description of simple negotiations does not address some of the complexi-

ties that might arise when service negotiations are more complicated than point-to-point negotiations. Below we present some of these complexities and the possible solutions. The issue of complex negotiations requires more in-depth research, and is considered as one of the directions of work that should be explored in future.

Let us assume that an application A sends its list of alternative configurations for ranking to a peer application B. The application B, which is a client of A (i.e. consuming the service provided by A), might need to be adapted as a result of the adaptation of A. Adaptation of B might imply a change in the service provided by B, and therefore B might need to carry out negotiations with its own clients before informing its ranking decision to A. This should not be a problem in general, because B can finish its own negotiations first before informing its decision to A. However, the chain of negotiations might be long, and might even have cycles. That is, what happens if B needs to negotiate with its client C, but C itself needs to negotiate with its client A (refer Figure 2.6)? A may not be able to give a definite answer to C, as it is still waiting for the result of its negotiations with B. To break this kind of deadlock, every application might set a timeout period after which it takes a decision based on the already received responses, and simply communicates its final decision to the clients. This may not result in the best solution for all the applications involved, but it offers a way out of deadlock.

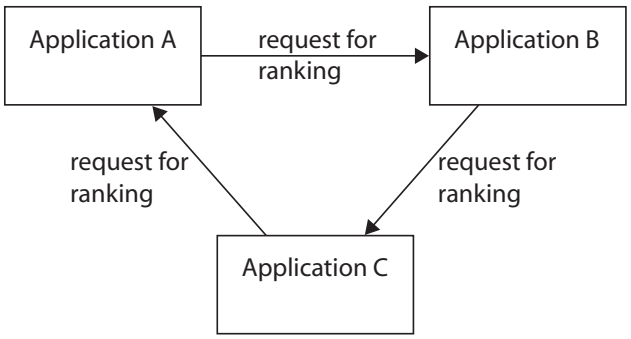


Figure 2.6: Cycle of service negotiations

The problem gets more complex if in the above situation of a cycle between A-B-C, the options given by C to A may require a change in the list of alternative configurations that A had submitted to B. This would require renegotiations between A and B, and possibly have a spiraling effect on the negotiations between B

and C, and so on.

Similarly, application A might have two clients B and C to whom it submits its list of alternative configurations for ranking. B in turn might be providing a service to C, i.e. C might be a client to both A and B. Consequently, C might get a list of alternative configurations from A as well as B (refer Figure 2.7). C might rank these two lists independently according to its best interest. However, since none of the applications have a global picture of the negotiations taking place, the independent rankings by C may not serve its best interest. That is, even if rankings of A by C are in the best interest of C, rankings of B by C might result in a situation that B gives different rankings to A than what C had given to A. If B is a more important client for A, rankings of B will be approved by A. This will result in a situation that the service provided by A will not be the best one for C, and another combination of rankings might have been a better option for C. In general, an application can be faced with ranking different lists that should not be treated in isolation. Different lists present different ranking combinations to the application. However, in the absence of a global view of negotiations taking place, the decisions taken by the application might be sub-optimal.

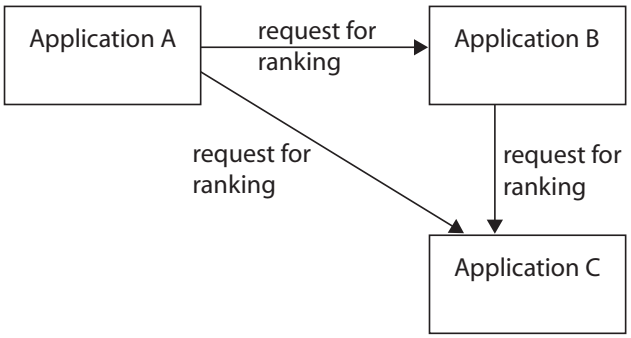


Figure 2.7: Concurrent rankings in service negotiations

The issue of multi-party service negotiations can indeed be very complex, and is a part of the future work.

2.7 Discussion

2.7.1 Capabilities and limitations of the CASA framework

In this chapter, we have presented the overall design and working of the CASA framework. As seen from the description above, the CASA framework is able to comprehensively meet a wide range of adaptation needs of software applications, thanks to its ability to adapt any parameter of an application's configuration. The design decisions taken in the CASA framework (independent and reusable adaptation infrastructure, and contract-based adaptation policy) enable separating the adaptation concerns of an application from its business concerns, thereby considerably reducing the complexity involved in developing and maintaining dynamically adaptive applications. The contract-based adaptation policy further allows modifying the adaptation policy at runtime.

The design of the CASA framework is very modular with only a loose coupling between various entities of the framework. The loosely coupled design allows the entities of the CASA framework to evolve independent of each other. This would help in easily replacing any of the existing monitoring or adaptation mechanisms with their improved versions in the future. For example, the existing context or resource monitoring mechanisms can be replaced with more efficient or effective mechanisms in future. Similarly, a new improved adaptive middleware can replace an existing adaptive middleware, or new and better components or aspects adaptation approaches can be used in place of the existing ones.

The CASA framework relies on externally developed context sensing and resource monitoring mechanisms. Therefore, the ability to monitor different parameters of the execution environment, and the efficiency with which these parameters can be monitored depend on the functionality and performance offered by these external mechanisms. However, this is considered a practical approach in the design of the framework, as the third-party monitoring mechanisms are specialized for the purpose and therefore offer reasonably good functionality and performance. Similarly, the adaptation of lower-level services and aspects is delegated to the external mechanisms, which are very well-suited for the purpose. For dynamic recomposition of components, CASA follows an indigenously developed approach, as none of the currently available approaches offer the required flexibility for recomposing components at runtime.

It is assumed in the design of the CASA framework that the correctness and completeness of the application contract defining the adaptation policy of an application is ensured by the application developer. We currently do not provide any tools to aid the application developer in ensuring the correctness and completeness of the application contract. However, this is considered an important part of the future work.

When changing the configuration of an application at runtime, it is assumed that the application components, aspects and lower-level services required by the new configuration are available and accessible to the CASA Runtime System. The location of the components and aspects to be added is specified in the application contract. These components and aspects can be available either locally, or be downloaded on-the-fly from a remote location. The procedures to activate or deactivate any lower-level services, or to change the parameters of some currently running services are defined in the application contract. Obviously, the adaptation of the lower-level services should be supported by the underlying adaptive middleware. Similarly, any callback methods required by the new configuration are specified in the application contract, and these should be provided by the application. However, the correctness of the new configuration (i.e. the correctness of the corresponding components, aspects, attributes and lower-level services) for the required application behavior is not verified by the CASA framework. Ensuring this correctness is considered a part of ensuring the correctness and completeness of the overall application contract.

Every adaptation mechanism supported in the CASA framework is individually responsible for ensuring that the change in configuration carried out by this mechanism does not compromise the consistency and integrity of the application. There is no centralized mechanism in CASA for ensuring or verifying the overall consistency and integrity of the application during or after the adaptation process. However, this should not be a problem in practice, as the different adaptation mechanisms are responsible for adapting *orthogonal* parts of an application's configuration. The future work might include developing more stringent mechanisms for ensuring and verifying the consistency and integrity of the application.

Another limitation of the CASA framework is that if a legacy application wants to take advantage of the adaptation capabilities provided by CASA, then it needs to be considerably refactored. This is particularly true for the components recomposition approach followed in CASA, though a legacy application might take advantage

of other adaptation mechanisms with relatively minor changes. The components recomposition approach followed in CASA requires a replaceable component to be accessed through a handler, and to define safe points for its termination. This implies considerable refactoring of the existing components of a legacy application.

2.7.2 Using the CASA framework in practice

Figure 2.8 shows the deployment diagram of the CASA framework being used by an adaptive application. A prototype CASA Runtime System has been implemented in Java. However, the design of the CASA framework presented in this dissertation is generic, and not bound to any specific programming language.

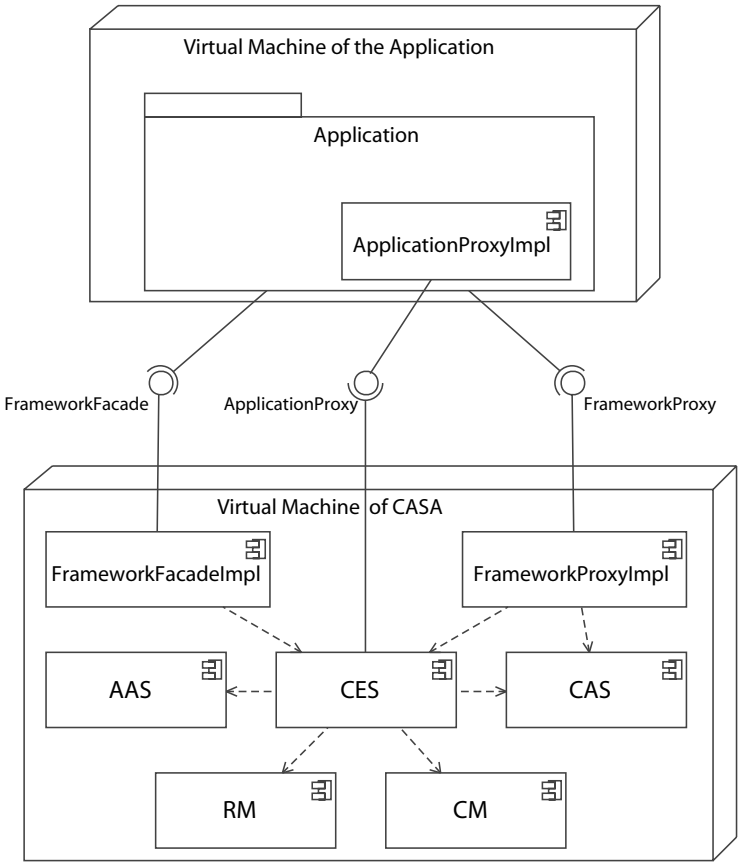


Figure 2.8: Deployment diagram of the CASA framework

As an example, consider the *Tourist-Guide* application described earlier. When

the *Tourist-Guide* application wants to begin its execution, it registers itself with the CASA Runtime System. The registration with the CASA Runtime System is handled by the Framework Façade, as shown in Figure 2.8. This allows exposing only a simplified interface of the CASA Runtime System to an adaptive application. Since the application and the CASA Runtime System are running in different virtual machines, the communication between the *Tourist-Guide* application and the Framework Façade takes place using Java RMI. The Framework Façade can be discovered by the *Tourist-Guide* application using the RMI registry. During the registration, the *Tourist-Guide* application provides an Application Proxy to the CASA Runtime System. The Application Proxy is used by the CES for invoking callback methods on the application. In return, a Framework Proxy is sent to the application, which is used by the application for identification and future communication with CASA. The application contract of the *Tourist-Guide* application is now forwarded to the CES.

Once the registration of the *Tourist-Guide* application with CASA is completed, the CASA Runtime System determines the currently valid configuration of the application and activates this configuration using the process described for initial activation of an application in Section 2.5.1.

Let the current contextual information relevant to the *Tourist-Guide* application be "location = city center" and "time = morning", as discovered by the CM. The most preferred configuration of the *Tourist-Guide* application in the above context requires activating the components that display information about the touristic points-of-interest near the city center, including short videos of these points-of-interest along with audio commentary. At the same time, the above configuration requires a high bandwidth connection to the information server of the department of tourism. Let initially the bandwidth availability be high, and the RM allocates the required bandwidth to the *Tourist-Guide* application. The CES then activates the above configuration by instructing the corresponding entities of the CASA Runtime System, including the CAS to activate the required components.

Runtime changes in the execution environment of the *Tourist-Guide* application are handled by the CASA Runtime System using the adaptation process described in Section 2.5.2. For instance, if the network bandwidth available to the *Tourist-Guide* application drops a little later, this change in discovered by the RM. The RM reallocates resources to the application, and selects a new application configuration

corresponding to the changed availability of resources. The new configuration is communicated to the CES, which instructs other entities for a change in configuration. This change might involve adapting the lower-level video fetching service to fetch low-quality video (e.g. black and white video instead of colored video), while maintaining the good quality of the audio commentary. If the bandwidth drops further a little later, the components displaying video with audio commentary may need to be replaced with components that display only still images with textual description of the touristic points-of-interest. The replacement of above components is carried out by the CAS, at the instructions of the CES.

Assume that after a few hours, the tourist is heading to visit a castle outside the city. Let the new contextual information relevant to the *Tourist-Guide* application be "location = highway-M1" and "time = noon". The change in context is discovered by the CM. The most preferred configuration in the new context requires activating the components that allow searching and browsing information about different restaurants near the highway M1, and booking a table in a selected restaurant. This configuration also requires activating a security aspect that enables secured communication between the application and the restaurant booking server (e.g. by encrypting the credit card details that are required for booking a table). The above configuration requires a resource of restaurant search and booking service to be available. The restaurant search and booking service is discovered by the SRC (Software Resource Coordinator), and its reference is communicated to the application. This means that the above configuration can be selected for activation, as its resource requirements are fulfilled. The CAS and the AAS activate the corresponding components and aspects for this configuration accordingly.

However, if during the browsing of restaurants by the tourist, the connection to the restaurant search and booking service is broken, the application needs to reconfigure itself in order to deal with the loss of resource. The reconfiguration may involve replacing the current components for searching and booking restaurants online with alternative components that display a short information about a few popular restaurants in the area and their phone numbers. This information is assumed to be stored in the local memory of the device hosting the *Tourist-Guide* application. The tourist can use this information for selecting a restaurant from the limited list of restaurants, and calling the selected restaurant to book a table. The security aspect used in the most preferred configuration above can be similarly removed from the application,

as it is not required in the new configuration. These changes in the components and aspects are carried out by the CAS and the AAS accordingly.

The above example shows that the same application may need to adapt to the changes in contextual information as well as changes in resource availability. The adaptation may involve recomposing application components, adding and removing crosscutting aspects, or adapting the lower-level services used by an application. In general, an adaptation decision may affect multiple levels of an application simultaneously, e.g. adapting components, aspects, as well as lower-level services of an application. This example illustrates the versatility offered by the CASA framework (by allowing different parameters of an application's execution environment to be monitored, and different parameters of the application's configuration to be adapted), which allows it to meet a wide range of adaptation needs of software applications executing in dynamic environments.

In the remaining chapters, we present details of the monitoring and adaptation mechanisms supported in the CASA framework. As we mentioned earlier, our focus in this work has been mostly on the ways of adapting applications, rather than on monitoring execution environment. Therefore, we present only an abstract design of the monitoring mechanisms for contextual information and resources in Chapter 3. For adapting lower-level services and crosscutting aspects, the CASA framework relies on externally developed mechanisms. We discuss these mechanisms in Chapter 4. For dynamic recomposition of application components, we have developed an indigenous approach which we describe in detail in Chapter 5, and present performance evaluation results for this approach in Chapter 6. Finally, we give an overview of related work in Chapter 7, and present concluding discussion and possible directions for future work in Chapter 8.

Chapter 3

Monitoring the Execution Environment

In this chapter, we give an abstract design of the mechanisms used for monitoring the execution environment of applications (contextual information and resource availability) in the CASA framework. In addition, we present details of the resource allocation algorithm employed by the Resource Manager (RM) in CASA.

Details of resource monitoring in the CASA framework are discussed in Section 3.1. The resource allocation algorithm employed by the RM is described in Section 3.2. Finally, details of monitoring contextual information are presented in Section 3.3.

3.1 Monitoring resources

As discussed in Chapter 2, resource requirements of an application configuration are specified within <`resources`> element in the application contract. Resource requirements can be of two types: hardware requirements (such as CPU, memory, communication bandwidth etc.) and software requirements (i.e. software services required by the corresponding application configuration).

3.1.1 Monitoring hardware resources

Dynamically adaptive applications might be executing on small handheld devices, where the amount of local hardware resources is limited. The limited availability of

hardware resources implies increased contention among software application for using these resources. Similarly, network resources are usually the most contended among applications due to the scarce availability and high demand for these resources.

The RM allocates resources to applications in accordance with the resource requirements specified in the respective application contracts. Resource allocation is done in a fair manner, such that the competing applications are allocated resources according to their relative priorities. Different applications are assigned default priorities by their application developers, and these priorities can be changed by the user according to the user's own needs and preferences.

An application specifies only those resource requirements in its application contract that are necessary for its functional and performance commitments, and that cannot be satisfied by simple sharing of resources as controlled by the underlying operating system.

The RM allocates resources only for the user applications. The total amount of resources available for allocation to the user applications is communicated to the RM, after deducting the amount of resources necessary for the operating system and other critically important system applications.

The first step to allocating resources is the ability to monitor the availability of various resources.

For monitoring various hardware resources required by applications, several resource monitoring systems have been proposed and developed, such as [LMK+03, APK+03, TG04]. These systems operate directly at the platform (operating system, network) level, where resources can be monitored efficiently. The RM relies on an external resource monitoring system for monitoring hardware resources. Many of the existing adaptive middleware systems have built-in resource monitoring services for monitoring hardware resources. If CASA is integrated with any such adaptive middleware system, then the information about the current availability of resources can be provided by the middleware system itself. For the discussion in this dissertation, we assume that CASA is integrated with one such adaptive middleware system called Odyssey [NSN+97]. Details of Odyssey and other resource monitoring systems are discussed next.

Remos [LMK+03] provides a powerful monitoring system for network resources. It provides two types of queries: *flow* queries for getting information about simple network flows, and *topology* queries for getting complex topological information.

Depending on the applications' requirements, the RM may use either of these two interfaces for interacting with Remos for getting information about the current availability of resources. The Remos architecture consists of *Collectors*, *Predictors*, and *Modeler*. Collectors are responsible for acquiring and consolidating the resource information, and forwarding it to the Modeler. Predictors are responsible for predicting future behavior based on the measurement history. Modeler provides interface for resource queries, and is responsible for modeling the Collector-gathered information about the network into the information abstractions required by the client (possibly the RM, in our case). An overview of the Remos architecture, reproduced from [LMK+03], is shown in Figure 3.1.

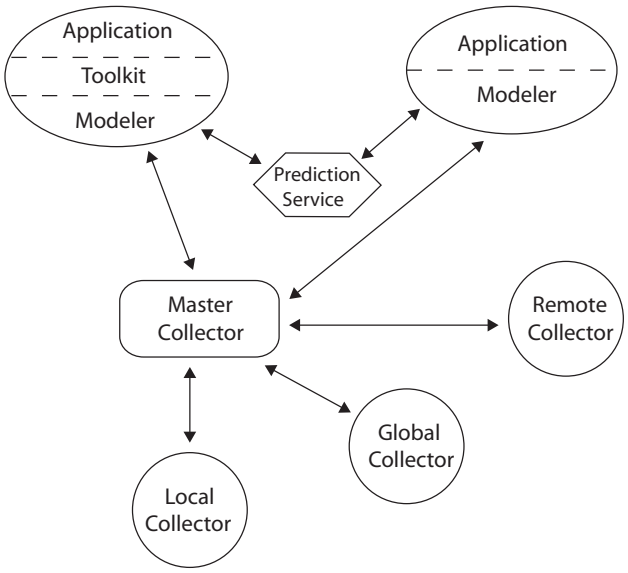


Figure 3.1: Architecture of Remos [LMK+03]

Dproc [APK+03] is able to monitor different hardware resources, such as CPU, memory, disk space, network bandwidth etc. It utilizes the /proc virtual file system, and extends this interface with resource information collected from both local and remote hosts. In order to predictably capture and distribute monitoring information, Dproc uses a kernel-level group communication facility KECho, which implements events and event channels. An overview of the Dproc architecture, reproduced from [APK+03], is shown in Figure 3.2.

Tuduce and Gross [TG04] have proposed an extensible resource monitoring infras-

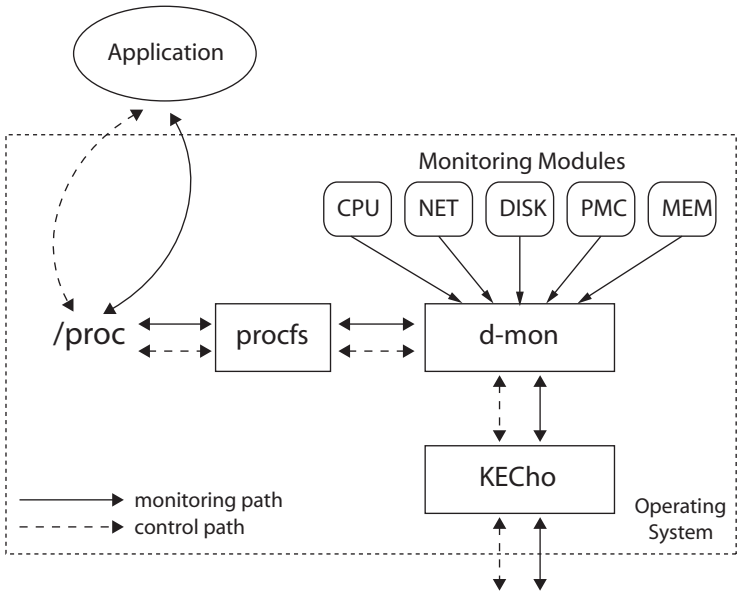


Figure 3.2: Architecture of Dproc [APK+03]

tructure and specific sensors for monitoring different network parameters of interest. The monitoring infrastructure includes abstractions for capturing a node's environment, neighborhood, and the swath (nodes within transmission range along a path). Their approach is mainly targeted towards mobile ad-hoc networks. Here also the interface between a client and monitoring system is query based, so that the client (possibly the RM, in our case) is able to control the level of information needed.

There are some other resource monitoring mechanisms developed for specific resources, such as for CPU [YN01].

All the above resource monitoring systems have been implemented and tested to show that they incur reasonably low overhead for practical applications. Details of their performances can be found in the respective papers.

As mentioned earlier, many of the existing adaptive middleware systems incorporate resource monitoring services for hardware resources. For the discussion in this dissertation, we assume that CASA is integrated with one such adaptive middleware system called Odyssey [NSN+97].

Odyssey is used for two distinct purposes in CASA: for monitoring resources, and for adapting lower-level services. These two purposes are in principle independent

of each other. That is, different systems can be used for monitoring resources and for adapting lower-level services. However, since adaptation of lower-level services is typically carried out in response to a change in the availability of resources, many adaptive middleware systems include the resource monitoring capability.

We have decided to use Odyssey for monitoring resources and for adapting lower-level services for the following reasons. Odyssey is able to monitor a *wide variety* of resources, in particular the most commonly required resources like CPU, memory and network bandwidth. Many of the other resource monitoring systems are specialized to specific types of resources, e.g. the Remos system [LMK+03] is used for network resources, and [YN01] provides a system for monitoring CPU. Moreover, Odyssey is *generally applicable*, and is not targeted to any particular network environment, unlike the approach by Tuduce and Gross [TG04] which is targeted to mobile ad-hoc networks. Odyssey is designed to provide resource information to a client (i.e. the RM in our case) in a *timely* manner, unlike Dproc [APK+03] which is more useful for distributing resource information using group communication. For adapting lower-level services, the strength of Odyssey lies in its *broad applicability* (for different types of lower-level services) and *easy extensibility*.

Below we briefly discuss the resource monitoring capability of Odyssey, while the role of Odyssey in adapting the lower-level services used by an application is discussed in Chapter 4.

Resource monitoring using Odyssey: The architecture of Odyssey is shown in Figure 3.3 (reproduced from [Nob00]).

The Odyssey system consists of *Viceroy* and a set of *Wardens*. Viceroy is responsible for centralized resource management, i.e. Viceroy acts as the single point of resource control for a client. In particular, Viceroy is responsible for monitoring resources, and notifying a client (i.e. the RM, in our case) about any changes in the availability of resources. The RM can request resources on behalf of running applications to Viceroy (using the `request` method). When Viceroy discovers that the availability of a resource has strayed outside the requested window of tolerance for that resource, it notifies the RM accordingly (using the `handler` method), thereby enabling the RM to reallocate resources. Wardens are used for adapting lower-level services used by an application, and are discussed in more detail in Chapter 4.

Odyssey is able to monitor commonly required resources such as network band-

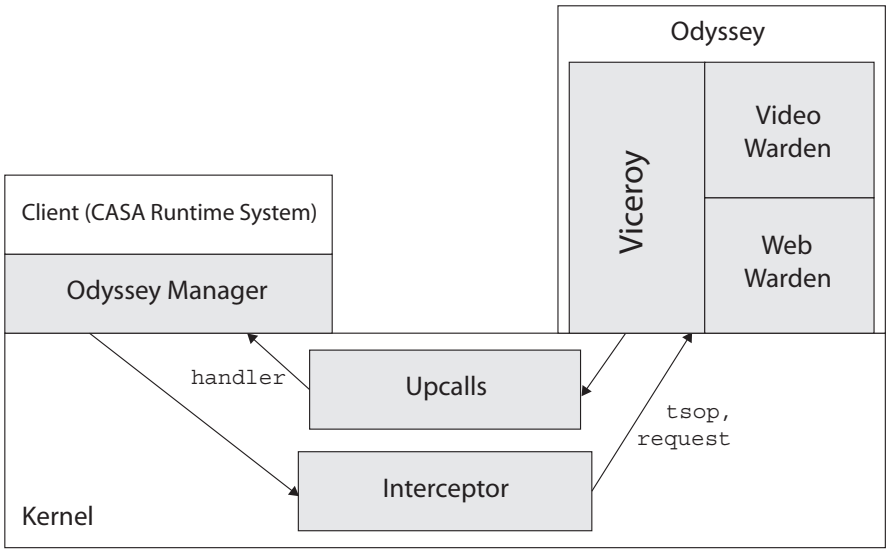


Figure 3.3: Architecture of Odyssey [Nob00]

width, CPU, memory, battery power etc. Since Odyssey is implemented at the operating system level, the resources can be monitored efficiently and any changes in the runtime availability of resources can be detected quickly. Resource monitoring functions of Odyssey are illustrated in Figure 3.4, which is reproduced from [NSN+97].

In the current implementation, Odyssey provides an interface called `request` which accepts a unique resource identifier and tolerance levels for the allocation of the corresponding resource. If the current availability of the requested resource lies outside the bounds of the requested tolerance window, Odyssey returns an error along with the value of current availability of the resource. We propose that Odyssey be extended to provide an additional interface called `availability` that accepts a unique resource identifier and returns the current availability of that resource. The RM requires the values of the current availability of various resources for allocating these resources according to the relative priorities of the running applications. In the absence of such an interface, the RM may simply use the `request` interface and set the tolerance window higher than the maximum possible value of a resource. This way, Odyssey will be forced to return an error and indicate the current availability of the corresponding resource to the RM. Once resource allocation has been done,

```
request(in path, in resource-descriptor, out request-id)
cancel(in request-id)
```

(a) Resource Negotiation Operations

resource-id
lower bound
upper bound
name of upcall handler

(b) Resource Descriptor Fields

Network Bandwidth	bytes/second
Network Latency	microseconds
Disk Cache Space	kilobytes
CPU	SPECint95
Battery Power	minutes
Money	cents

(c) Generic Resources in Odyssey

```
handler(in request-id, in resource-id, in resource-level
```

(d) Upcall Handler

Figure 3.4: Resource monitoring functions of Odyssey [NSN+97]

the RM can use the `request` interface for reserving the desired amount of resources.

Whenever the availability of resources varies outside the requested tolerance bounds, Odyssey automatically informs its client (i.e. the RM) about the new availability using the `handler` method (refer Figure 3.4d). This enables the RM to carry out reallocation of resources, and submit a new request to Odyssey based on the changed allocation.

Interactions between the RM and Odyssey are shown in Figure 3.5. The RM sends a request for resources to the Odyssey Manager, which forwards the request to Viceroy. Similarly, the upcalls made by Viceroy are forwarded by the Odyssey Manager to the RM. The `request` and `handler` method calls exchanged between the Odyssey Manager and Viceroy are forwarded through the Kernel as shown in Figure 3.3.

Please note that in future, Odyssey can be replaced or complemented with any

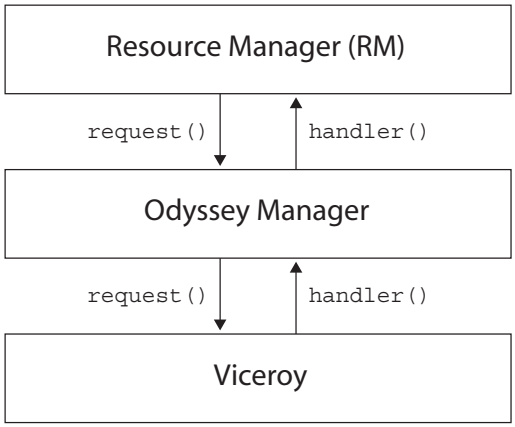


Figure 3.5: Interactions between the RM and Odyssey

other adaptive middleware systems, which may offer better functionality or performance. The only requirement for a middleware system to be integrated with CASA is that the middleware system should be reflective in nature, i.e. allow external regulation of its adaptation strategy, as discussed in Chapter 4.

With regard to resource monitoring, CASA can be integrated with any of the resource monitoring systems that are able to monitor hardware resources efficiently and provide information about the current availability of these resources. CASA can even be integrated with any number of different resource monitoring systems for monitoring different hardware resources. Naturally, it is of no concern to CASA whether these resource monitoring systems are embedded within an adaptive middleware system or deployed independently.

The resource monitoring systems integrated with CASA are assumed to be independent and self-contained. The details of how monitoring is actually performed (e.g. how many samples are taken, how frequently and how long sampling is done etc.) are encapsulated within the design of the resource monitoring systems. The RM does not control the working of the resource monitoring systems, but simply uses the monitoring services provided by these systems as an external client. Different resource monitoring systems for the same resource might differ somewhat in the details of their working, e.g. one system might be able to better hide temporary peaks or drops in the availability of a resource by considering values over a longer period of time than another system. However, most of the resource monitoring systems are

not customizable, i.e. they do not allow any control over their monitoring details, as these details are presumed to be optimized for the design of these systems. The differences in these systems are mostly because of certain trade-offs made in their design, i.e. one system might be more suitable than others in a particular situation but not in some other situations. Therefore, the choice of a resource monitoring system for integration with CASA might depend on the target environments where the monitoring system is to be deployed.

3.1.2 Monitoring software services

For monitoring software services, the RM includes a Software Resource Coordinator (SRC) entity. Software services required by an application configuration might be residing anywhere in the network. The SRC can use a variety of service discovery infrastructures for discovering software services required by a configuration. For the discussion in this dissertation, we assume that software services required by an application configuration are implemented as Web services. The basic Web service technologies include: UDDI (for service discovery), WSDL (for service description) and SOAP (for service communication) [ACKM04]. However, the above basic technologies have not been designed for runtime automated discovery and binding of services. For instance, WSDL is able to describe only the syntactic specification of a service, with no semantic information. UDDI, in its current specifications, supports only WSDL descriptions for advertising and searching services.

Nevertheless, in the last couple of years, there has been a tremendous amount of work done in the area of semantic specification of Web services. The most popular approach for semantic specification of services is OWL-S [OS07]. It is currently in the process of being standardized by World Wide Web Consortium (W3C). OWL-S is an OWL (Web Ontology Language [OWL07]) based ontology for the semantic specification of services.

As discussed in Section 2.3.2, an OWL-S specification of a service has three parts: profile, process model and grounding. A service requestor needs to provide only an OWL-S profile, while a service provider should provide at least the grounding information in addition to the profile description. UDDI has been recently enhanced with OWL-S, as a part of the research effort at CMU [SPS04]. This allows searching services based on their OWL-S descriptions using the UDDI enhanced with OWL-S.

The OWL-S profiles of the required services are provided within the specifica-

tion of the corresponding application configuration in the application contract. The SRC may search for the services that match these OWL-S profiles using the UDDI enhanced with OWL-S. A number of matchmakers for matching the services specified in OWL-S have been developed. One of the efficient matchmakers for OWL-S is called OWL-S API [API07], which is developed as a part of the MINDSWAP project at the University of Maryland.

Once all the services required for a given configuration are discovered, the SRC communicates the successful discovery to the RM. In addition, the SRC passes the references to the service providers of the matched services to the CES. The CES communicates these references to the concerned application, once this configuration is approved for activation.

As an example, the *Tourist-Guide* application might require a restaurant-search service in one of its configurations. The profile of the restaurant-search service is described in OWL-S, and this service can be used for searching a restaurant according to the current time and location of the user, preferred cuisine, and number of persons. The specification of the restaurant-search service within the application contract of the *Tourist-Guide* application is shown in Figure 3.6, and the OWL-S profile of the restaurant search service (as referred by the `reference` attribute in Figure 3.6) is shown in Figure 3.7. The SRC uses the profile to search for a matching service using the UDDI enhanced with OWL-S. Once a matching service is discovered, the SRC communicates the successful discovery to the RM. The reference to the provider of the restaurant-search service is passed to the application by the SRC through the CES. As another example, the *Tourist-Guide* application might require a map service for planning a route to the next point-of-interest.

In future, the SRC can be extended to discover services other than those accessible as Web services, e.g. UPnP [UPn07] and OSGi [OSG07] services.

```
<sw name="restaurant-search"
    reference="/sw-folder/restaurant-search.owl"/>
```

Figure 3.6: Example <sw> element in the application contract

```
<rest:RestaurantSearchService rdf:ID="RestaurantSearchProfile">
    <profile:hasInput rdf:resource="#date"/>
    <profile:hasInput rdf:resource="#time"/>
    <profile:hasInput rdf:resource="#location"/>
    <profile:hasInput rdf:resource="#cuisine"/>
    <profile:hasInput rdf:resource="#numberOfPersons"/>
    <profile:hasOutput rdf:resource="#restaurantList"/>
</rest:RestaurantSearchService>
```

Figure 3.7: Example OWL-S profile for restaurant search service

3.2 Resource allocation algorithm

The Resource Manager (RM) is responsible for allocating resources to various running applications, and reallocating resources whenever there is a change in the availability or demand for resources.

Resource allocation decisions are governed by the relative priorities of various running applications. Relative priorities of applications are based on the user-perceived preferences for these applications. Every application has a certain default priority (expressed as a number between 1 and 10) defined by the application developer. The default priority of an application can however be changed by the user according to the user's own needs and preferences. The RM maintains a priority list PL of all running applications. This list is sorted from the lowest priority to the highest priority applications.

Below we present simplified versions of the algorithms used for resource allocation by the RM, hiding away some of the details and optimizations in order to make the algorithms easy to follow. Detailed algorithms are provided in Appendix B.

Please note that in the following algorithms, wherever a change in configuration or suspension is mentioned, it is carried out only within the data structures of the algorithm, until the execution of the algorithm is over. Only after the successful completion of the algorithm and approval by the CES, all the decided changes in configurations and suspensions are actually carried out on the concerned applications. This is analogous to database transactions where all modifications in a transaction are made tentatively until the transaction commits, and only then the modifications become persistent. Or as in a virtual shopping cart of an online shopping system,

where all items added to or removed from the shopping cart are tentative until the customer checks out and pays for the items, and only then the order is actually processed.

3.2.1 Allocating resources for initial activation

The first algorithm we present is `New_Request` (refer Figure 3.8), which is invoked when resources for a configuration from a given context element are to be allocated, and accordingly the configuration is selected for activation. The input parameters for this algorithm are an application identifier A (for referring the corresponding application contract) and a context identifier C (as specified within the application contract). The algorithm either returns the identifier of the selected configuration, or `null` if no configuration could be selected in the current availability of resources.

The `New_Request` algorithm depends on `Allocate_Resources` (refer Figure 3.9) and `Must_Allocate_Resources` algorithms (refer Figure 3.10).

`Allocate_Resources` accepts an application identifier and a configuration identifier, and returns true if resources for the given configuration are allocated successfully or false otherwise.

If enough free resources are available, `Allocate_Resources` simply allocates resources for the given configuration. Otherwise (i.e. if enough free resources are not available), `Allocate_Resources` reconfigures lower priority applications (i.e. applications with lower priority than the requesting application) in order to free up resources for accommodating the given configuration, but without reconfiguring any of the higher priority applications and without suspending any of the lower priority applications.

If the `Allocate_Resources` algorithm fails to allocate resources for any of the configurations in the given context element, then `New_Request` invokes the `Must_Allocate_Resources` algorithm. The `Must_Allocate_Resources` algorithm tries to allocate resources for the configuration passed to it, by reconfiguring all running applications (including the higher priority ones) and even suspending lower priority applications to free up resources. `Must_Allocate_Resources` also accepts an application identifier and a configuration identifier, but it returns the numbers and net priorities of all applications that need to be reconfigured or suspended for accommodating the given configuration. This data is used by the `New_Request` algorithm for selecting an appropriate configuration that results in least number of

suspensions and reconfigurations of other applications. In case of the same number of suspensions or reconfigurations, net priority of the applications to be suspended or reconfigured is taken as the selection criteria.

The criteria behind allocating resources to a new application (as manifested in the above algorithms) are: (1) first, try to allocate resources without disturbing any of the running applications, (2) if this is not possible, then reconfigure the applications with lower priority than the new application one-by-one (starting with the lowest priority) to free the resources required by the new application, (3) if this does not work either, then reconfigure the same priority and higher priority applications to free the resources required by the new application, (4) if all the above fail to work, start suspending the lower priority applications one-by-one to free the resources for the new application. If in spite of all the above, sufficient amount of resources for the new application are not available, the failure is communicated to the application.

If accommodating a new application results in suspending other lower priority applications (step 4 above) and/or reconfiguring the same priority and higher priority applications (step 3 above), then the configuration that results in the least number of suspensions and reconfigurations of other applications is selected for activation.

The decision to reconfigure higher priority applications in order to accommodate a new (low priority) application (step 3 above) is based on the assumption that the execution of the new application is important for the user, even though not as important as the execution of the higher priority applications. That is, in order to accommodate the new application, the user is ready to sacrifice the functionality or performance of the higher priority applications (without suspending these applications), just like she would have done in case of loss of certain resources required by these higher priority applications. More importantly, the user is ready to sacrifice the functionality or performance of the higher priority applications, rather than suspending some of the lower priority applications (which is done in step 4 above).

However, when a higher priority application is to be reconfigured for accommodating a new application (or even when a lower priority application is to be suspended for accommodating the new application), the user is prompted for confirmation (a feature that the user may turn off, if required), at which point the user may accept or deny the reconfiguration (or suspension). If a reconfiguration or suspension is denied by the user, the resource allocation algorithm considers other applications for reconfiguration and/or suspension. The user may also disable some of the low-preference

```
New_Request(App A, Context C)
BEGIN
     FOR(each Config G in C) /* starting from top to bottom */
     DO
          Allocate_Resources(A, G);
          IF(resources allocated successfully for G) THEN
               RETURN G;
          ENDIF
     ENDFOR
     FOR(each Config G in C) /* starting from top to bottom */
     DO
          Must_Allocate_Resources(A, G);
          IF(resources allocated successfully for G) THEN
               STORE G in a list Q along with information about
               applications to be suspended and reconfigured for
               accommodating G;
          ENDIF
     ENDFOR
     IF (the list Q is empty) THEN
          RETURN NULL;
     ENDIF
     Selected_G = first Config in the list Q;
     FOR(each Config G in the list Q, starting from second Config)
     DO
          IF(number of applications to be suspended for G is less
             than that for Selected_G) THEN
               Selected_G = G;
               CONTINUE;
          ENDIF
          IF(number of applications to be suspended for G is the
             same as that for Selected_G) THEN
               IF(net priority of applications to be suspended for
                  G is less than that for Selected_G) THEN
                    Selected_G = G;
                    CONTINUE;
               ENDIF
          ENDIF
          IF(number and net priority of applications to be suspended
             for G are the same as that for Selected_G) THEN
               IF(number of applications to be reconfigured for G
                  is less than that for Selected_G) THEN
                    Selected_G = G;
                    CONTINUE;
               ENDIF
               IF((number of applications to be reconfigured for G
                  is the same as that for Selected_G) and (net priority
                  of applications to be reconfigured for G is less than
                  that for Selected_G) THEN
                    Selected_G = G;
                    CONTINUE;
               ENDIF
          ENDIF
     ENDFOR
     RETURN Selected_G;
END New_Request
```

Figure 3.8: Algorithm for new request

```
Allocate_Resources(App A, Config G)
BEGIN
     Allocate free resources required by G;
     IF(resource requirement of G is fulfilled) THEN
          RETURN TRUE;
     ENDIF
     FOR (each application K in PL with Priority(K) < Priority(A))
     /* starting from lowest to highest priority */
     DO
          Replace current Config of K with the one (within the
          same Context) that results in freeing up maximum amount
          of resources required by G;
          Allocate freed-up resources to G;
          IF(resource requirement of G is fulfilled) THEN
               RETURN TRUE;
          ENDIF
     ENDFOR
     RETURN FALSE;
END Allocate_Resources
```

Figure 3.9: Algorithm for allocating resources

```
Must_Allocate_Resources(App A, Config G)
BEGIN
     Allocate free resources required by G;
     FOR (each application K in PL)
     /* starting from lowest to highest priority */
     DO
          Replace current Config of K with the one (within the
          same Context) that results in freeing up maximum amount
          of resources required by G;
          Allocate freed-up resources to G;
          IF(resource requirement of G is fulfilled) THEN
               RETURN (number and net priority of all applications
               that need to be reconfigured for accommodating G);
          ENDIF
     ENDFOR
     FOR (each application K in PL with Priority(K) < Priority(A))
     /* starting from lowest to highest priority */
     DO
          IF(K has allocated resources required by G) THEN
               Suspend K to free up its resources;
               Allocate freed-up resources to G;
               IF(resource requirement of G is fulfilled) THEN
                    RETURN (number and net priority of all
                    applications that need to be reconfigured
                    and suspended for accommodating G);
               ENDIF
          ENDIF
     ENDFOR
     RETURN NULL;
END Must_Allocate_Resources
```

Figure 3.10: Algorithm for compulsorily allocating resources

configurations of the higher priority applications within the respective application contracts, so that these configurations are not considered by the resource allocation algorithm. Obviously, the user may, at any time, voluntarily stop the execution of an application (for instance, some of the lowest priority applications) to free up resources for better configurations of some other applications.

A more sophisticated way of managing the priorities of applications – for instance, different priority levels for different adaptation actions (suspensions and reconfigurations) and for different reasons for adaptation (loss of resources, increase in resources, accommodating new application requests) – is a part of the future work.

In the `Allocate_Resources` and `Must_Allocate_Resources` algorithms, we have mentioned replacing the current configuration (say, GK) of an application K with the one (say, GK') that results in freeing up maximum amount of resources required by the requested configuration G. The choice of GK' depends not merely on the absolute quantity of all the resources that are freed, but also on the *Resource Contention Factor (RCF)* of each resource.

In particular, the RM dynamically computes RCF for every resource, based on the recent history of contention for the resources. RCF for a resource R is computed as the total demand for the resource R divided by the total availability of the resource R at any time. That is,

$$RCF_R = \frac{(Total\ \ demand)_R}{(Total\ \ availability)_R}$$

RCF for a resource is computed every time there is a change in the demand or availability of the resource. The final RCF value of a resource (for the purpose of comparing with RCFs of other resources) is calculated as an average of the latest five RCF values computed. This is done to mitigate the effects of any temporary peaks or downfalls in the contention for a resource.

For every configuration that is a potential candidate for replacing GK, the freed-up amount of each resource (i.e. the amount of resource that will be freed-up by switching to this configuration) is divided by the total availability of the resource (for normalization, i.e. to compare all resources at an equal level). This value is then multiplied with the RCF of that resource, to arrive at the *saving value* for that particular resource. That is,

$$Saving\ \ value_R = RCF_R \cdot \frac{(Freed\text{-}up\ \ amount)_R}{(Total\ \ availability)_R}$$

Finally, saving values for all resources required by G are summed up to arrive at the saving value for that configuration. The saving value for a particular resource can also be negative, if the freed-up amount for that resource is actually negative (i.e. the requirement of the new configuration for that particular resource is actually more than that of the current configuration). However, a configuration change is allowed only if sufficient resources are available for the new configuration.

Among the freely available resources, the resources required by G are already allocated to it before a change in the configurations of other applications is considered. Please note that the saving values are computed only for those resources which are still required by G, i.e. which are not already allocated to G in the required amounts.

The configuration with the highest saving value is selected for replacing GK. The idea here is that resources that are highly contended (with high RCF) are likely to be available in scarce quantities, and should be given a priority for freeing them up.

In case of suspending applications in the `Must_Allocate_Resources` algorithm, if suspension of an application results in freeing up some extra resources than those required by G, then the RM tries to revoke the suspensions (or reconfigurations) of other applications that were decided to be suspended (or reconfigured) earlier but could use these extra resources to continue their execution – either with their original configurations or with alternative configurations. Similarly, reconfiguration of an application (in the `Allocate_Resources` and `Must_Allocate_Resources` algorithms) might result in revoking the reconfigurations of some other applications that were decided to be reconfigured earlier.

If there is a runtime change in an application's contextual information, the CM detects this change and communicates the new <`context`> element to the CES. The CES communicates the new <`context`> element to the RM. The RM frees up the resources allocated to the currently active configuration of the application in its internal data structures, and invokes the `New_Request` algorithm with the new context identifier.

If the user changes anything in the application contract, the whole contract is re-evaluated by the CES. If there is any change in the ordering of <`context`> elements such that a lower <`context`> element has now become higher than the currently active <`context`> element, then the CM is invoked again. If the CM concludes that there is a change in the current <`context`> element due to the reordering, then the

CES communicates the new <context> element to the RM as above.

If the current <context> element remains valid (either the <context> elements were not reordered as above, or the CM concludes that current <context> is valid even after reordering), but the ordering of <config> elements has been changed within the current <context> element such that a lower <config> has now become higher than the current <config>, then also the New_Request algorithm is invoked with the current context identifier.

Similarly, any changes in the definition of contextual parameters of the current or any higher <context> elements, or a change in the resource requirements of the current or any higher <config> elements, is treated as a new request from the application.

3.2.2 Reallocating resources in response to a change in availability

The RM needs to reallocate resources whenever there is a change in the availability of resources, either in the form of reduced availability or increased availability.

In response to a change in the availability of resources, the RM uses the following two algorithms for reallocating resources: Less_Resources (when informed about the reduced availability of resources, refer Figure 3.11) and More_Resources (when informed about the increased availability of resources, refer Figure 3.12).

The reduced availability of resources might result in reconfiguring or even suspending some of the running applications. The running applications are reconfigured or suspended according to their relative priorities. The decision for reconfiguring an application takes into account the RCF for the freed-up resources, similar to the one for accommodating a new application, as described in the previous sub-section.

For the increased availability of resources, a change to a more preferred configuration of each running application is attempted, in the order of the relative priorities of the applications (i.e. starting from the high priority applications to the low priority ones). The concerned application may approve or reject a change to a more preferred configuration (e.g. the application may decide not to disrupt its current execution even for a more preferred configuration).

If suspension of an application in the Less_Resources algorithm results in freeing up some extra resources than those required to reduce the resource deficit, then the RM tries to revoke the suspensions (or reconfigurations) of other applications that were decided to be suspended (or reconfigured) earlier but could use these extra

```
Less_Resources(current resource availability)
BEGIN
     resource_deficit = (current resource allocation - current resource availability);
     FOR (each application K in PL)
     /* starting from lowest to highest priority */
     DO
          Replace current Config of K with the one (within the same Context)
          that results in freeing up maximum amount of resources required to
          reduce resource_deficit;
          Compute new resource_deficit;
          IF(resource_deficit <= 0) THEN
               RETURN;
          ENDIF
     ENDFOR
     FOR (each application K in PL)
     /* starting from lowest to highest priority */
     DO
          IF(K has resources required to reduce resource_deficit) THEN
               Suspend K to free up its resources;
               Compute new resource_deficit;
               IF(resource_deficit <= 0) THEN
                    RETURN;
               ENDIF
          ENDIF
     ENDFOR
END Less_Resources
```

Figure 3.11: Algorithm for reduced availability of resources

```
More_Resources(current resource availability)
BEGIN
     resource_surplus = (current resource availability - current resource allocation);
     FOR (each application K in PL in the reverse order of priority)
     /* i.e. starting from highest to lowest priority */
     DO
          IF(current Config of K is not the highest in its current Context) THEN
               Try to replace current Config of K with a higher Config in its
               current Context, starting from the highest to current Config;
               Compute new resource_surplus;
               IF(resource_surplus == 0) THEN
                    RETURN;
               ENDIF
          ENDIF
     ENDFOR
END More_Resources
```

Figure 3.12: Algorithm for increased availability of resources

resources to continue their execution. Similarly, the reconfiguration of an application might result in revoking the reconfigurations of some other applications that were decided to be reconfigured earlier, if some extra resources than those required become available.

When an application terminates, its resources are freed up and the `More_Resources` algorithm is invoked.

3.2.3 Complexity of the resource allocation algorithms

Here we discuss the complexity of the algorithms described above for the initial allocation of resources to an application, and for reallocating resources to applications in response to a change in the availability of resources. For allocating resources to a new application for its initial activation, the RM uses three algorithms: `New_Request`, `Allocate_Resources` and `Must_Allocate_Resources`. For simplicity, let us define the average number of alternative configurations in a given <`context`> element of an application as n. Let the average number of running applications (that are managed by the CASA Runtime system) at any given time be p (this number is the same as the length of the priority list PL maintained by the RM, which contains all running applications).

The complexity of both `Allocate_Resources` and `Must_Allocate_Resources` algorithms is $O(pn)$. The relatively low complexity of these two algorithms is because of our approach of using the RCF of resources for selecting an appropriate configuration for an application to be reconfigured (as discussed in the previous sub-sections). Otherwise, a more exhaustive search for selecting the best possible configuration for every application to be reconfigured, i.e. choosing among n^p different combinations in the worst case, would result in a much higher complexity. The approach based on RCF of resources works satisfactorily in most cases, even though it may not give the best possible results in all cases as could be possible with the exhaustive search. However, given the huge difference in the complexities of the two approaches, the approach based on the RCF of resources has been preferred. The `New_Request` algorithm depends on `Allocate_Resources` and `Must_Allocate_Resources` algorithms. The complexity of the `New_Request` algorithm is $O(pn^2)$.

The complexity of the `Less_Resources` algorithm (for reduced availability of resources) is $O(pn)$, and that for the `More_Resources` algorithm (for increased availability of resources) is also $O(pn)$. Here also, the use of our approach based on RCF

of resources for deciding an appropriate configuration for a reconfiguring application results in the relatively low complexity of the `Less_Resources` algorithm.

Since the average number of alternative configurations in a given <`context`> element of an application (i.e. n) is likely to be low, and the average number of running applications at any given time (i.e. p) is also likely to be low, the complexity of the above resource allocation algorithms is reasonable. We believe that for most common scenarios in practice, a typical value of n is likely to be less than 10, and that for p is likely to be less than 20.

A discussion on the runtime performance of a CASA prototype system is given in Chapter 6.

3.3 Monitoring contextual information

As briefly outlined in Chapter 2, monitoring contextual information consists of the following steps:

- acquiring the data related to contextual information,
- structuring the acquired data based on an application domain-specific ontology, and
- deducting the final knowledge, i.e. the contextual information relevant to the application, from this data.

An overview of the architecture of the Context Monitor (CM) is given in Figure 3.13.

At the lowest level, a variety of Context Sensors are used to acquire the data related to contextual information. At the middle level, the acquired data is structured based on an application domain-specific ontology by a collection of Context Interpreters (each one responsible for a different context parameter), and the values of the corresponding context parameters are deducted from this data. And, at the top level, a Context Matcher combines these values of context parameters, and matches these values with the context parameter values defined in the application contract, in order to determine the currently valid <`context`> element (i.e. the final contextual information relevant to the application). The Context Matcher then communicates the determined <`context`> element to the CES (Contract Enforcement System).

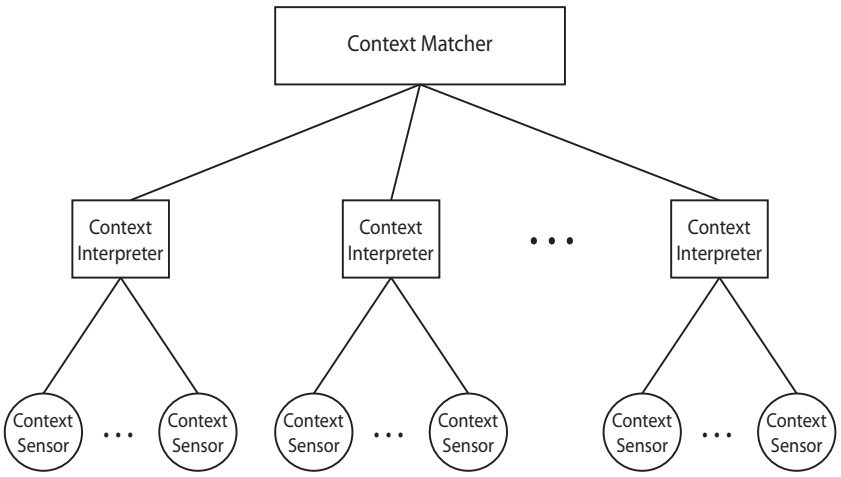


Figure 3.13: Context Monitor

Context sensors can be used for sensing a wide variety of contextual data. This includes providing measurements of physical variables such as temperature, lighting, physical motion etc., or a combination of sensors for identifying objects or persons. Simply reading the GPS data for ascertaining the current location, or even referring to the user's agenda for detecting the current user activity is also a part of context sensing. It is obvious that different application domains might require entirely different sets of context sensors. The context sensors can range from highly sophisticated hardware sensors to relatively simple software sensors.

The data sensed by different context sensors is passed to the corresponding Context Interpreters. Context Interpreters structure the acquired data according to an application domain-specific ontology for deducting the values of the relevant context parameters.

The role of an ontology is very important in the design of the CM, as it actually provides the semantics for the observed data in order to extract useful information about the contextual parameters of interest from this data. Design and development of an ontology is a separate and broad field of research, and we do not address this in detail here. However, the design of an ontology should allow sufficient expressiveness, and extensibility to accommodate new context parameters. OWL (Ontology Web Language) [OWL07] is an emerging standard for defining an ontology. OWL is based on RDF, which in turn is based on XML. Being XML-based, an ontology

defined in OWL can be easily extended or modified. OWL provides rich expression capabilities for defining an ontology, and is very well suited for ontological knowledge representation of contextual information.

Figure 3.14 shows an example OWL ontology for discovering whether the user is currently in a meeting. This requires satisfaction of two conditions. First the user must be in a meeting room, and second the number of persons in the room must be at least two. The current location of the user and the number of persons in the vicinity of the user are detected by different context sensors. The data collected by these sensors is passed to the corresponding Context Interpreter. The Context Interpreter structures this data according to the OWL ontology, and decides whether the user is currently in meeting. This information is then passed by the Context Interpreter to the Context Matcher. The Context Matcher uses the information about different context parameters sent by different Context Interpreters in order to decide the currently valid <context> element.

```
<owl:Class rdf:ID="inMeeting">
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#currentLocation"/>
                <owl:hasValue rdf:resource="#meetingRoom"/>
            </owl:Restriction>
        </owl:Class>
        <owl:Class>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#numberOfPersons"/>
                <owl:minCardinality rdf:datatype="xsd:nonNegativeInteger">2
                </owl:minCardinality>
            </owl:Restriction>
        </owl:Class>
    </owl:intersectionOf>
</owl:Class>
```

Figure 3.14: Example OWL ontology

The architecture of the CM is very modular, and therefore allows easy extensibility. The extensibility of the CM is very important as different applications can be interested in different parameters of contextual information, some of which may be highly specific to an application and thus could not be provided by a standard CM. The architecture of the CM allows new Context Sensors and new Context Interpreters to be added to the CM. The job of the Context Matcher is simply to collect the values of different context parameters from all available Context Interpreters, and match these values for deciding the currently valid <`context`> element in the application contract. Since the Context Matcher provides only a general functionality, it needs not be evolved.

The Context Matcher provides an interface whereby new Context Interpreters can get registered with it. Similarly, Context Interpreters provide interfaces for registering new Context Sensors.

Context-awareness is a relatively new field of research and development, and therefore there are not many general-purpose context monitoring mechanisms available. Most of the techniques for context monitoring presented in the literature have been tightly bound to the target applications for which these techniques have been primarily developed. Examples of such techniques are ParcTab [WSA+95], Active Badge [WHFG92], Stick-e Document [Bro96] etc.

The Context Toolkit [SDA99] developed at Georgia Institute of Technology provides by far the most general-purpose toolkit for developing context monitoring mechanisms. The Context Toolkit relies on the concept of context widgets, which mediate between a client and its contextual environment. Context widgets are used for sensing contextual data, and providing this data to clients. The Context Toolkit includes a widget library for sensing common contextual data such as presence, identity and activity of people and things. The context widgets act as building blocks for developing context monitoring mechanisms.

In the design of the CM, the Context Toolkit can be used for developing context widgets, which would actually act as context sensors and provide contextual data to the Context Interpreters. The Context Interpreters will then combine the data received from the relevant context widgets, and decide the value of the corresponding context parameter with the help of the context ontology. The advantage of using context widgets is that they already abstract the data collected from various sources in a format which can be readily understood by their clients, i.e. Context Interpreters

in our case.

We leave further details on monitoring contextual information and resources as a future work. In the remaining chapters of this dissertation, we focus on mechanisms for dynamic adaptation of applications.

Chapter 4

Dynamic Adaptation of Lower-level Services and Aspects

As discussed in Chapter 2, lower-level services in this dissertation mean underlying services required by an application for its execution, e.g. data compression, transmission, encryption, caching, video coding/decoding etc. These services can be shared among different applications, and are provided by the deployment platform. Lower-level services that are part of an application configuration are listed in the application contract in the <`llservices`> element within the corresponding <`config`> element. These services are considered to be always available, i.e. an application can always rely upon the availability of these services. If the availability of a service cannot be relied upon, then this service should be listed as a software resource in the <`resources`> element in the application contract, and not in the <`llservices`> element.

For the discussion here, we assume that all the required lower-level services are managed by an underlying adaptive middleware system. However, if a certain lower-level service is not managed by the middleware system, then a manager for the service should be implemented and its reference should be provided to the CES, so that the CES may activate or deactivate the lower-level service, or change some parameters of the service as required for the selected application configuration.

The term aspect is used here in the sense of aspect-oriented programming (AOP)

[KLM+97]. That is, aspects implement the crosscutting concerns of an application, in contrast to components that implement the core functional concerns of the application. Examples of crosscutting concerns are access control, persistence management, transaction management etc. The aspects that can be added or removed dynamically are listed in the <aspects> element within the corresponding <config> element in the application contract. Whereas, any other aspects that need to be always present in an application irrespective of its current configuration need not be specified in the application contract. These aspects are activated during the application initialization.

Details of dynamic adaptation of lower-level services are discussed in Section 4.1, and details of dynamic adaptation of aspects are discussed in Section 4.2.

4.1 Dynamic adaptation of lower-level services

A number of adaptive middleware systems have been developed for adapting lower-level services used by an application. Many of these systems are quite mature, offering good functionality as well as performance. Therefore, in CASA, we have decided to use an external adaptive middleware system for adapting lower-level services.

In order to be integrated with the CASA framework, the adaptive middleware system should allow external regulation of its adaptation strategy, so that the CES may instruct to activate or deactivate the corresponding lower-level services, and control the quality of these services according to the requirements of the selected application configuration. In other words, an adaptive middleware should be reflective in nature in order to be integrated with the CASA framework.

There are several reflection-based adaptive middleware systems available, which can potentially be integrated with the CASA framework. These different adaptive middleware systems have their respective strengths, and depending on a specific application's domain and adaptation requirements, one of these systems may outperform others. We envision that in practice different flavors of CASA will be available, with each one suited for a different application domain and adaptation requirements, and therefore integrated with a different adaptive middleware system. In fact, the CASA framework can be integrated with more than one adaptive middleware system at the same time. As discussed later, the choice of an adaptive middleware system does not affect the design of the CASA Runtime System (CRS).

For the discussion here, we consider an adaptive middleware system called Odyssey [NSN+97, Nob00], which is a strong contender for integration with the CASA framework, and is one of the most general-purpose adaptive middleware systems. Odyssey is very *broadly applicable*, as it can be used for adapting a broad range of lower-level services. Moreover, the design of Odyssey allows *easy extensibility*, i.e. it allows integrating new adaptation capabilities for lower-level services. Some other adaptive middleware systems that may potentially be integrated with CASA are discussed in Chapter 7 on Related Work.

Adapting lower-level services using Odyssey: As discussed in Section 3.1.1, the Odyssey system consists of *Viceroy* and a set of *Wardens*. Viceroy is responsible for centralized resource management, and Wardens are responsible for adapting lower-level services used by an application. In addition to being the single point of resource control for clients, Viceroy also acts as the single point for the clients' interactions with different Wardens. All communication between a client and Wardens is brokered by Viceroy.

Odyssey was originally designed for adaptations related to remote data access, and Wardens were used solely for managing lower-level services dealing with data access by client applications, e.g. video coding/decoding service, Web caching service etc. A Warden for managing a lower-level data access service supports multiple *fidelity* levels for the corresponding service. A client application can select any of these fidelity levels dynamically depending on the current execution environment conditions, by instructing the corresponding Warden accordingly.

However, Wardens can in fact be implemented for managing all types of lower-level services, and not just those dealing with data access (though most of the lower-level services anyway deal with data access only). For the purpose of CASA, for every lower-level service required by applications, a corresponding Warden is implemented in Odyssey. In Odyssey terms, a Warden should support different fidelity levels (i.e. quality levels) for the corresponding lower-level service.

The CES acts as a client of Odyssey for managing lower-level services. That is, the CES can instruct the relevant Wardens to activate the corresponding lower-level services at the desired quality levels, as dictated by a selected application configuration.

In order to avoid explicitly encoding each fidelity changing operation carried out

by Wardens, Odyssey provides a general mechanism called *type-specific operation* or *tsop*. The parameters of a `tsop()` call include the target object, the operation to be performed, any arguments to be passed and results to be returned by the operation. Figure 4.1 shows the interface of a `tsop()` call. More details on the functionality of Odyssey can be found in [Nob98].

```
tsop(in path, in opcode, in insize, in inbuf, inout outsize, out outbuf)
```

Figure 4.1: Type-specific operation in Odyssey

The `path` passed as an argument to a `tsop()` call specifies the path of the corresponding Odyssey object on which an operation is to be performed. The `opcode` argument specifies the operation to be performed. The `insize` and `inbuf` arguments specify the size and buffer for the input parameters passed to the operation. Similarly, `outsize` and `outbuf` specify the size and buffer for the return parameters. As can be observed, the format of a `tsop()` call is very generic, and can be used for any type of Warden managing any type of lower-level service. The `tsop()` methods are traditionally called type-specific operations because these methods are typically used for adapting the access to a particular type of data. For example, a `tsop()` method can be used for changing the quality of video data, or for changing the caching policy for Web data.

Wardens implement two types of operations: one for consuming the lower-level services, and other for adapting the lower-level services. That is, a client might use `tsop()` calls not just for adapting the lower-level services, but also for invoking these services to get the desired results. In case of CASA, the operations for consuming lower-level services are invoked directly by the application, while operations for adapting the services are invoked by the CES.

As an example, Odyssey provides a Warden for managing data access to QuickTime [Qui07] movies. The Warden supports an operation for fetching movie frames, and another operation for changing the quality of the movie being played [Nob98]. The operation for fetching movie frames is called `QT_GetFrame`, which takes a movie reference and time as arguments, and returns the frame of the specified movie after the specified time. The Warden supports three different quality levels for QuickTime movies: full color uncompressed, full color with lossy JPEG compression, and black and white. These three alternative quality levels have obvious differences with

respect to their network bandwidth requirements. Different qualities of video are referred as different *tracks*. The operation for dynamically adapting the quality level of a movie is called `QT_SwitchTracks`, which takes a track reference as an argument, and adapts the quality level of the movie to the specified track accordingly.

The <`llservices`> element in the application contract contains a number of <`lls`> elements. Every <`lls`> element specifies an operation to be performed for adapting a particular lower-level service. An <`lls`> element has a `manager` attribute specifying the corresponding manger of the lower-level service, `name` attribute specifying the name of the lower-level service, `operation` attribute specifying the name of operation to be performed, and a number of <`arg`> sub-elements specifying the values and types of the arguments to be passed to the operation. The role of a manager here is to make the CASA Runtime System (CRS) decoupled from the particular adaptive middleware system being used, thereby allowing any adaptive middleware system to be easily integrated with CASA.

An adaptive middleware system integrated with the CASA framework needs to provide a manager entity for managing all the lower-level services and resources that are controlled by the middleware system. The manager provides an interface to the CRS for managing lower-level services as well as resources. In effect, the manager acts as a Façade (in the sense of the Façade design pattern [GHJV95]) for the underlying adaptive middleware system. A client of an adaptive middleware system interacts only with the manager of the system. The manager in turn forwards the client's requests to the appropriate components of the system.

When Odyssey is integrated with CASA, the <`lls`> elements are used for specifying corresponding `tsop()` calls for controlling the quality of the lower-level services. In detail, the `manager` attribute of an <`lls`> element contains a reference to the Odyssey Manager (which is an interface to all Odyssey-managed lower-level services), the `name` attribute specifies the name of the lower-level service (this name is used for internal reference only, as this name is not passed to the Odyssey Manager), the `operation` attribute has a value `tsop` (as this is the only operation to be performed whenever a lower-level service needs to be adapted), and a number of <`arg`> sub-elements specifying the values and types of the arguments to be passed to the corresponding `tsop()` call. We assume that the quality changing operation performed by a `tsop()` call does not have any return parameters.

Continuing with the above example of adapting the quality of QuickTime movies

in response to a change in the network bandwidth available, the corresponding `tsop()` call will have parameters specifying the object on which the quality changing operation needs to performed (this object provides a reference to the corresponding Warden), the name of the quality changing operation (i.e. `QT_SwitchTracks`, in this example), and the arguments to be passed to this operation. An example <`lls`> element is shown in Figure 4.2, and the corresponding `tsop()` call is shown in Figure 4.3. In this example, the arguments passed to the `QT_SwitchTracks` operation indicate the movie identifier and the new track identifier for the movie respectively. There are no return parameters to be passed, as this quality changing operation does not have any return value.

```
<lls manager="Odyssey" name="video" operation="tsop">
    <arg value="/VideoWarden/QT_Movie" type="string"/>
    <arg value="QT_SwitchTracks" type="string"/>
    <arg value="5" type="int"/>
    <arg value="2" type="int"/>
</lls>
```

Figure 4.2: Example <`lls`> element

```
tsop(/VideoWarden/QT_Movie, QT_SwitchTracks, 5, 2)
```

Figure 4.3: Example `tsop()` call

Once an application configuration is selected for activation, the CES issues the `tsop()` calls specified in the corresponding <`lls`> elements to the Odyssey Manager. The Odyssey Manager modifies a `tsop()` call into a format expected by Viceroy (i.e. passing arguments in a buffer), and forwards the call to Viceroy. The Viceroy determines the appropriate Warden for this `tsop()` call by referring to the first argument of the call. Thereafter, the Viceroy invokes the appropriate Warden, which carries out the specified operation for adapting the corresponding lower-level service. The sequence of these calls is shown in Figure 4.4.

Odyssey offers very good runtime performance. In particular, both resource monitoring as well as dynamic adaptation of lower-level services are carried out with minimal time delays and overhead. More information on the runtime performance

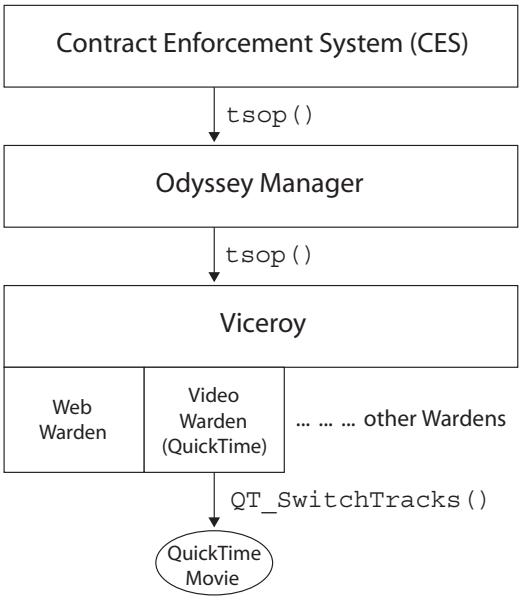


Figure 4.4: Interactions between the CES and Odyssey

of Odyssey is given in [NSN+97].

As discussed earlier, Odyssey is only one of several potential adaptive middleware systems that may be integrated with the CASA framework. Some adaptive middleware systems may be more suited to certain application domains than others. Therefore, the final choice of an adaptive middleware system depends upon the application domain, and the given applications' requirements with respect to the adaptation of lower-level services.

Since the coupling between the CASA Runtime System (CRS) and the adaptive middleware system is very loose, the choice of an adaptive middleware system does not affect the overall design of the CRS. As discussed earlier, the <llservices> element in the application contract contains generic <lls> elements, specifying the calls to be made for adapting the lower-level services as required by the corresponding application configuration. These specified calls are made by the CES to the corresponding manager of the adaptive middleware system integrated with CASA. The manager receives the calls made by the CES, and translates these calls into a format accepted by the corresponding adaptive middleware system, before forwarding the calls to the middleware system. Similarly, more than one adaptive middleware

system can be simultaneously integrated with CASA, with each middleware system providing a manager for interacting with the CRS.

4.2 Dynamic adaptation of aspects

As discussed earlier, aspects implement crosscutting (orthogonal) concerns of a software application. The functionality implemented by an aspect is crosscutting in nature, because different components across an application rely on this functionality. This implies that the crosscutting functionality provided by an aspect usually needs to be executed at several points within an application. The description of an aspect consists of two parts: *join-points* (defining where the aspect is to be executed) and *advice* (defining what is the code that needs to be executed). That is, the join-points define the execution points within an application where the aspect is to be executed, and the advice defines the code that is to be executed at these join-points.

Dynamic adaptation of aspects here implies dynamic weaving and unweaving of aspects into / from an application. In the last few years, a number of approaches for dynamic weaving and unweaving of aspects have been proposed and implemented. These approaches are commonly referred as dynamic AOP approaches. CASA relies on a flexible and efficient dynamic AOP system called PROSE [NA05]. Below we briefly describe the PROSE system, and its role in the CASA framework.

PROSE is a Java-based dynamic AOP system. The architecture of PROSE is presented in Figure 4.5. The architecture is divided into an *AOP engine* layer, and an *execution monitor* layer. The AOP engine is responsible for managing aspects and executing advices, while the execution monitor deals with weaving of aspects at the join-points.

Figure 4.5 illustrates the steps involved in weaving and execution of aspects. For more details on these steps, please refer to [NA05].

PROSE does not define any new aspect language, and uses Java as the aspect language. An aspect file in PROSE defines the advice code, and the corresponding set of join-points where the advice code is to be executed. An example PROSE aspect is shown in Figure 4.6 (reproduced from [NA05]). This aspect redefines the original version of a method with a new one.

Every aspect definition contains one or more crosscut objects. As shown in

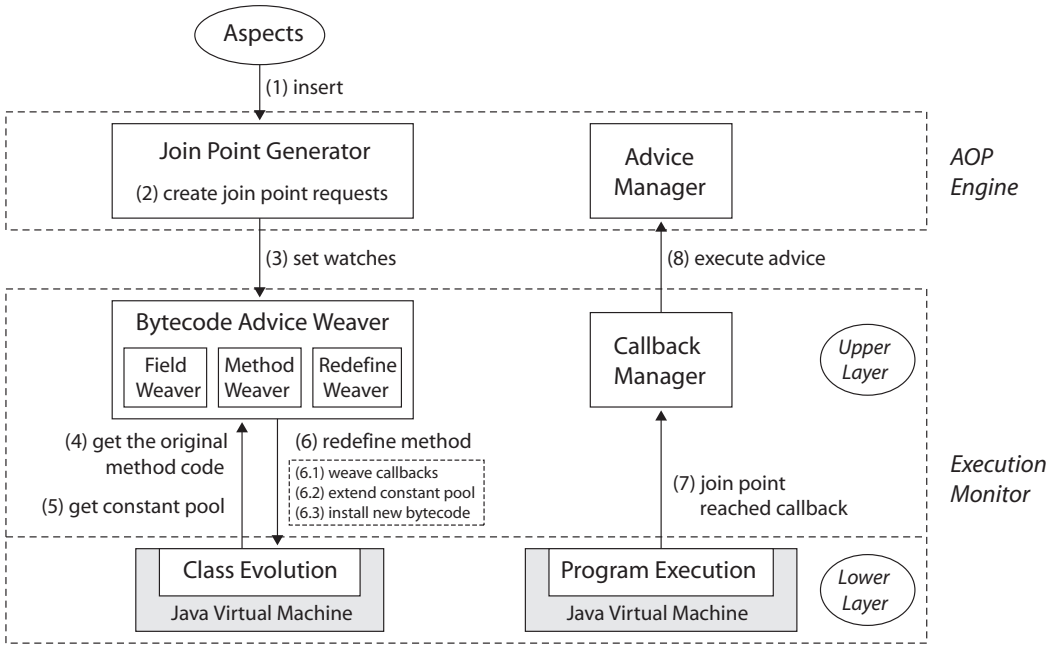


Figure 4.5: Architecture of PROSE [NA05]

```
public class ExampleAspect extends DeafultAspect {
      public Crosscut doRedef = new MethodRedfineCut() {
            public void METHOD_ARGS(Foo ob, int x) {
                  // the new method code
            }
            protected PointCutter pointCutter() {
                  return Within.method("bar");
            }
      };
}
```

Figure 4.6: Example of a PROSE aspect [NA05]

Figure 4.6, a crosscut object defines an advice method (called `METHOD_ARGS()`) and a pointcut method (called `pointCutter()`). The pointcut method defines a set of join-points where the advice should be executed. The aspect shown in Figure 4.6 specifies replacing the method `bar` from the `Foo` class with the code specified in the advice.

The AOP engine accepts aspect files, extracts information about join-points, and sends join-point requests to the execution monitor. The execution monitor performs bytecode manipulations and method instrumentations for activating join-points by adding advice callbacks at the corresponding bytecode locations. For advices that are executed externally, when the program execution reaches one of the activated join-points, the execution monitor notifies the AOP engine which then executes the corresponding advice. When aspects are removed, the join-points are deactivated and the original bytecode of methods is installed. Detailed working of PROSE can be found in [NA05].

The PROSE system supports a number of aspect weaving mechanisms, namely: stub weaving, advice weaving, hook weaving and stop-point based weaving. The first two weaving mechanisms, stub weaving and advice weaving, are of particular interest here. In stub weaving, the original bytecode is augmented with stubs by the execution monitor, which then call external advice codes. Whereas in advice weaving, the original bytecode is overwritten to include the advice code. These two mechanisms together provide flexibility for weaving different types of aspects and join-points, according to the application's requirements. The other two weaving mechanisms, hook weaving and stop-point based weaving, were implemented in the earlier versions of PROSE. Details of these two weaving mechanisms can be found in [PAG03] and [PGA02] respectively. However, stub weaving and advice weaving provide better performance as well as flexibility when compared to the other two weaving mechanisms.

The join-points supported in PROSE are method boundaries (method entry and exit), method redefinition, field access and modification, and exception (catch and throw).

The Aspects Adaptation System (AAS) in CASA interacts with PROSE for dynamically weaving and unweaving aspects. In the application contract, the <`aspects`> element corresponding to a configuration contains a number of <`aspect`> elements. Every <`aspect`> element contains a `name` attribute specify-

ing the name of an aspect (this name is used for internal reference only), and a `reference` attribute specifying the reference to the corresponding aspect definition.

Once an application configuration is selected for activation, the CES communicates the new configuration to the AAS (Aspects Adaptation System). The AAS discovers which aspects are new (i.e. part of incoming configuration but not of outgoing configuration) and thus need to be added, and which aspects are outdated (i.e. part of outgoing configuration but not of incoming configuration) and thus need to be removed. If the above configuration is the first configuration of the application, then obviously all the aspects specified within the corresponding <`aspects`> element need to be added. The AAS forwards the details of the identified aspects to PROSE for dynamically weaving and unweaving these aspects.

The PROSE system provides an Aspect Manager for managing the dynamic weaving and unweaving of aspects. The Aspect Manager provides `insert()` method (for weaving aspects), `withdraw()` method (for unweaving aspects) and `getAllAspects()` method (for getting a list of all active aspects). The AAS interacts with the Aspect Manager for dynamic weaving and unweaving of aspects. That is, the AAS invokes the `insert()` and `withdraw()` methods according to its decision for weaving and unweaving the corresponding aspects.

Figure 4.7 shows interactions between the AAS and the Aspect Manager for weaving an aspect. The rest of the steps involved in the weaving and execution of an aspect by the PROSE system are as shown in Figure 4.5. For more details on these steps and working of PROSE, please refer to [NA05].

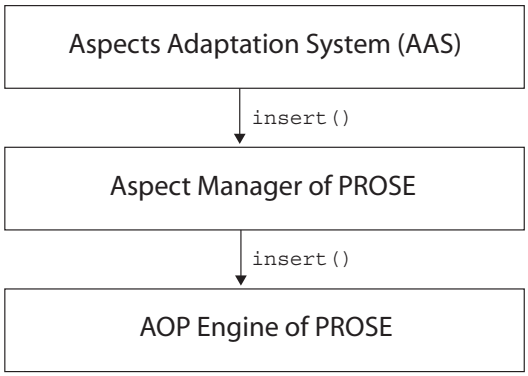


Figure 4.7: Interactions between the AAS and PROSE

As an example, let us assume that an application has moved from a low-risk area to a high-risk area, and accordingly wants to weave an access control aspect. The purpose of this aspect is to control the access to certain critical methods of the application. Therefore, the join-points for this aspect are the entry points to these restricted-access methods. Whenever a call to one of the restricted-access methods is made, the corresponding access control advice is executed to authenticate the access before the call is allowed to execute the method.

Weaving the access control aspect can be done by adding a stub to this aspect before the original bytecode of every restricted-access method. The stub is woven before the first instruction of the method but after the call to this method has taken place, so that the stub can inspect the stack, extract the parameters of the call and pass these parameters to the advice code. The arguments of the called method are accessible to the advice, because these are passed as parameters to the advice code.

An example access control aspect as specified in the application contract is shown in Figure 4.8. The `name` attribute of the <`aspect`> specifies the name of the aspect. This name is used for internal reference only. The `reference` attribute specifies the location of the compiled aspect file. The aspect file contains the executable advice code and the join-points where this code is to be executed.

```
<aspect name="access-control"
        reference="/aspect-folder/AccessControl"/>
```

Figure 4.8: Example access control aspect

As the application moves from a low-risk area to a high-risk area, the CASA Runtime Systems detects this change in context and select a new configuration of the application. The change in configuration includes weaving the access control aspect into the application. The AAS invokes the `insert()` method of the Aspect Manager of PROSE for weaving this aspect. An instance of the corresponding aspect as specified in the application contract is passed as an argument to the `insert()` method. Similarly, the AAS can invoke the `withdraw()` method of the Aspect Manager for unweaving an already weaved aspect, if this aspect is no longer required as a result of the change in application configuration. In addition to the advice code and join-points, the aspect code contains `insertionAction()` method which is called before/after the aspect is inserted, and `withdrawalAction()` method which is called

before/after the aspect is withdrawn. PROSE offers very good runtime performance, the details of which can be found in [NA05].

Chapter 5

Dynamic Recomposition of Components

5.1 Introduction

In this chapter, we discuss the CASA approach for dynamic recomposition of components.

We start with the definition of a software component, or just component as in this dissertation. In literature, various definitions of a component are available. For our purpose, we use the definition provided by Clemens Szyperski [Szy98]: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

A component composition, or just composition, is a collection of components that are able to jointly carry out the given task of the application, in conjunction with the other appropriate parts of the application configuration i.e. aspects, lower-level services etc.

A typical dynamically adaptive application is required to carry out different tasks under different states of its execution environment. Accordingly, an application may have a number of alternative compositions, only one of which will be valid at a given time depending on the current state of the execution environment.

A composition is specified within the <`components`> element in the corresponding <`config`> element in the application contract.

Dynamic recomposition of components can now be defined as changing from one

alternative composition of an application to another at runtime.

Any two alternative compositions of an application may vary in just a few components, while the rest of the components remain the same across both the compositions.

When changing from one alternative composition to another, there may be some new components to be added and some old components to be removed. In other words, a dynamic recomposition may involve any number of dynamic addition and/or dynamic removal of components.

There may be some components that remain the same across all the alternative compositions of an application, and provide some common core functionality. The components that remain the same across all the alternative compositions of an application are called non-adaptable components, while all other components that can be added or removed dynamically are called adaptable components.

It is obvious that only the adaptable components constituting a composition need to be specified within the corresponding <`components`> element in the application contract.

Dynamic replacement of components is a special case of dynamic removal of a component A followed by dynamic addition of a component A', such that the external components being served by A can now be served by A', without requiring any change in the external components themselves.

In order for components A and A' to be dynamically replaceable[1], both A and A' must conform to the same component contract[2].

Conforming to the same component contract implies that the following two requirements need to be satisfied by A and A'.

Requirement 1: Both A and A' must have the same interface, i.e. the method signatures of the publicly-accessible methods of A and A' must be the same.

Requirement 2: The pre and post conditions of the publicly-accessible methods of A and A' must be the same (the pre and post conditions may also include certain

[1] In CASA, dynamic replaceability is a reflexive relation. That is, if A can be dynamically replaced by A', then it automatically implies that A' can also be dynamically replaced by A.

[2] The term "component contract" is used here in the sense of the Design by Contract approach [Mey92]. Please note that even while conforming to the same component contract, the components are free to differ in their internal implementations.

non-functional assertions or constraints).

The Components Adaptation System (CAS) in CASA is responsible for dynamic recomposition of components.

Please note that ensuring the completeness and correctness of various alternative compositions of an application (as specified in the application contract for different states of the application's execution environment) is the responsibility of the application developer. The CAS is responsible for simply carrying out a change from one alternative composition to another at runtime.

The completeness of a composition implies that the following requirement be satisfied by the composition.

Requirement 3: For every component present in a composition, all the components on which this component depends must also be present in the same composition.

In terms of dynamic recomposition, requirement 3 can be divided into following two sub-requirements.

Requirement 3a: If a component A is removed as a part of a dynamic recomposition, then either A must be dynamically replaced by another component, or else all the components depending on A must also be removed as a part of the dynamic recomposition.

Requirement 3b: If a component A is added as a part of a dynamic recomposition, then the components on which A depends either must already be present in the old composition, or else they must also be added as a part of the dynamic recomposition.

The correctness of a composition implies that the following requirement be satisfied by the composition.

Requirement 4: A composition must be able to carry out its stated task correctly, where the correctness is defined by the ability of the composition to satisfy the functional and non-functional requirements of the application in the corresponding state of the execution environment, in conjunction with the other appropriate parts of the application configuration i.e. aspects, lower-level services etc.

In the following sections, we discuss the implementation issues related to dynamic recomposition of components for applications developed using object-oriented programming languages. In particular, we consider Java as a target language, because of its widespread use and popularity. However, we will try to keep our discussion as language-neutral as possible, so that the results are applicable for a wide range of object-oriented programming languages.

Components vs. objects: Before discussing the implementation details of the dynamic recomposition of components, we need to understand how components are implemented in an object-oriented programming language. In the literature, there is often confusion over the terms 'components' and 'objects', with the result that these two terms are frequently used to denote similar abstractions. Sometimes these two terms are used so interchangeably that it gives an impression that a component is merely a commercial name for an object. At best, the difference between a component and object is recognized by some as a component being a collection of objects, and by some others as a component being an object decorated with some bells and whistles (such as the degree of encapsulation being high in components, and an explicitly defined interface being a necessity for components).

In a panel discussion on "Are Components Objects?" at the OOPSLA'99 conference, Clemens Szyperski [OOP99] helped explain the difference between these two terms. According to Szyperski,

> "Objects form abstractions over identifiable parts of a state space: they have a unique identity and they encapsulate the variables and operations over those variables that define the abstracted part of the state space. Hence, no two objects occupy the same partition of the universal state space.
>
> Components form abstractions over namable parts of a deployment space: they have a name but not a unique identity; they encapsulate static implementation decisions and are restricted to explicitly specified dependencies on other components only and, preferably, have dependencies that are configurable."

The difference between a component and object is more clearly discussed and explained by Luigia Petre in [Pet00]. According to Petre,

> "Components are important as software construction concepts based on the notion of service. Objects are also important, but in providing a construction solution to each component. Objects are best in underlying (modeling) a problem domain, while components are most suitable in describing a system functionality."
>
> "Software systems grow continuously and need to evolve. The notion of component as a self-contained software system that can be reused or easily replaced meets these requirements. The implementation of a component is however best realized when objects are used."

The above implies that the terms 'component' and 'object' are used for different purposes in a software development and deployment process. Component is essentially a modularization concept that is used for organizing an application into interacting reusable components. Whereas, objects are used for modeling a problem domain, i.e. modeling things and relationships between things, and implementing solutions for this domain.

A component may or may not be implemented using object-oriented programming technologies. Even when a component is implemented in an object-oriented programming language, there may not be a direct correspondence between a component and an object. That is, a component can be implemented as a single object or as a collection of objects.

When a component is implemented as a collection of objects, these objects are usually hierarchically structured. That is, a top-level object instantiates other objects for its use. These objects may in turn instantiate more objects for their own use. The hierarchical structuring of these objects results in a tree with a single root object, as illustrated in Figure 5.1. The parent-child relation shown in Figure 5.1 represents the relation between the creator and the instance being created.

In this dissertation, we assume that a component in CASA is implemented in an object-oriented programming language. A component can be implemented as a single object or a collection of objects. Adding or removing a component implies adding or removing all the corresponding objects constituting the component. All the objects that need to be added or removed dynamically by the CASA Runtime System need to be specified in the application contract. However, if a component is implemented as a collection of hierarchically-structured objects, then only the root

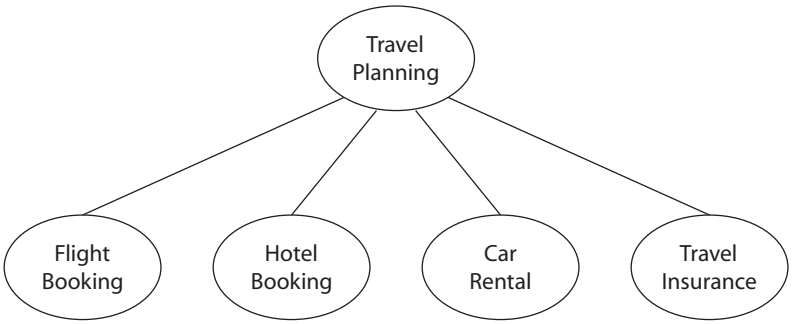


Figure 5.1: Hierarchical structuring of objects

object needs to be added or removed by the CASA Runtime System. Every other object in the hierarchical collection is added or removed by its parent object, using the facilities for dynamic addition and removal of objects provided in the modern object-oriented programming languages. Only the objects that need to be added or removed explicitly by the CASA Runtime System are specified in the application contract.

A dynamic recomposition involves adding, removing and/or replacing components dynamically. Dynamic replacement of components is of particular interest here, as it is more critical than the addition or removal of components which is relatively straightforward to carry out. We discuss details of dynamic replacement of components in Section 5.2, and that of dynamic addition and removal of components in Section 5.3. A discussion of sequential vs. atomic recomposition is given in Section 5.4, followed by a concluding discussion in Section 5.5.

5.2 Dynamic replacement of components

In principle, there are two possible strategies for dynamic replacement of components: lazy replacement and eager replacement. Below we briefly discuss the two.

Lazy replacement: In this strategy, once the decision for dynamic recomposition is taken, if a component to be replaced is currently running then this component is allowed to complete its current execution before being replaced by a new component.

Eager replacement: In contrast to the lazy replacement strategy, here the execution of a currently running component is suspended once the decision for dynamic recomposition is taken, and the execution resumes on the new component (from the point where it was suspended) after the component has been replaced.

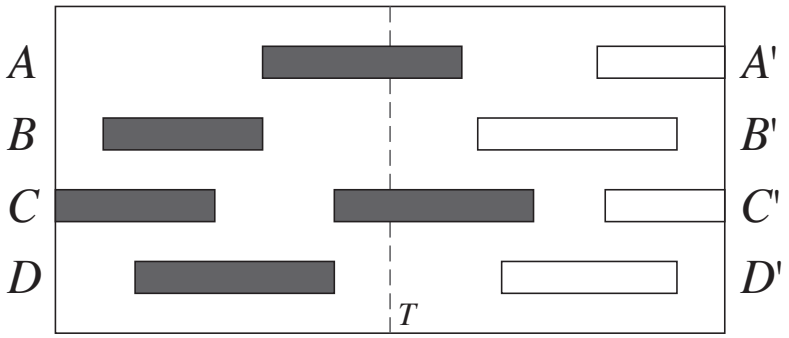
Figure 5.2 illustrates lazy replacement (Figure 5.2a) and eager replacement (Figure 5.2b). In Figure 5.2, the horizontal axis represents the time line, and the vertical dashed line represents the time T when the decision for dynamic recomposition is taken. In this example, the components A, B, C and D are to be replaced by the components A', B', C' and D' respectively as a result of dynamic recomposition (dark bars denote the execution of old components, and light bars denote the execution of new components). Only the components A and C are under execution at time T. In Figure 5.2a (representing lazy replacement) A and C are allowed to complete their execution before being replaced by A' and C' respectively. Whereas in Figure 5.2b (representing eager replacement), the execution of A and C is suspended at time T, they are replaced by A' and C' respectively, and the execution resumes on A' and C'.

Since the eager replacement strategy is able to give a faster response to a change in the execution environment than the lazy one, we decide in favor of eager replacement for CASA. However, as discussed later in Section 5.2.4, it may not always be possible to use eager replacement, and thus sometimes lazy replacement may be the only option.

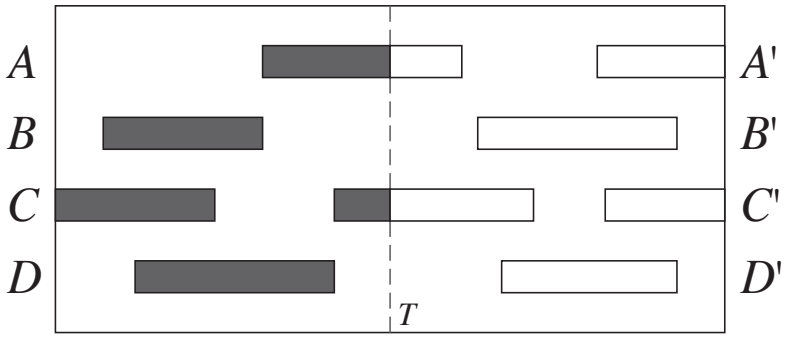
In Figure 5.2b (representing eager replacement), it is shown that the replacement from an old component to a new component takes place almost instantaneously at time T, and there is no break in the execution due to the replacement. However, this is just a theoretical representation of the eager replacement strategy. In practice, there will always be some delay after time T when the old component is actually suspended, and also there will be a small time interval during which the execution remains suspended before it may be resumed on the new component. The detailed dynamic replacement process is discussed next.

5.2.1 Dynamic replacement process

As discussed in Section 5.1, a component can be implemented as an object or a collection of objects in an object-oriented programming language. Thus replacing a



a. Lazy replacement strategy



b. Eager replacement strategy

Figure 5.2: Replacement strategies

component implies replacing the objects constituting the component. Replacing an object is equivalent to replacing the class definition of the instance representing the object. Below we discuss the process of replacing the class definition of an instance at runtime. In the following, the term *instance* refers to a class instance or object.

We place a restriction that an instance cannot have any externally-visible state. This restriction is also in accordance with the good software engineering practice which says that the state of an instance should be accessed using methods like `getState` and `setState` only, and not accessed directly from external objects.

We now define a *replaceable class* as one whose instances are dynamically replaceable i.e. can replace, or be replaced by, instances of other classes dynamically.

Additionally, we define a *set of alternative classes* as a collection of replaceable classes whose instances can dynamically replace each other.

This implies that all replaceable classes which are members of the same set of alternative classes (and by implication, the instances of these replaceable classes) conform to the requirements 1 and 2 identified above.

We further impose the following constraints, in order to simplify our implementation process:

(i) An instance of a class can be dynamically replaced by an instance of another class only if both these classes belong to the same set of alternative classes.

(ii) Any given composition may contain instances of only one of the replaceable classes from any given set of alternative classes. That is, no two instances in a given composition can be of two different classes belonging to the same set of alternative classes.

We believe that the above two constraints are quite reasonable to impose. The first constraint concerns only the way a set of alternative classes is defined. And the second constraint simply states that at any given time (i.e. for any given state of the execution environment), only one of the classes from a given set of alternative classes will be valid, which is intuitively what will be required in practice anyway.

We use a variant of the Bridge pattern [GHJV95] for hiding the complexities of dynamic replacement from the application code. The use of the variant of Bridge pattern works as follows.

Every set of alternative classes is associated with a unique *Handle* class. The Handle class conforms to the same interface as the replaceable classes in its associated set.

The Handle class acts as an abstraction that can be bound to any of the replaceable class implementations from its associated set of alternative classes at runtime (the terms *abstraction* and *implementation* are used here in the sense of the Bridge pattern [GHJV95]).

From the constraint (ii) above, we know that for any specific composition, only one of the replaceable classes from any given set of alternative classes will be valid. Since every set of alternative classes has a unique Handle class associated to it, we can conclude that for any specific composition there is a unique replaceable class bound to a given Handle class.

In order to provide a layer of transparency between the application code and the dynamic replacement process, wherever there is a need for creating an instance of a replaceable class in the application code, an instance of the corresponding Handle class is created instead. This Handle class instance is then linked to an instance of the replaceable class that is currently bound to the Handle class, at runtime. Details of establishing a link between the Handle class instance and the replaceable class instance are discussed later.

The relationship between an instance of a Handle class, an instance of the corresponding replaceable class, and the external application code interested in invoking an operation on the replaceable class instance is shown in Figure 5.3.

Figure 5.3: Invoking an operation through a Handle class instance

The binding between a Handle class and its corresponding replaceable class for a given composition is specified as a part of the composition specification in the application contract (refer Figure 5.4). In Figure 5.4, `HC` is the Handle class, which is bound to the replaceable class `CdefA` (with namespace location `/class-folder/CdefA`) for the configuration with id 1.

Let a set of alternative classes S consist of the replaceable classes `CdefA`, `CdefB` and `CdefC`, and the associated Handle class for the set S be `HC`. At any given time, `HC` will be bound to a unique replaceable class from the set S, depending on the currently active composition. However, this binding may change dynamically as a result of dynamic recomposition.

In the application code, when a new instance `instHC` of the Handle class `HC` is created (refer Figure 5.5), the constructor of `instHC` invokes the CAS (Components Adaptation System) (refer Figure 5.6).[3]

The CAS first registers `instHC` for future recompositions. Next, the CAS gets the

[3]Please note that in this chapter, we will show only some small and simplified code snippets in order to make the discussion easy to follow.

```
<config id="1">
        ... ... ...
        ... ... ...
        <components>
                <binding handle="HC" boundto="/class-folder/CdefA"/>
                ... ... ...
                ... ... ...
        </components>
        ... ... ...
        ... ... ...
</config>
```

Figure 5.4: Specification of a <binding> element

```
void externalCode() {
        ... ... ...
        ... ... ...
        HC instHC = new HC();
        ... ... ...
        ... ... ...
}
```

Figure 5.5: Creation of a Handle class instance

information about the replaceable class currently bound to `HC`, say `CdefA`, from the specification of the currently active composition, and returns this information back to the constructor of `instHC` (this class is stored as `currentBinding` in Figure 5.6). The constructor of `instHC` creates an instance of this class and stores it internally as currently active instance, say `activeInst`.

Although a Handle class conforms to the same interface as the classes in its associated set of alternative classes, it does not provide a real implementation for any of the methods in this interface. The methods of a Handle class instance simply forward the method calls invoked on them to the corresponding methods of the currently active instance, and return the results as received from the latter. For example, if a method `foo()` is invoked on `instHC`, then `instHC.foo()` simply invokes the method `activeInst.foo()`, and returns the result as received from `activeInst.foo()` (re-

```
class HC {
        ... ... ...
        ... ... ...
        HC() {
                ... ... ...
                Class currentBinding = CAS.register(this);
                activeInst = currentBinding.newInstance();
                ... ... ...
        }
        ... ... ...
        ... ... ...
}
```

Figure 5.6: Handle class instance invoking the CAS and creating `activeInst`

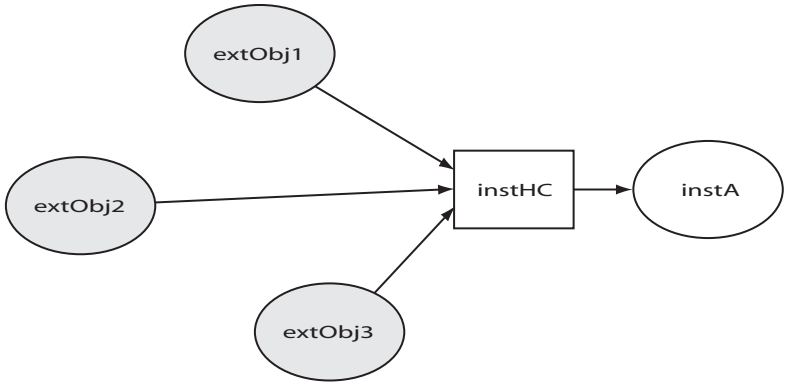
fer Figure 5.3).

If there is a change in the binding between the Handle class `HC` and its corresponding replaceable class, due to dynamic recomposition, then the CAS passes the information about the newly bound replaceable class, say `CdefB`, to all the instances of `HC` (including `instHC`). The instances of `HC` then replace the old replaceable class instances (i.e. instances of `CdefA`) with the instances of `CdefB` as active instances (the sequence of steps for this replacement is discussed next).

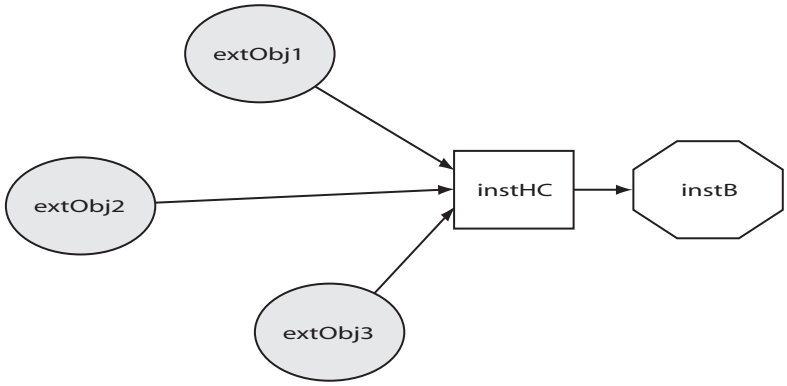
The calls to an instance of `HC` will now be forwarded automatically to the newly designated active instance in place of the old one. This way, the Handle class instances help to hide the details of dynamic replacement from the application code.

Figure 5.7 illustrates the above example of dynamic replacement. In Figure 5.7a, the Handle class instance `instHC` is linked to the old instance (an instance of `CdefA`), just before the dynamic replacement is carried out. And in Figure 5.7b, `instHC` is linked to the new instance (an instance of `CdefB`), just after the dynamic replacement is over. The external objects (`extObj1`, `extObj2` and `extObj3`) are not affected by this dynamic replacement, as their links to `instHC` remain undisturbed by the change.

Below we discuss the sequence of steps to be carried out when replacing an old instance (instance of `CdefA`) with a new instance (instance of `CdefB`) as the currently active instance (according to the eager replacement strategy).



a. Before dynamic replacement



b. After dynamic replacement

Figure 5.7: Dynamic replacement process

Sequence of steps:

1. Deactivate the old instance
2. Suspend the execution of the old instance
3. Create the new instance
4. Transfer the state of the old instance to the new instance
5. Activate the new instance

```
class HC {
        ... ... ...
        ... ... ...
        public boolean replace(Class newBinding) {
                oldInst = activeInst;
                activeInst = null;
                ... ... ...
                ... ... ...
        }
        ... ... ...
        ... ... ...
}
```

Figure 5.8: Handle class instance nullifying `activeInst`

Below we discuss the implementation of the above-mentioned steps.

Step 1: Deactivate the old instance

First, on receiving an instruction from the CAS about dynamic replacement, `instHC` deactivates the reference to the old instance, which is the currently active instance. This is done practically by nullifying `activeInst`. However, before nullifying `activeInst`, the reference to `activeInst` is stored in a temporary instance, say `oldInst`, for the purpose of rest of the steps required for instance replacement (refer Figure 5.8).

Nullifying `activeInst` ensures that the calls made to `instHC` during the instance replacement process are not forwarded to the old instance. These new calls are actually queued within `instHC` until the completion of the instance replacement process, after which they are forwarded to the new instance (i.e. these calls are stored in a queue within `instHC` waiting for the `activeInst` to be non-null, before being forwarded to the new instance).

Step 2: Suspend the execution of the old instance

Suspending the execution of the old instance implies suspending all the calls currently executing on the old instance.

In order to keep our discussion in this section easy to follow, we assume that at most one call will be executing on the old instance at any given time (i.e. including

the time of instance replacement). The case for multiple calls executing concurrently on the old instance is discussed in Section 5.2.3.

It is possible that at the time of instance replacement, there is actually no call executing on the old instance. In such a case this step will not be required (`instHC` keeps a record of the calls currently executing on the old instance). The discussion below assumes that there is a call executing on the old instance at the time of instance replacement.

Before actually suspending the call executing on the old instance, it needs to be ensured that the execution of the call has reached a "safe" point, from where it can be resumed correctly by the new instance at the end of instance replacement. For this purpose, the safe points need to be explicitly defined in the body of the old instance. More discussion on safe points follows in Section 5.2.4.

After deactivating the reference to the old instance (step 1), `instHC` sets a signal for the suspension of the old instance (refer Figure 5.9). This is done practically by setting a `suspend` flag in the old instance (the `suspend()` method in Figure 5.9 is used for setting the `suspend` flag). At every safe point, the call executing on the old instance checks if the `suspend` flag has been set. The setting of the `suspend` flag is an indication for the call to suspend its execution and return to `instHC`.

The suspended call is made to wait within `instHC` until the completion of the instance replacement process, after which the call is reinvoked on the new instance.[4]

Step 3: Create the new instance

After setting the signal for the suspension of old instance, `instHC` creates an instance of the new replaceable class passed by the CAS, i.e. the new instance, say `newInst` (refer Figure 5.9). The creation of the new instance may take place while step 2 is still on, i.e. during the time that the call executing on the old instance reaches a safe point and returns to `instHC`.

Step 4: Transfer the state of the old instance to the new instance

This step consists of two sub-steps: capturing the state of the old instance, and initializing the state of the new instance with this captured state.

[4] `instHC` is responsible for reinvoking the suspended calls with correct method argument values. In Java, the method argument values for each call are stored locally in the thread stack. These method argument values are used for correctly reinvoking the suspended calls on the new instance.

```
class HC {
        ... ... ...
        ... ... ...
        public boolean replace(Class newBinding) {
                ... ... ...
                ... ... ...
                oldInst.suspend();
                newInst = newBinding.newInstance();
                ... ... ...
                ... ... ...
        }
        ... ... ...
        ... ... ...
}
```

Figure 5.9: Handle class instance suspending the old instance and creating the new instance

Before discussing further about the state transfer, we first define the *transient* and *persistent* states of an instance. In general, the transient state is defined as the temporary state of an instance at any time during the execution of the instance (i.e. while a call is executing on the instance), while the persistent state is defined as the state of the instance that needs to remain persistent in between consecutive executions of the instance. In other words, persistent state is the state maintained by an instance when no method call is currently executing on this instance.

For our purpose, we practically define the persistent state to consist of any global state of an instance, and the transient state to be made up of the local states of the currently executing methods of the instance. The global state of an instance is initialized at the time of the instance creation. The global state is shared by all the methods of the instance, and is required to be persistent. The local state of a method of the instance is initialized every time the method is invoked, and is not required to be persistent.

The state of the old instance must be transferred to the new instance, so that the new instance is able to correctly resume the execution at the end of dynamic replacement. The state here refers to both the transient as well as persistent states of the instance. It is obvious that if the old instance is not executing at the time of

```
class CdefA {
        ... ... ...
        ... ... ...
        // this is a safe point
        if(suspend) {
                // code for storing transient state
                // in a transient state object
        }
        ... ... ...
        ... ... ...
}
```

Figure 5.10: Storing the transient state at a safe point

instance replacement, then there will be no transient state and only the persistent state of the old instance will need to be transferred to the new instance.

Below we discuss the procedures for transferring the transient and persistent states.

4a. Transferring the transient state

Please recall from step 2 that `instHC` sets a `suspend` flag in the old instance as a signal for the instance suspension. At every safe point, the call executing on the old instance checks if the `suspend` flag has been set. If the `suspend` flag is set, then, before the call returns to `instHC`, the transient state of the old instance is stored in a transient state object (refer Figure 5.10). Some additional information, e.g. the information about the safe point where the call is suspended, is also stored in the transient state object, as this information may be required by the new instance to resume the execution of this call correctly. More discussion on transferring the transient state follows in Section 5.2.2.

In the practical implementation, an exception is thrown on the call being suspended at a safe point, to be eventually caught by `instHC`. This exception serves as an indication to `instHC` that the call has returned after being suspended at a safe point, and not normally at the end of its regular execution. The transient state object containing the transient state of the old instance is also passed to `instHC` along with the exception. Eventually, this transient state object is forwarded by `instHC` to the new instance.

It is possible that the call executing on the old instance returns to `instHC` normally at the end of its regular execution, and not after being suspended at a safe point. In this case, the call is simply returned back to its original caller, and only the persistent state of the old instance needs to be transferred to the new instance (as in step 4b).

4b. Transferring the persistent state

For transferring the persistent state, i.e. storing the state and loading the state, every dynamically replaceable instance needs to provide appropriate `storeState` and `loadState` methods. This is because state parameters (names and types) of the persistent state may vary across the old and new instances, which means that the semantic information necessary for transferring the persistent state can be provided by the respective instances only.

As the name indicates, the `storeState` method is responsible for storing the persistent state of the old instance in a persistent state object. And the `loadState` method is responsible for loading the persistent state of the new instance using the state information stored in the persistent state object.

The `storeState` method of the old instance may need to convert its own internal representation of the persistent state into a standard representation (standard for the corresponding set of alternative classes), which the `loadState` method of the new instance understands and may convert into its own internal representation.

If the old instance is executing at the time of instance replacement, `instHC` waits until the call executing on the old instance returns to `instHC` (either after being suspended, or normally at the end of its regular execution) before initiating the persistent state transfer.

Once the execution of the old instance is halted, `instHC` calls the `storeState` method of the old instance. The `storeState` method returns a persistent state object back to `instHC`, containing the persistent state of the old instance. Next, `instHC` calls the `loadState` method of the new instance passing it the persistent state object received from the old instance. Using the state information stored in the persistent state object, the `loadState` method should be able to initialize the persistent state of the new instance correctly.

```
class HC {
        ... ... ...
        ... ... ...
        public boolean replace(Class newBinding) {
                ... ... ...
                ... ... ...
                activeInst = newInst;
                oldInst = null;
                newInst = null;
        }
        ... ... ...
        ... ... ...
}
```

Figure 5.11: Handle class instance setting the new instance as `activeInst`

Step 5: Activate the new instance

The new instance is now prepared to be used as the active instance by `instHC`. Therefore, `instHC` now sets the new instance as the currently active instance (refer Figure 5.11).

If the execution of the old instance was suspended in step 2 above, then the suspended call is now reinvoked on the new instance. The new instance should be able to resume the execution of this call correctly, because the transient state of the old instance at the time of suspension, as well as the persistent state of the old instance, have already been transferred to the new instance.

Any new calls that were made during the instance replacement process, which were consequently queued within `instHC`, are also forwarded now to the new instance.

The reference to the old instance is now completely deleted (by nullifying `oldInst`), making way for the garbage collection of the old instance.

5.2.2 Transient state transfer from the old instance to the new instance

As described in step 4 of the instance replacement process, the transient state of an instance is stored in a transient state object at the safe point where the call is suspended.

```
class CdefA {
        ... ... ...
        public void foo() {
                ... ... ...
                ... ... ...
                bar();
                ... ... ...
                ... ... ...
        }
        public void bar() {
                ... ... ...
                ... ... ...
                // this is a safe point
                if(suspend) {
                        // code for storing transient state
                        // in a transient state object
                }
                ... ... ...
                ... ... ...
        }
}
```

Figure 5.12: The `foo()` method of the currently active instance calling the `bar()` method

It is possible that the call being suspended has reached the current method (where the safe point is located) not directly from `instHC`, but after being forwarded from some other methods.

For example, consider a call invoked by an external object on the `foo()` method of `instHC`. The `foo()` method of `instHC` forwards this call to the `foo()` method of the currently active instance. During the execution of the `foo()` method of the currently active instance, a call may be made to the `bar()` method of this instance (refer Figure 5.12). Now, during the execution of the `bar()` method, the call may reach a safe point where the `suspend` flag is already set indicating that the call needs to suspend its execution and store the transient state.

The transient state to be stored at the safe point in this example consists of the

local state of `bar()` method and the local state of `foo()` method. In addition to these states, the new instance will also need information about the specific safe point (note that there may be several safe points in an instance) where the call is suspended, in order to be able to resume the execution of this call correctly. Therefore, this additional information also needs to be stored as a part of the transient state of the instance.

Forwarding a call from one method to another within an instance can also be viewed as going a level deeper in the instance. At the time of suspension, a call might be at an arbitrary level of depth within the instance.

Once at the safe point indicating its suspension, the call needs to store the local state of the current level (i.e. the current method), and return to the next higher level (i.e. the method that last forwarded this call). Upon reaching the higher level, the call needs to store the local state of this higher level, and then move to the still higher level. This process goes on till the call finally reaches the highest level within the instance, and from there it returns to `instHC`.

When the transient state object is passed to the new instance, the new instance needs to reconstruct its own transient state in the reverse order, i.e. from the top level to the level where the reinvoked call actually resumes its execution.

Because of an arbitrary number of levels involved in storing and loading the transient state, we follow a stack-based approach for storing and retrieving the state information. In addition to the stack containing the state information, the transient state object also needs to store the unique call identifier (for Java, it can be the thread identifier) of the call being suspended. This call identifier will be required by the new instance in order to resume the execution of the call correctly on its reinvocation.

Continuing with our example above, at the safe point where the call needs to be suspended, a new transient state object is created (refer Figure 5.13). First, the unique call identifier of the call is stored in the transient state object. Next, a new stack entry is created. This stack entry will contain the unique execution point identifier of the safe point, and the local state of the `bar()` method.

After storing the above state information in a stack entry, an exception is thrown on the call, to be caught by the method that last forwarded the call, i.e. the `foo()` method in our example. The transient state object is passed along with this exception.

```
class CdefA {
        ... ... ...
        public void foo() {
                ... ... ...
                ... ... ...
                // this execution point has an ID = F2
                try {
                        bar();
                } catch(SuspendException ex) {
                        TransientState ts = ex.getTransientState();
                        ts.newStackEntry();
                        ts.storeExecPointID(F2);
                        ts.storeState(x, y);
                        throw ex;
                }
                ... ... ...
                ... ... ...
        }
        public void bar() {
                ... ... ...
                ... ... ...
                // this safe point has an ID = B5
                if(suspend) {
                        TransientState ts = new TransientState();
                        ts.storeCallID(Thread.currentThread.getId());
                        ts.newStackEntry();
                        ts.storeExecPointID(B5);
                        ts.storeState(a, b);
                        throw new SuspendException(ts);
                }
                ... ... ...
                ... ... ...
        }
}
```

Figure 5.13: Storing the transient state in a transient state object

Every execution point where a call can be forwarded to another method which contains a safe point needs to be identifiable by a unique execution point identifier. In the above example, this means that the execution point within the `foo()` method where a call to the `bar()` method is made needs to have a unique execution point identifier. This execution point identifier, in addition to the safe point identifier, will be required by the new instance to resume the execution of this call correctly.

After catching the exception within the `foo()` method, a new stack entry is created in the transient state object. The unique identifier of the execution point where the `bar()` method is called, as well as the local state of the `foo()` method is stored in this stack entry. Thereafter, an exception is thrown on this call to be caught by the method at the next higher level. In this example, `foo()` is the highest level method for this call, therefore the exception will be caught by `instHC`. The transient state object is passed along with this exception, and is forwarded to the new instance by `instHC`.

When the call is reinvoked at the end of the instance replacement process, the new instance uses the information provided in the transient state object to reconstruct the transient state correctly, and to forward the call to the appropriate safe point in the new instance.

In our example, when the call is reinvoked on the `foo()` method of the new instance, first the local state of the `foo()` method is initialized using the state information stored in the topmost stack entry. Thereafter, the call is forwarded to the appropriate execution point within the `foo()` method, which is also stored in the stack entry. At this execution point, the call to the `bar()` method is made. Within the `bar()` method, the local state of the `bar()` method is initialized, and the call is forwarded to the appropriate safe point where the execution of the call can finally be resumed.

It is possible that the names and types of the state parameters constituting the transient state may vary across the mutually replaceable instances. In this case, the outgoing instance may need to convert its own internal representation of the transient state into a standard representation (just like in the case of transferring the persistent state), which the incoming instance is able to understand and convert into its own internal representation.

5.2.3 Multiple calls executing concurrently on the old instance

In case of multiple calls executing concurrently on the old instance at the time of instance replacement, the instance replacement process will be only slightly different than the one discussed above for a single call. Below we discuss the required modifications in the instance replacement process for multiple calls.

It is obvious that every call executing on the old instance needs to suspend its execution at a safe point, and return to `instHC`.[5]

Every call stores its own transient state in a separate transient state object. `instHC` keeps a record of all the calls currently executing on the old instance, and waits till all these calls return to `instHC`. Once all the calls return, `instHC` forwards the persistent state object to the new instance. Next, the suspended calls are reinvoked on the new instance by `instHC`. The new instance is able to reconstruct the transient state corresponding to each of the reinvoked calls using the state information stored in the corresponding transient state object, as discussed above.

We assume that only the global state of an instance is shared among different calls (threads) executing concurrently on the instance, and no local state of the methods is shared.

The above approach might lead to some problems because a call reinvoked on the new instance might find the global state to be different from the state when this call was suspended in the old instance. This is because the global state might have been changed by other calls, after this call was suspended. However, this situation might arise even when the instance replacement is not involved, as a call might take an arbitrary amount of time for executing a statement during which the global state might be changed by other calls.

The above simply means that the access to the global state needs to be carefully managed, and the safe points need to be carefully designed, when multiple calls are allowed to execute concurrently on a replaceable instance.

Multiple calls executing concurrently on an instance may also potentially lead to a deadlock situation due to arbitrary suspension of calls. A deadlock situation

[5] We are assuming here that even when a number of calls suspend their executions at different safe points within the old instance, the execution of these calls can still be resumed correctly on the new instance without compromising the consistency of the application. That is, any combination of safe points used for the suspension of executing calls is still considered a safe point for the whole instance.

can occur if synchronization / mutual exclusion locks are used by the calls. For example, one call may acquire a lock over certain object and then reach a safe point and suspend its execution without releasing the lock. Whereas, at the same time, another call may be waiting for the lock to be released and is not able to continue with its execution. In this case, the waiting call may never reach a safe point to suspend its execution, and therefore it leads to a deadlock situation. We assume that it is the responsibility of the application developer to take into account such a situation, and design the replaceable classes in a way that the deadlock situations are avoided. For example, a call may be forced to release all the locks it holds just before it is suspended.

5.2.4 Discussion

Ensuring the consistency of the application

An important requirement of the instance replacement process is that the integrity of the application must not be compromised as a result of dynamic replacement of instances. The requirement of ensuring the application's integrity can be further divided into following two requirements.

Requirement 5: The integrity of the calls executing on the old instance at the time of instance replacement must be maintained.

Requirement 6: The integrity of the ongoing interactions among instances at the time of instance replacement must be maintained.

Ensuring requirement 5 implies that the calls suspended due to dynamic replacement must be resumed on the new instance without compromising the integrity of these calls. This requirement is satisfied in the above instance replacement process by suspending the calls only at safe points, and by transferring the state of the old instance to the new instance.

Safe points can be defined as those execution points in the old instance where, if the state of the old instance is transferred to the new instance then upon transfer it will become a valid reachable state of the new instance. It is obvious that only if the state transferred is a reachable state of the new instance, will the new instance be able to resume the execution of the suspended calls correctly. Thus, we can say that

a necessary condition for ensuring the integrity of the application under the eager replacement process is that the state of the old instance should get transformed into a reachable state of the new instance upon transfer.

The instance replacement process described above provides only some general guidelines that can help in ensuring the integrity of the suspended calls. It ultimately rests on the ability of the application developer to identify safe points correctly, and provide appropriate state transfer functions for ensuring the correct and safe resumption of the suspended calls on the new instance.

Requirement 6 is satisfied by queuing any new calls made during the instance replacement process within `instHC`, as well as the calls that were suspended and returned to `instHC`, and invoking these calls on the new instance at the end of the instance replacement.

It should be noted here that the calls queued within `instHC` may need to wait indefinitely during the instance replacement process, because in many cases the time required for the suspension of the currently executing calls may not be predicted in advance. Therefore, any hard guarantees on the response time for the waiting calls cannot be provided.

Since the mutually replaceable instances (as well as the corresponding Handle instance) must conform to exactly the same interface (according to the requirement 1 in Section 5.1), the type safety of the dynamic replacement is automatically ensured.

Eager replacement vs. lazy replacement

In this section, we show that eager replacement may not be viable for some instances, leaving lazy replacement as the only option.

As discussed in the previous sub-section, a necessary condition for ensuring the integrity of the application under the eager replacement process is that the state of the old instance should get transformed into a reachable state of the new instance upon transfer.

The execution points in the old instance that ensure the above condition to be satisfied are called safe points in our description of the instance replacement process.

However, we argue that there is no guarantee that a safe point exists in an arbitrary instance to be replaced (not counting the control points just before the initial point and just after the final point of execution of a call, as these are not practically very useful).

To support our argument, we refer to the results provided by Gupta et al. [GJB96] in the context of a runtime change in software version. They define a valid change as the one in which the state of the old software version gets transformed into a reachable state of the new software version. They also show that locating the points of execution where a valid change may be guaranteed is in general undecidable, and approximate techniques based on data-flow analysis and knowledge of the application developers are required. This effectively implies that there may not exist any point of execution in the old software version that may guarantee a valid change.

This result can be directly extended to the case of dynamic replacement of instances, to support our argument that there is no guarantee that a safe point exists in an arbitrary instance to be replaced.

If an instance does not contain any safe point, then the possibility of adopting the eager replacement strategy is automatically ruled out, leaving the lazy replacement strategy as the only option.

Even with the lazy replacement, though there is no transient state to be transferred, the persistent state of the old instance still needs to be transferred to the new instance. We assume here that the persistent state of the old instance when transferred to the new instance (using the appropriate state transfer functions provided by the concerned instances) is automatically a reachable state of the new instance.

An exception to above is if the new instance is designed to recover from a state loss. In this case, the execution of the currently executing calls can be suspended abruptly (when following eager replacement) or allowed to terminate normally (when following lazy replacement), while no state needs to be transferred to the new instance. However, recovering from a state loss is likely to result in certain functional / performance penalties.

For lazy replacement, the replacement process discussed before can be suitably modified in a straightforward manner. In any case, the implementation of either eager replacement or lazy replacement is localized within a Handle class and the corresponding replaceable classes.

Another option for implementing the eager replacement process could be to use transactions in place of safe points. That is, all new calls could be started as transactions, and in case of suspension these transactions could be simply rolled back. However, such an approach would result in a loss of computation carried out be-

tween the time when the decision for suspension is taken and the last checkpoint. Moreover, it incurs additional runtime overhead for check-pointing etc. during the normal execution (in our approach, major overhead is incurred during the instance replacement process, while only a small overhead is incurred during the normal execution). Therefore, a transaction-based approach has not been preferred for our purpose.

5.3 Dynamic addition and removal of components

5.3.1 Dynamic addition of components

If an instance is dynamically added as a replacement of another instance, then the addition of this instance will take place according to the procedure for dynamic replacement of instances discussed in Section 5.2.

In this section, we discuss the dynamic addition of an instance that is not replacing any other existing instance. The process of dynamic addition of an instance varies according to the actual creator of the instance. Two possible creators of the instance are: (i) another instance (which itself may be dynamically added, either as a part of dynamic replacement or otherwise), or (ii) the CAS.

Let the instance to be dynamically added be `instA`. Below we discuss the implementation of dynamic addition of `instA` according to the above two possible creators.

Case 1: `instA` is created by another instance, say `instB`:

In this case, the specification of `instA` is not provided in the application contract. `instB` simply creates `instA` using the dynamic instance creation facilities provided by modern OOP languages.

Case 2: `instA` is created by the CAS:

The specification of `instA` is provided in the application contract within a <comp> element. The <comp> element is defined inside the <components> element. An example of a <comp> element is given below:

```
<comp name="instA" class="/class-folder/C1"/>
```

In the above example, the CAS will create an instance of the class `C1`, and name it `instA`.

The constructor of `instA` is responsible for integrating `instA` into the application, in addition to carrying out the initialization of `instA`.

Integrating `instA` into the application may involve ways to make the reference to `instA` available to other instances of the application interested in using the services provided by `instA`. This can be done, for example, by registering `instA` with an instance directory lookup service so that other instances of the application are able to discover `instA` for interactions. Alternatively, `instA` may itself locate and initiate interactions with other instances of the application.

The above implies that the instances that are dynamically added (and removed) by the CAS should provide *plug-and-play* properties [ML98].

5.3.2 Dynamic removal of components

If the instance to be dynamically removed is eventually replaced by another instance, then the removal of this instance will take place according to the procedure for dynamic replacement of instances discussed in Section 5.2.

In this section, we discuss the dynamic removal of an instance that is not replaced by another instance. As in the dynamic addition, the process of dynamic removal of an instance varies according to the actual remover of the instance. Two possible removers of the instance are: (i) another instance (which itself may be dynamically removed, either as a part of dynamic replacement or otherwise), or (ii) the CAS.

Let the instance to be dynamically removed be `instA`. Below we discuss the implementation of dynamic removal of `instA` according to the above two possible removers.

Case 1: `instA` is removed by another instance, say `instB`:

In this case, `instB` simply removes `instA` using the dynamic instance removal facilities provided by modern OOP languages.

It is possible that some other instances in addition to `instB` also hold the reference to `instA`. These instances should either be informed explicitly by `instB` about the removal of `instA`, or these should be provided with appropriate fault tolerance mechanisms to withstand the removal of `instA`. It is generally discouraged to share the reference of a dynamically removable instance.

If the instance to be removed is currently executing then, depending on the properties of the specific instance, either its execution may be immediately terminated,

or it may also be provided with safe points (similar to the ones for dynamic replacement of instances) where its execution may be terminated without compromising the consistency of the application.

Case 2: `instA` is removed by the CAS:

In response to a change in the composition of the application, if the CAS discovers that certain <`comp`> elements that existed in the specification of outgoing composition do not appear in the specification of incoming composition, then this serves as an indication for the CAS to remove the corresponding instances. The CAS already holds the references to these instances, as these instances must have been initially created by the CAS.

In order to remove `instA`, the CAS calls the `finalize` method of `instA`. The `finalize` method is responsible for ensuring that the removal of `instA` does not compromise the consistency of the application (e.g. it does not result in the problem of dangling references etc.), in addition to carrying out the normal finalization operations.

The exact semantics of the `finalize` method as well as the constructor method of an instance to be dynamically added / removed by the CAS are instance-specific. The application developer needs to ensure that these two methods are implemented appropriately so as to fulfill their individual responsibilities.

5.4 Sequential vs. atomic recomposition

So far, we have discussed addition, removal and replacement of individual instances as a part of a dynamic change in the composition of an application. In this section, we discuss the process of overall recomposition of an application, i.e. addition, removal and replacement of all the instances affected by the recomposition.

In principle, there are two approaches for recomposition of instances: sequential recomposition and atomic recomposition.

As the name suggests, in a sequential recomposition process, instances are added, removed or replaced one after the other. That is, the process of adding, removing or replacing one instance is completed before initiating the addition, removal or replacement of the next instance. This implies that in a sequential recomposition, some instances from the outgoing composition may coexist with some other instances from the incoming composition during the recomposition process.

```
Sequential_Recomposition
BEGIN
1.      FOR every instance affected by the recomposition DO
2.          Add, remove or replace the instance;
3.      ENDFOR
END Sequential_Recomposition
```

Figure 5.14: Sequential recomposition process

```
Atomic_Recomposition
BEGIN
1.      Remove all the outgoing instances;
2.      Add all the incoming instances;
END Atomic_Recomposition
```

Figure 5.15: Atomic recomposition process

Though for many applications sequential recomposition may not pose any problems, some applications may demand that instances from only one of the alternative compositions should be active at any given time.

A recomposition process which ensures that instances from only one of the compositions (either outgoing or incoming) are active at any given time is referred to as atomic recomposition. That is, in an atomic recomposition, only when all the outgoing instances have been removed, the incoming instances can be added. The outgoing instances here refer to the instances belonging to the outgoing composition but not to the incoming composition, and incoming instances refer to the instances belonging to the incoming composition but not to the outgoing composition.

The basic algorithm for sequential recomposition is presented in Figure 5.14, and that for atomic recomposition is presented in Figure 5.15.

In the above algorithm for the atomic recomposition process, the replacement of instances is carried out in two steps (recall that a replacement is in fact a combination of two steps: removal of the old instance and addition of the new instance). In the first step of the algorithm, all the old instances to be replaced are deactivated and their execution suspended, while all the new instances are prepared for activation (i.e. steps 1-4 of the instance replacement process described in Section 5.2.1). In the second step of the algorithm, all the new instances that are already prepared for

activation in the first step are simply activated (i.e. step 5 of the instance replacement process).

In case of a sequential recomposition, no additional measures are required for ensuring the consistency of the application. This is because in a sequential recomposition, only one addition, removal or replacement of an instance takes place at a time. This means that once each individual addition, removal or replacement of an instance is ensured to be consistent, the overall recomposition is automatically ensured to be consistent.

In case of an atomic recomposition, for dynamic addition and removal of instances (the ones which are not part of instance replacement), no additional measures need to be taken by the recomposition process for ensuring the consistency of the application. This is because multiple additions and removals of instances are actually independent of each other. Once the measures for ensuring the consistency of an application in case of a single addition or removal of an instance are provided, adding or removing multiple instances does not impose any extra consistency concerns.

However, the atomic recomposition process adds a new complexity for dynamic replacement of instances. This additional complexity is caused because the instance replacement is carried out in two steps in the atomic recomposition process, forcing interdependence among multiple instance replacements. Thus replacing multiple instances simultaneously can potentially lead to a deadlock situation.

Before discussing further about the potential deadlock situation, let us see how the replacement process actually works in case of an atomic recomposition.

First, the CAS instructs all the concerned Handle instances to remove old instances, and prepare new instances for activation. That is, the Handle instances carry out steps 1-4 of the instance replacement process described in Section 5.2.1. Once a Handle instance completes the above steps, it sends a `ready` signal back to the CAS indicating that it is now ready to activate the new instance. The CAS waits until it receives the `ready` signals from all the concerned Handle instances. Once the CAS receives the `ready` signals from all the concerned Handle instances, it instructs the Handle instances to activate the new instances, i.e. to carry out step 5 of the instance replacement process. This way, it is ensured that instances from only one of the compositions are active at any given time.

Now to understand a potential deadlock situation, consider two instances `instA1` and `instB1` to be replaced by `instA2` and `instB2` respectively. A call issued from

`instA1` may be executing within `instB1` at the time when the CAS sends an instruction for instance replacement to the Handle instances corresponding to `instA1` and `instB1`. Once the Handle instance corresponding to `instB1` sets a signal for the suspension of `instB1`, all the calls currently executing on `instB1` will be suspended at their respective following safe points, and made to wait within the Handle instance. The call issued from `instA1` to `instB1` may also be among the calls which are suspended and made to wait within the Handle instance. However, at the same time, `instA1` will be waiting for the above call to return before its execution can be suspended.

The above situation implies that the Handle instance corresponding to `instA1` is not able to send a `ready` signal to the CAS until the call issued from `instA1` to `instB1` returns. Whereas, on the other hand, the above call cannot return until all the concerned Handle instances (including the one corresponding to `instA1`) send the `ready` signals to the CAS, and the call is reinvoked on the new instance (i.e. `instB2`). Therefore, this situation is a deadlock situation.

One possible solution to avoid the above deadlock situation will be for the CAS to know the dependencies between instances, and instruct the suspension of instances in a particular order. For instance, in the above example, if the CAS knew that `instA1` may potentially issue a call to `instB1`, then the CAS may first instruct the suspension of `instA1`, and only when `instA1` is suspended would the CAS instruct the suspension of `instB1`. That is, the instances may be suspended in the downstream order of their dependency.

However, the above solution has two drawbacks. First, it requires the information about the dependencies between instances to be available to the CAS, which obviously imposes an additional overhead. And second, this solution cannot work when there are any cycles in the dependency graph of instances.

A simpler and lightweight solution, followed in our approach, is to give special treatment to the calls issued from the instances that can be dynamically removed, so that these calls are not suspended but are allowed to complete their normal execution and return to their callers.

For the above solution to work, the calls issued from the instances that can be dynamically removed need to be identifiable as special calls. We follow a simple implementation of the above solution in Java, which is to append a special identifier called `nosuspend` to the names of the threads executing these calls. At a safe point,

the execution of a call is suspended only if the name of the thread executing the call does not have the above identifier appended to it. Otherwise, the execution of the call is not suspended, and instead the call is allowed to complete its normal execution and return to its caller.

The above solution should work for most practical applications, but does not guarantee 100% immunity from deadlock situations. For example, in case of a never-ending loop, a call with the above special identifier appended to it may never be suspended, even if there is a safe point within the code of the loop. For such special cases, the application developer needs to make sure that appropriate provisions for avoiding deadlock situations are provided within the application logic. However, for most practical applications, the above solution would be sufficient for avoiding a deadlock situation when the atomic recomposition process is followed.

5.5 Discussion

In this chapter, we have described the CASA approach for dynamic recomposition of components. The approach discussed in this chapter is generic, and can be used for any modern object-oriented programming language (even though we have used Java as a target language for explaining some of the concepts in this chapter). In order to use our approach, an object-oriented programming language must provide mechanisms for dynamically creating and deleting objects (all modern object-oriented programming languages satisfy this basic requirement). The Handle instances used for dynamic replacement of instances in our approach can be implemented in any language. The interactions between the Handle instances and the CAS currently takes place using Java RMI. However, any remote procedure call (RPC) mechanism can be used for such interactions instead.

The main decision to develop our own approach for dynamic recomposition of components, instead of using any of the existing approaches, is the flexibility required for dynamic replacement in CASA. A number of approaches have been proposed for dynamic recomposition of components in the last few years. However, these approaches are targeted to software evolution rather than software adaptation, with the result that these approaches can adapt only those components that are not running (i.e. not serving any client requests) at the time of adaptation. If a component is running at the time of adaptation, then either the component is allowed to finish

its current execution, or its current execution is discarded.

A discussion of the related approaches for dynamic recomposition of components is provided in Chapter 7. Two of the most popular technologies for replacing components dynamically are Java HotSwap [Dmi01] and OSGi [OSG07]. The Java HotSwap technology allows replacing classes of a running Java program by redefining the methods of these classes. However, it does not allow currently active methods of a class to be replaced. OSGi provides a framework for managing the life cycle of Java components. It allows Java components to be dynamically added and removed, and bindings between the components to be dynamically changed. However, OSGi also does not provide support for replacing currently running components. On the other hand, for simple addition or removal of components in CASA (i.e. without replacement), technologies like OSGi can be quite useful.

The problem of replacing currently running components is much more complex than the problem of replacing non-running components. A significant advantage offered by some of the existing approaches for dynamic recomposition is that the component replacement can be done without application participation. In our approach, some amount of application participation is required in the form of implementing Handle instances. However, since the Handle instances offer standard functionality, templates for these instances can be provided by the programming environment, which can be customized by the application developer for the specific application. Moreover, our approach requires a replaceable instance to explicitly define safe points where its execution can be terminated safely. We envision that appropriate tools for identifying such safe points will be available to the application developer in future. Similarly, our approach requires mutually-replaceable instances to agree to a common representation of the state to be exchanged. This is a reasonable requirement for any scenario that involves information exchange between two distinct computational entities.

In spite of the mechanisms for suspending the calls only at safe points, transferring the instance state, and maintaining the integrity of ongoing interactions provided in CASA, the ultimate onus of resuming the execution of suspended calls correctly is on the new instance. This implies that even if the state of the old instance is transferred to the new instance, and the calls suspended at the safe points in the old instance are reinvoked on the new instance, the new instance may need to implement certain additional recovery mechanisms to be able to resume the execution of the

suspended calls correctly. Such recovery mechanisms are instance and application specific, and cannot be generalized.

In our design of the components recomposition approach, we have not placed any special requirements on the programming language. However, a future version of this approach can benefit from the support provided at the programming language level. In particular, a programming language can provide special constructs for defining safe points in the instance code. Similarly, the programing language can support state transfer between instances. However, we believe that the application developer will still need to define the safe points explicitly using the constructs provided by the programming language, as it seems highly unlikely at this stage that a programming language could identify the safe points automatically (e.g. by simply inspecting the stack formats of the methods of old and new instances). Similarly, we believe that the application developer will need to identify the parameters of the state to be transferred between mutually replaceable instances, though the actual state transfer can be greatly facilitated by the programming language (by transferring the stack frames and initializing the new instance). The application developer can be assisted in performing these activities by appropriate tools. We believe that developing an adaptive application will eventually be a matter of partnership between the programming language features, appropriate analysis and development tools, and the knowledge of the application developer. All these three are important and indispensable ingredients for successfully developing an effective adaptive application.

Chapter 6

Prototype Implementation and Performance Evaluation

We have implemented a prototype of the CASA Runtime System (CRS), and a demo system for demonstrating the dynamic adaptation capabilities of the CASA framework. The prototype and the demo system are implemented in Java. The demo system implements an *Emergency Coordination System (EmCoS)* consisting of two applications: *Monitoring* and *Support*. *Monitoring* is responsible for monitoring a disaster-affected area, and sending its observations to *Support*. *Support*, in turn, is responsible for coordinating the rescue operations based on the information received from *Monitoring*.

Further details on the implementation are discussed in Section 6.1, and results of the performance evaluation are presented in Section 6.2.

The main purpose of implementing the prototype was to study the feasibility, and evaluate the performance of our components adaptation approach presented in the previous chapter. In addition, proof-of-concept experiments were carried out for integration of CASA with PROSE (for dynamic adaptation of aspects), for integration of CASA with a dummy adaptive middleware system[1] (for dynamic adaptation

[1]We decided to integrate CASA with a dummy adaptive middleware system instead of any of the potential real adaptive middleware systems for our proof-of-concept experiment, as many of the potential middleware systems have restrictions with respect to platform (e.g. the Odyssey implementation is available currently only for NetBSD OS) and many others required significant instrumentation for developing and integrating a manager for the middleware system. Therefore, integration with a real adaptive middleware system was not considered worth the effort for a proof-

of lower-level services), and for dynamic adaptation of application attributes through callbacks.

6.1 Implementation

In *EmCoS*, the communication bandwidth available between *Monitoring* and *Support* is considered highly unreliable because of the nature of the operation (*Monitoring* uses a wireless link for transferring data to *Support*; *Monitoring* may need to move frequently while surveying the disaster-affected area). Moreover, local resources available to *Monitoring* and *Support* may also change at runtime because of resource contentions with other applications running on the respective nodes (e.g. the node hosting *Monitoring* may need to start other applications for interacting with local rescue teams in the area).

In view of the runtime changes in resource availability, in particular the bandwidth availability, *Monitoring* provides different alternative configurations suited for different resource conditions. These configurations vary in the richness of information about the disaster-affected area sent by *Monitoring* to *Support*.

Monitoring provides two types of information to *Support*: damage information and rescue status. Different levels of information richness for damage information are: high-resolution images, low-resolution images, detailed textual description and brief textual description. Different levels of information richness for rescue status are: detailed rescue information and brief rescue information.

Depending on the resources currently available, CASA activates the appropriate configuration of *Monitoring*. Service negotiations between *Monitoring* and *Support* take place before any major change in the configuration of *Monitoring*, as *Support* may need to reconfigure itself in response to a reconfiguration of *Monitoring* (e.g. service negotiations are required for a change from images to text, but not for a change from high-resolution images to low-resolution images). Runtime changes in the resource availability are artificially simulated in the prototype implementation.

The alternative configurations of *Monitoring* and *Support* are defined in the respective application contracts. In this prototype, a configuration defines the constituent components and the resource requirements. The CASA Runtime System is responsible for monitoring the current availability of resources (which is simulated

of-concept experiment, given the time and resources at hand.

in the prototype), and activate the most appropriate configuration for the current resource availability. A change in configuration is carried out by replacing the corresponding components in the prototype. A runtime replacement of the information sending components of *Monitoring* involves state transfer, so that the new component does not resend the information that was already sent by the previous component. The replacement of components is carried out using the dynamic replacement process described in Chapter 5. This means that the concerned components need to provide appropriate support for their dynamic replacement, e.g. by defining the safe execution points where the components can be replaced, and the procedures for importing and exporting the state.

In future, new adaptable components can be added to either of these applications, and thereby the adaptation capability of the *EmCoS* system can be evolved. A new component that is designed to replace an existing component must conform to the same component interface (a requirement that was discussed in Section 5.1). For example, *Monitoring* can provide a new component that sends high-resolution images annotated with the locations of critical infrastructure in the area, when sending the damage information to *Support*. This component might have even more resource requirements than the component that sends normal high-resolution images. Similarly, a new component of *Monitoring* might be able to send the exact locations of the rescue team members in the area when sending the rescue status to *Support*. These new components must satisfy all the requirements for dynamic replacement of components discussed in Chapter 5. The executable codes of these components need to be made accessible to the CASA Runtime System (the locations of these components are specified at the appropriate places in the application contract), in order to evolve the adaptation capability of the *EmCoS* system accordingly.

Figure 6.1 depicts the GUI of *Support* when *Monitoring* is sending high-resolution images for the damage information and detailed rescue information for the rescue status.

Figure 6.2 depicts the GUI of CASA running on the node hosting the *Monitoring* application (node A). The upper bar for each resource shows the total amount currently available, and the lower bar shows the total amount allocated for all the running applications. A slider below the bars can be used for artificially changing the total availability of the resource. The priority of an application can also be changed by the user by entering a new value in the priority field. Figure 6.2 shows

that *Monitoring* is the only application running on node A.

As the amount of resources that can be allocated to *Monitoring* drops (either due to other applications contending for the limited resources, or the total availability of the resources dropping due to external factors), CASA carries out reconfiguration of *Monitoring*. Reconfiguration here involves changing the component composition of *Monitoring*. A change in configuration of *Monitoring* may force a corresponding change in configuration of *Support*.

Figure 6.3 shows the GUI of CASA on node A when a new application (Dummy-App1) starts executing on node A and is allocated resources, thereby reducing the amount of resources allocated to *Monitoring*. This implies that *Monitoring* is reconfigured to send only low-resolution images for the damage information and brief rescue information for the rescue status to *Support*, as depicted in the GUI of *Support* in Figure 6.4.

As we know, an application contract is specified in XML, and it can be modified at runtime. CASA provides an API for modifying application contracts. For applications serving humans, a user-friendly GUI should be provided to allow the user to modify the application contract according to her needs and preferences. A sample GUI for application contacts has been developed in the context of the CASA prototype development.

The application contract of *Monitoring* provides a user-friendly GUI for customizing the adaptation policy (see Figure 6.5). Figure 6.5 shows the default adaptation policy. A user can change the order of alternative configurations under damage-info and rescue-status using the "up" and "down" buttons. An alternative configuration may be removed using the "remove" button. A removed configuration appears in the Invalid zones table, and can be added later again using the "add" button. Any changes carried out in the GUI immediately reflect a change in the application contract in the background.

Figure 6.6 shows a situation where the user has disabled the rescue-status service, and removed color images from the damage-info service. That is, the user is interested in textual descriptions (preferably detailed, otherwise brief) of the damage information only. The user in this case could have been a software application, (e.g. *Support*) rather than a human user. When the user is a software application, the GUI of the application contract is replaced with an API for revealing and modifying the application contract. The access control for using the above API needs to be

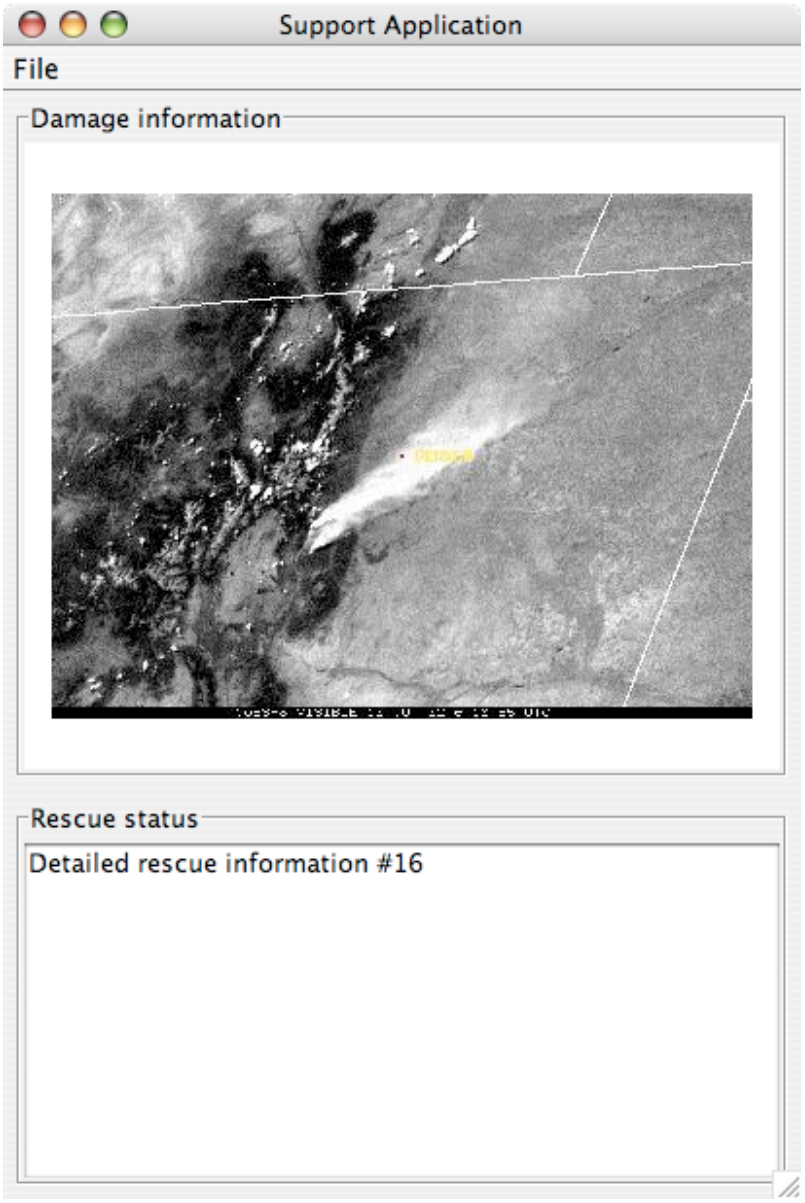


Figure 6.1: *Support* application

Figure 6.2: CASA on the node hosting *Monitoring* application

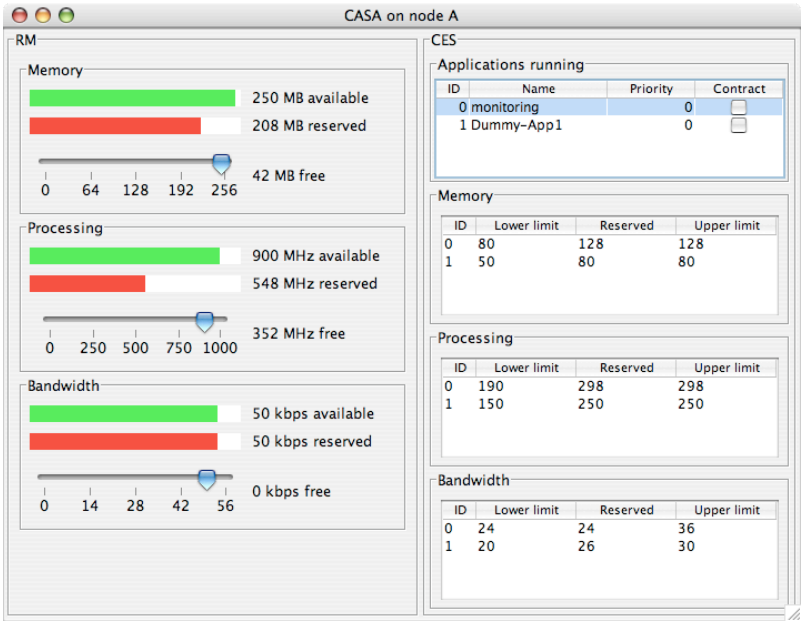


Figure 6.3: CASA on node A, hosting *Monitoring* and a dummy application

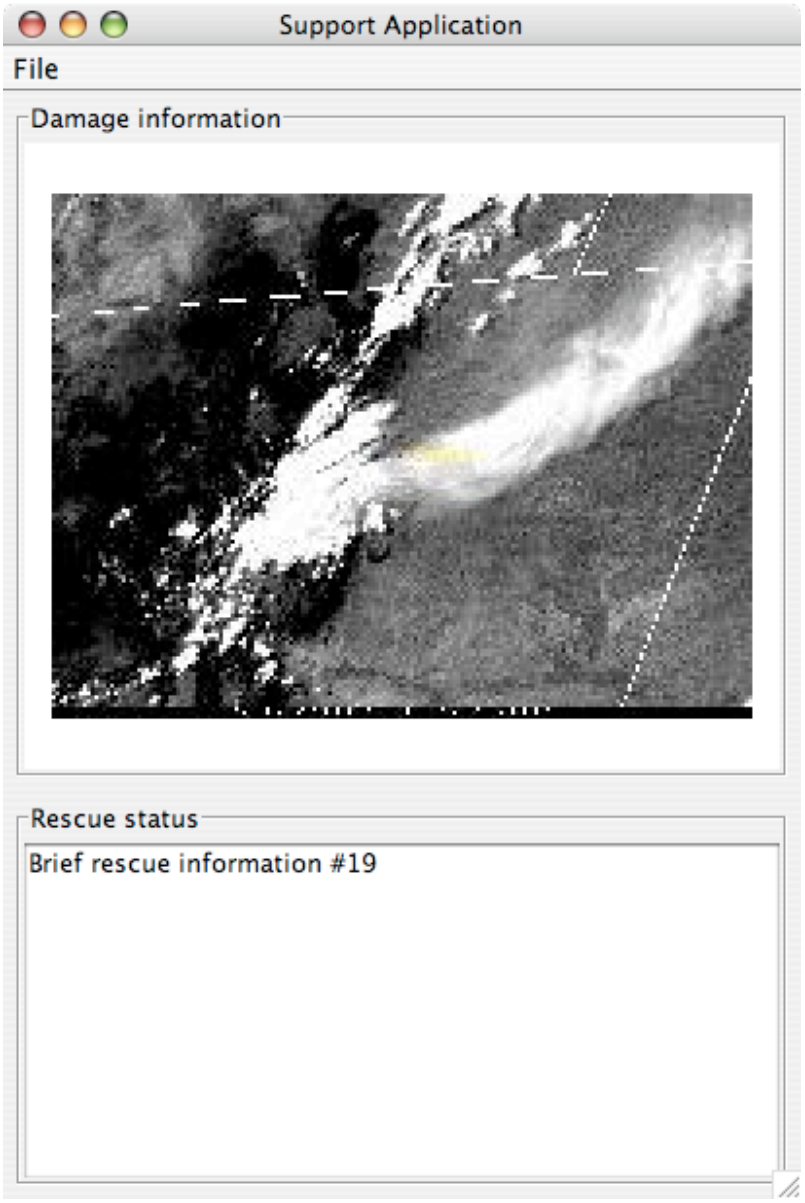


Figure 6.4: *Support* application after reconfiguration

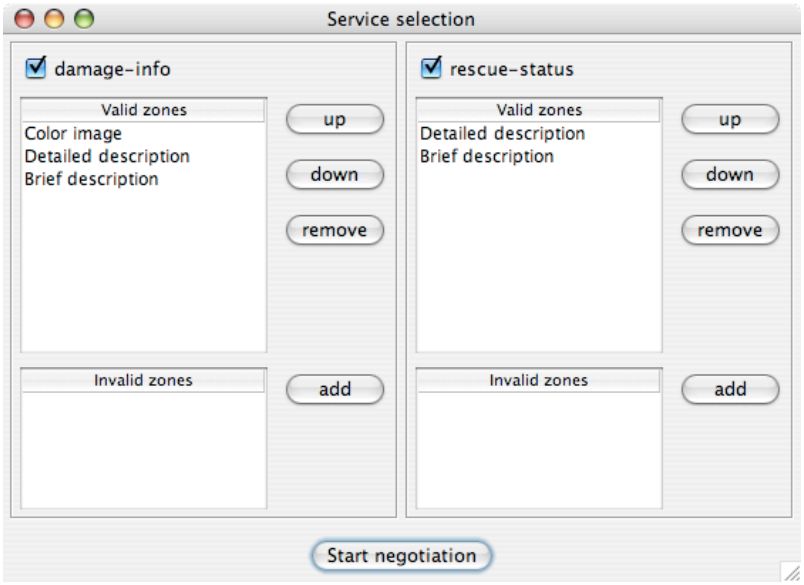


Figure 6.5: GUI of the application contract

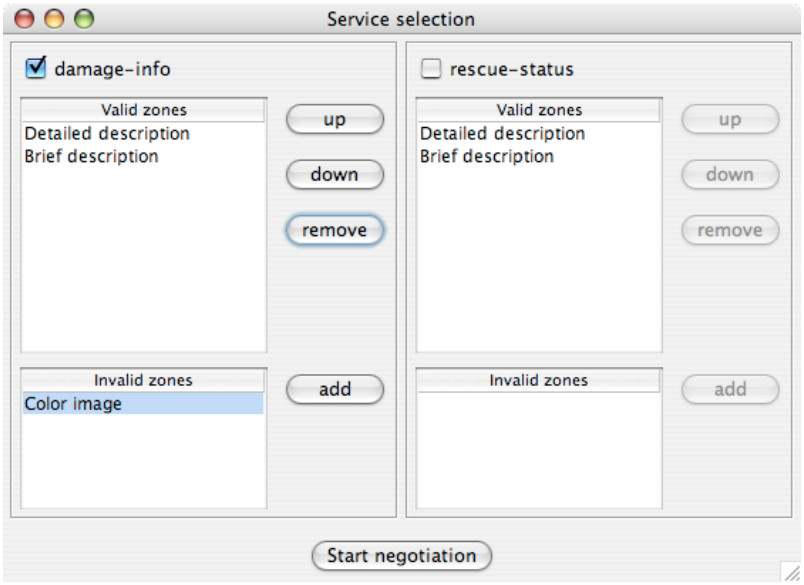


Figure 6.6: Modified application contract

applied to prevent any unauthorized modification of the application contract.

The effect of the above change in the application contract is shown in the GUI of *Support* in Figure 6.7.

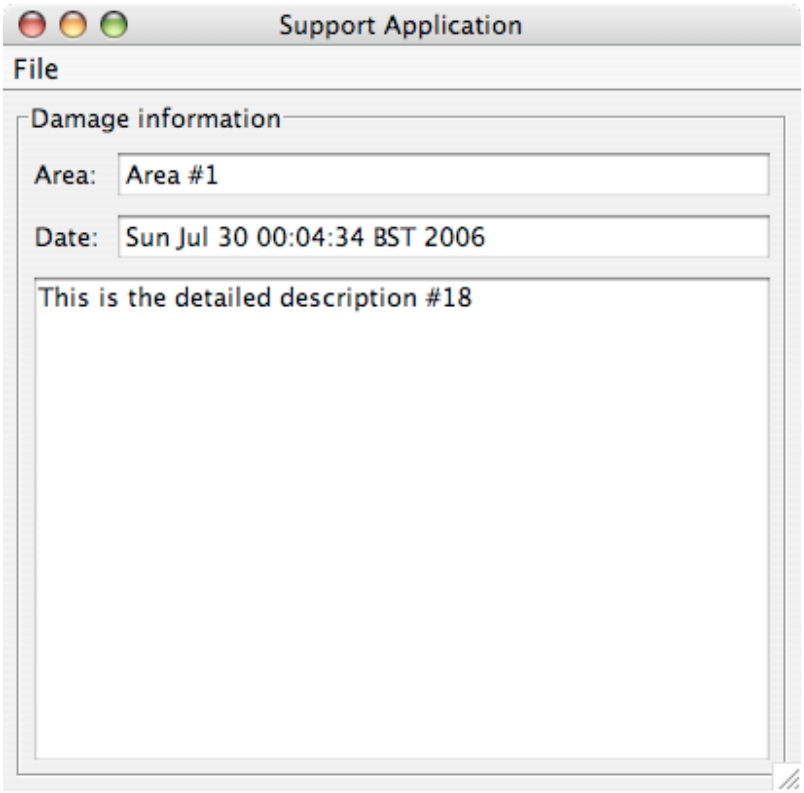


Figure 6.7: *Support* application after modified application contract

6.2 Performance evaluation

We have carried out performance evaluation tests for evaluating the overhead incurred by our components adaptation approach described in Chapter 5. The performance evaluation of the aspects adaptation approach followed by PROSE can be found in [NA05], and that of the lower-level services adaptation approach followed by Odyssey can be found in [NSN+97].

The performance evaluation of our components adaptation approach was originally carried out by a Master's student as a part of an internship project. Detailed

results of the performance evaluation can be found in the internship report [Gyg04]. Below we present some of the indicative results.

Test Environment

All tests were carried out on two platforms: Macintosh Platform (CPU: PowerPC G4 450 MHz, RAM: 640 MB, OS: MacOS X 10.3.5, Java Version: 1.4.2) and Linux Platform (CPU: AMD Athlon XP 1900+, RAM: 1024 MB, OS: SuSE Linux 9.1, Java Version: 1.4.2).

The profiling tool used for measurements was JProfiler 3.1.2 from EJ Technologies [JPr04]. With the JVM default heap size of 2 MB, the Garbage Collector runs could distort the results arbitrarily. Therefore, the heap size was set to 30 MB to prevent any Garbage Collector runs.

Test Results

Below we present the results obtained for various performance tests on the Macintosh platform (results for the Linux platform showed a similar pattern).

Test 1: Time required for an application adaptation with respect to the number of applications running on the same node

The purpose of this test is to study the influence of the total number of applications running on the same node as the application being adapted. The application adaptation here involves replacing only a single component. The component to be replaced does not have any state and does not process any data. This ensures that the component replacement is not delayed, and thus does not affect the results arbitrarily for different test runs. Tests were repeated 100 times to get a better average.

Results: The results of this test are presented in Table 6.1, and are depicted graphically in Figure 6.8. Total time here is the time from detecting a change in resource availability until completion of the component replacement. Slowdown factor in Table 6.1 indicates how many times the application adaptation is slowed down when n applications are running on the same node, as compared to only one running application. The increase in time for application adaptation with respect to the number

of running applications is due to the increase in time required by the adaptation algorithm for deciding which application to adapt.

Table 6.1: Time per application adaptation with respect to number of applications

Number of applications	Total time per adaptation (ms)	Slowdown factor
1	38.33	
2	39.29	1.03
4	44.38	1.16
7	61.97	1.62
14	117.32	3.06
20	176.16	4.6

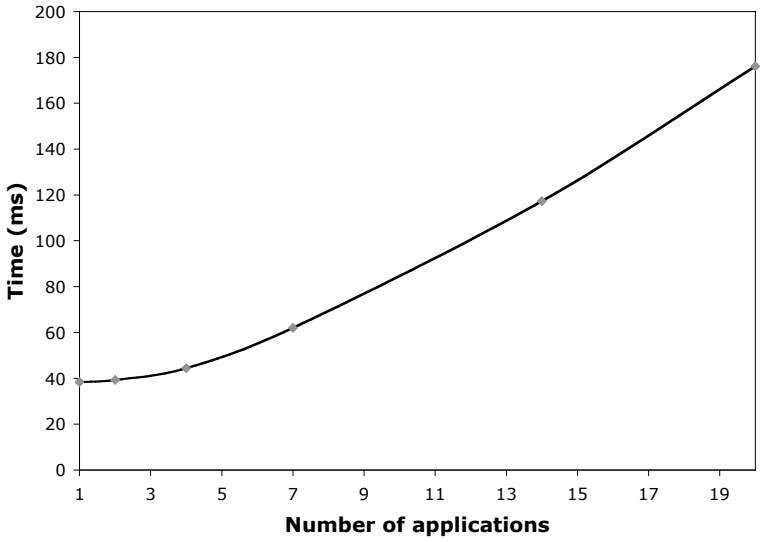


Figure 6.8: Time per application adaptation with respect to number of applications

Test 2: Time required for a change in component composition with respect to number of components to be replaced

The purpose of this test is to study the influence of the number of components to be replaced on the time required for recomposition. The components to be replaced

do not have any state and do not process any data. This ensures that a component replacement is not delayed, and thus does not affect the results arbitrarily for different test runs. Every component has exactly one Handle instantiated in the application. Tests were repeated 100 times to get a better average.

Results: The results of this test are presented in Table 6.2, and are depicted graphically in Figure 6.9. Time is measured from the moment when the CES invokes the CAS to carry out recomposition until the recomposition process is over. Slowdown factor in Table 6.2 indicates the factor by which the recomposition is slowed down when replacing n components instead of only one component. The slight reduction in the time required per component replacement with the increase in number of components to be replaced (as observed by the slowdown factor being slightly less than the number of components to be replaced) is mainly due to certain common operations performed by the CAS.

Table 6.2: Time for components replacement with respect to number of components

Number of components	Total time for replacements (ms)	Slowdown factor
1	6.93	
2	13.22	1.91
4	25.96	3.75
7	41.56	6
14	79.57	11.48
20	107.26	15.48

Test 3: Time required for a change in component composition with respect to number of Handle instances corresponding to components to be replaced

This test is similar to the test above, except that here we study the influence of number of Handles instead of number of components. In this test, there was only one component class to be replaced, and only the number of Handles instantiated in the application for the above component class were varied for measurements. Tests were repeated 100 times to get a better average.

Results: As expected, the time for recomposition is independent of whether every

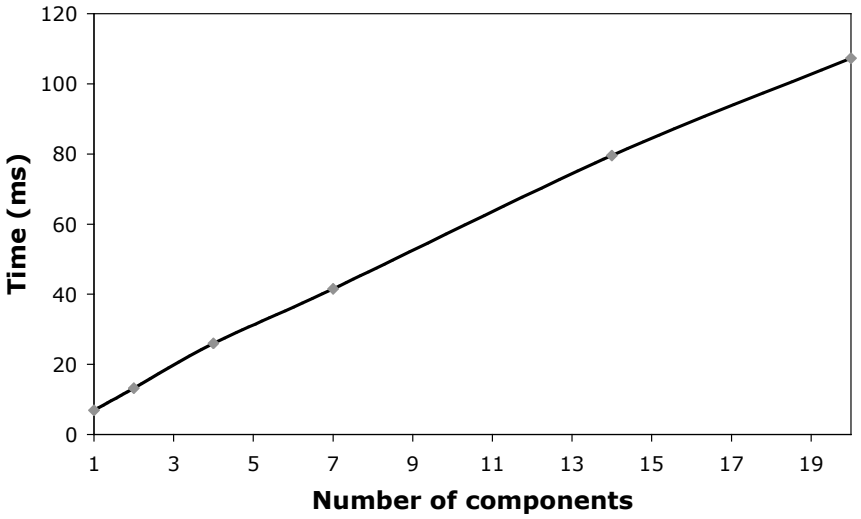


Figure 6.9: Time for components replacement with respect to number of components

Handle instance in the application belongs to a different set of alternative classes, or all Handle instances belong to the same set. Therefore the results obtained here, as presented in Table 6.3 and Figure 6.10, are very similar to those for the previous test.

Table 6.3: Time for components replacement with respect to number of Handles

Number of Handles	Total time for replacements (ms)	Slowdown factor
1	6.81	
2	13.46	1.98
4	26.18	3.84
7	46.42	6.82
14	79.61	11.69
20	108.03	15.86

Test 4: Time required for a component replacement with respect to number of safe points in the component code

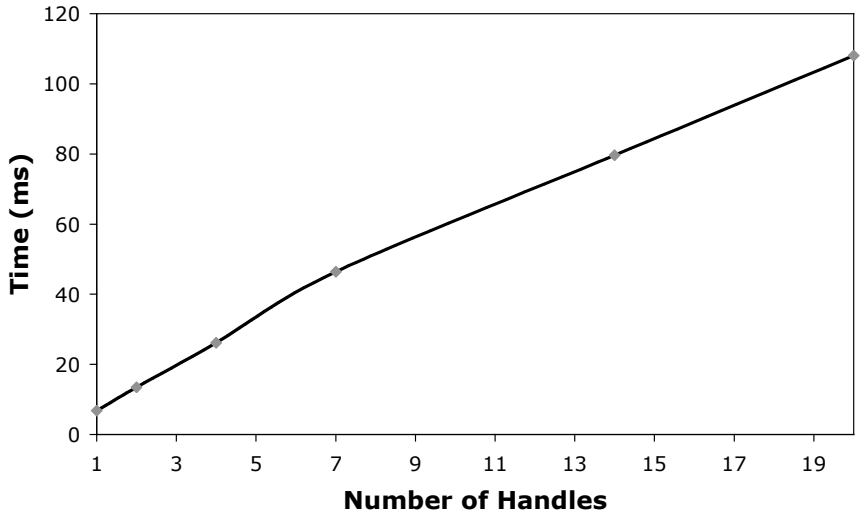


Figure 6.10: Time for components replacement with respect to number of Handles

The purpose of this test is to study the influence of number of safe points on the speed of component replacement. In this test, a component to be replaced implements a single method, which runs for 10 seconds. Tests were repeated 200 times to get a better average.

Results: The measured average times for a component replacement for different number of safe points matched quite well with their theoretically computed values, as shown in Table 6.4 and Figure 6.11. As expected, the frequency of safe points has a positive effect on the speed of component replacement.

Table 6.4: Time for a component replacement with respect to number of safe points

Number of safe points	Measured average (ms)	Theoretical average
0	5138.18	5007
1	2405.59	2507
2	1695.9	1674
4	926.99	1007

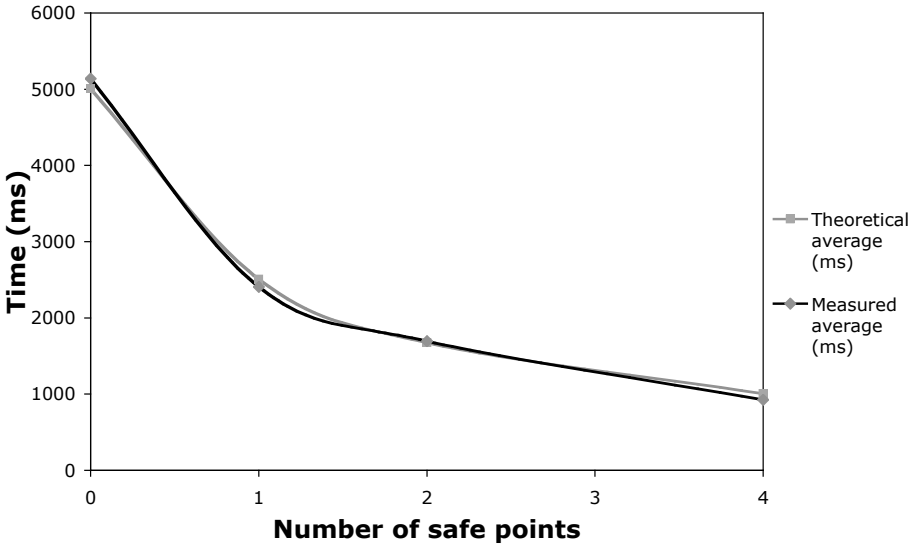


Figure 6.11: Time for a component replacement with respect to number of safe points

Test 5: Overhead due to safe points during normal operation

This test studies the overhead due to presence of safe points in the application code during normal operation (i.e. when no recomposition is involved). Tests were carried out on a component implementing an empty method containing a number of safe points. Total response time for the method was measured in different cases in order to calculate the overhead due to safe points. Tests were repeated 10,000 times for each variant (containing a different number of safe points) to get a better average.

Results: The results are presented in Table 6.5 and Figure 6.12. Please note that the time is measured here in microseconds, unlike previous tests where time is measured in milliseconds. The overhead due to each safe point during normal operation was found to be very low. As expected, this overhead is independent of the total number of safe points defined in the method.

Looking at the results from this and previous test, we observe that while the overhead due to safe points during normal operation is quite low, the presence of safe points can significantly speed up a component replacement during recomposition. Therefore, it might be worthwhile to define frequent safe points if possible, especially

after any time-expensive computations.

Table 6.5: Overhead due to safe points during normal operation

Number of safe points	Total time for a call (μs)	Overhead per safe point (μs)
0	2.74	
1	7.7	4.96
2	13.68	5.47
3	18.06	5.11
4	23.22	5.12
5	28.62	5.18
10	54.66	5.19

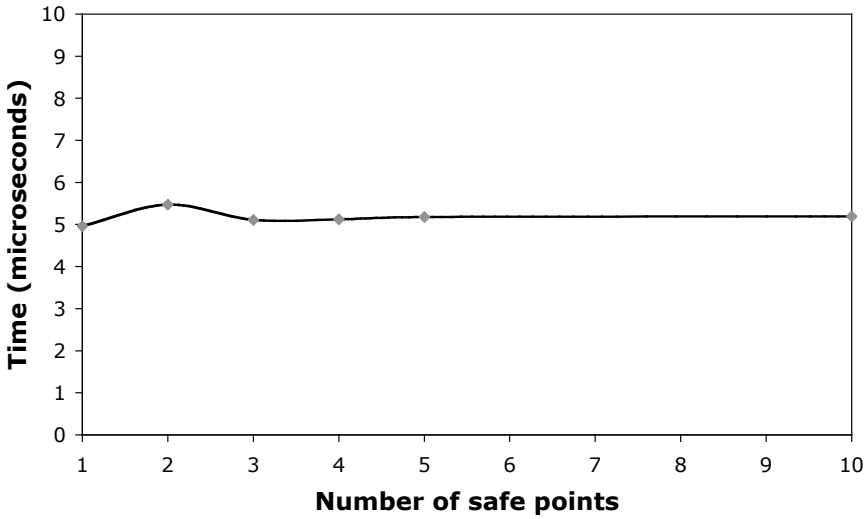


Figure 6.12: Overhead per safe point

Test 6: Overhead due to usage of Handles during normal operation

This test studies the overhead due to indirection of method calls through Handles, instead of direct calls to the target component, during normal operation. Tests were carried out on a component implementing an empty method that returns im-

mediately. Calls were made to the method directly and through a Handle. Tests were repeated 10,000 times to get a better average.

Results: Table 6.6 shows the overhead due to usage of a Handle, as compared to direct calls. Time is measured here in microseconds. It should be noted here that the overhead due to using a Handle, as mentioned in Table 6.6, is independent of the actual execution time of the method. Therefore, even though in Table 6.6 making a call through a Handle is 11 times slower than a direct call, if a direct call would take 0.1 second (including the processing time by the method), a call through a Handle would take 0.100027 second, i.e. an overhead of 0.027%.

Table 6.6: Overhead due to Handles during normal operation

	Total time for a call (μs)	Overhead due to Handle (μs)
Direct	2.74	
Via Handle	30.16	27.42

Test 7: Time for initializing a Handle

Time for initializing a Handle includes registering the Handle with the CAS, getting information about the currently active class, creating an instance of this class and setting the instance as the active instance. For the test, a number of Handles of the same type were initialized. Tests were repeated 100 times for getting a better average.

Results: The results for initializing Handles are shown in Table 6.7 and Figure 6.13. As seen from the results, the first initialization of a Handle takes significantly higher time than subsequent initializations. This is because RMI takes longer time for creating a connection and enabling communication between a Handle and the CAS for the first time, as compared to subsequent communications. In any case, the communication time between a Handle and the CAS dominates the total time for initializing a Handle.

Test 8: Overhead due to state transfer with respect to complexity of state

Table 6.7: Time for initializing Handles

Number of Handles	Total time (ms)	Time per Handle (ms)
1	61.4	61.4
4	117.6	18.73
7	182.4	21.6
14	331.6	21.31
20	466.8	22.53
50	946.6	15.99

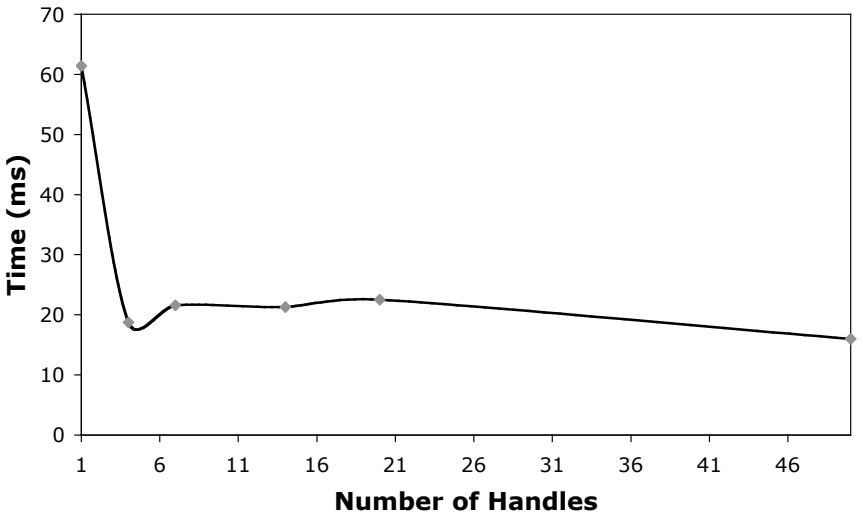


Figure 6.13: Time per Handle initialization

This test studies the influence of complexity of state parameters on the time for state transfer. Complexity of a state to be transferred is measured in terms of number of state parameters and types of these parameters (integer or object). These tests were repeated 10,000 times for getting a better average.

Results: Time for transferring the persistent state is measured, and results are presented in Table 6.8 and Figure 6.14. As seen from the results, there is not much difference in results whether primitives (integers) or objects are transferred. This

implies that the time for state transfer depends mainly on the number of state parameters being transferred, while the types of these parameters do not influence the results. As expected, time for transferring every additional state parameter (beyond the first parameter) is almost constant.

Table 6.8: Overhead due to state transfer

	Integers		**Objects**	
Variables	Total time for state transfer (μs)	Addl. time per variable (μs)	Total time for state transfer (μs)	Addl. time per variable (μs)
0	2.6		2.6	
1	32.1	29.5	32.3	29.6
2	38.7	6.6		
3	45.3	6.6	44.9	6.3
5	57.9	6.3		
10	90.1	6.4	90.0	6.5

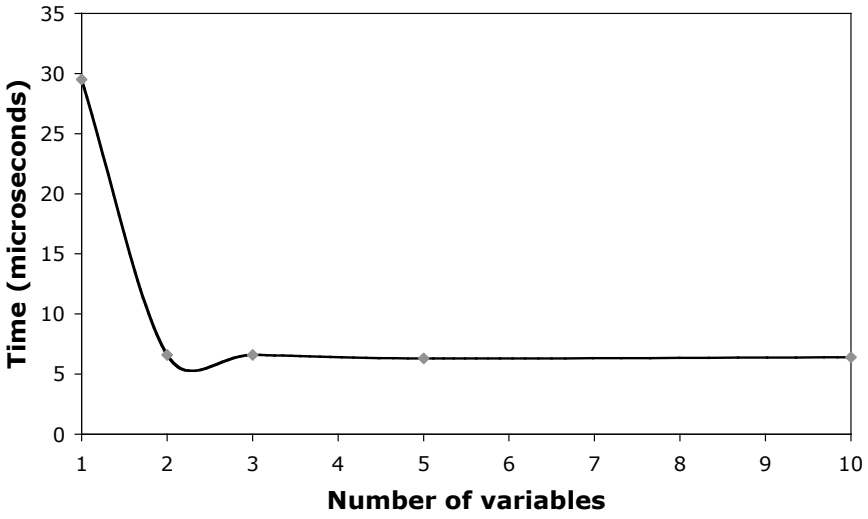


Figure 6.14: Additional overhead per variable

Test 9: Time for registering an application with CASA

Time for registering an application includes the time for the RMI lookup of CASA and the time for actual registration (i.e. passing the application contract to CASA etc.). This test was repeated 20 times to get a better average.

Results: The time required for registering an application with CASA is given in Table 6.9. The RMI lookup of CASA takes more time than the actual registration. Since the registration needs to be done only once in the beginning for an application, it is not likely to be time critical.

Table 6.9: Time for registering an application with CASA

	Time (ms)
RMI Lookup	665.20
Registration	395.20
TOTAL	1060.40

Discussion: The above results show that the overhead incurred by the components recomposition approach followed in CASA is reasonably small, and tolerable for most practical applications. We believe that the overhead incurred by an adaptation approach during normal operation of an application is more important than the overhead incurred during adaptation. This is because, in general, the number of times an application needs to be adapted during its lifetime is usually very low, when compared to the total lifetime of the application itself (i.e. the time the application is up and running). Moreover, the adaptation of an application is meant to provide certain value to the application (e.g. to allow it to continue its execution when the execution environment conditions get worse, or to improve its functionality or performance when the execution environment conditions get better). Therefore, the overhead incurred during adaptation is not a *real* overhead, while that during normal operation is a *real* overhead. However, having said that, the overhead incurred during adaptation should be kept as low as possible, as some applications may have strict availability requirements and may not be able to tolerate any significant disruptions in their execution due to adaptation.

In this respect, the overhead incurred by the components adaptation approach followed in CASA is found to be very small during normal operation, and reasonably

small during adaptation. In particular, the overhead due to an additional level of indirection of method calls, as induced by the use of Handles, during the normal application operation is quite small (less than 30 μs). An important observation obtained by the above results is that the overhead due to safe points in the component code is very small during the normal operation (around 5 μs per safe point). On the other hand, as expected, increasing the number of safe points in the component code has a considerably positive effect on the speed of component replacement. This implies that there is an incentive in defining frequent safe points in a component – in terms of increased speed of component replacement during adaptation, at the cost of very low overhead due to safe points during normal operation. The safe points should especially be defined after any time-expensive computations. The overhead due to state transfer during adaptation is also very small (in the order of a few microseconds), and the overhead due to the overall adaptation process is reasonably small (in the order of a few milliseconds). However, it should be noted that the adaptation process can take unpredictably long time depending on the distance between the current execution point and the next safe point within a component. A carefully designed software component is expected to keep this distance as short as possible on average, and not allow an indefinite waiting time before the execution reaches a safe point. Naturally, the speed with which an application is able to respond to the changes in its execution environment by adapting its behavior also depends on the speed with which these changes are discovered by the underlying resource and context monitoring services.

Chapter 7

Related Work

In this chapter, we give an overview of related work on dynamic adaptation of applications. Related work on monitoring the execution environment, i.e. contextual information and resources, has already been discussed in Chapter 3.

In the last few years, a vast number of approaches have been proposed for dynamic adaptation of applications. These approaches have tried to handle the issue of dynamic adaptation from different perspectives. One way of classifying these approaches would be their target of adaptation, i.e. the part of an application configuration that is adapted by these approaches. According to this classification, we classify the approaches as the ones adapting (i) application components, (ii) aspects, and (iii) lower-level services. To our knowledge, CASA is the first approach that tries to adapt all three above parts of an application configuration, in order to provide a comprehensive adaptation solution.

Another way of classification would be the application level where support for adaptation is provided. According to this classification, we classify the approaches as (i) middleware level approaches, (ii) application code level approaches, and (iii) software architecture level approaches. This kind of classification does not have strict boundaries, and there are significant overlaps. For example, software architecture level approaches ultimately perform adaptation actions at either the application code level or middleware level. However, since the adaptation logic and decisions are based at the architecture level, we classify these approaches as architecture level approaches. Similarly, application code level approaches usually have some middleware level components, but eventually carry out adaptation of the application code. Therefore, we classify these approaches as application code level approaches. On the

other hand, middleware level approaches may interact with the application code, but if the adaptation actions are performed at the middleware level (i.e. outside the application code), then we classify these approaches as middleware level approaches.

Below we present some of the important approaches according to the second classification. The field of dynamic adaptation of applications is very broad, and it is not possible to list all the related approaches in this chapter. Therefore, we restrict our discussion here to only some prominent approaches, or those approaches that are closely related to our work on the CASA framework. For every approach presented, we also specify the target of adaptation for that approach i.e. components, aspects or lower-level services.

Many of the middleware level adaptation approaches support callbacks to applications in case of changes in the execution environment. These callbacks can be used for adapting application attributes. Adapting application attributes through callbacks is a trivial activity, and we do not explicitly list the approaches supporting such callbacks.

Similarly, we do not list the approaches where the adaptation logic and mechanism are hard coded within an application, because such approaches are quite restrictive in practice. In particular, these approaches result in (1) the inability to change the adaptation strategy or add new adaptive behavior at runtime, and (2) increased complexity due to hard coding of the adaptation concerns within an application.

7.1 Middleware level adaptation

In some respects, middleware provides a natural place for dynamically adapting applications. This is mainly because of two reasons. First, since middleware is placed between an application and its computing environment, it is in a favorable position to monitor any runtime changes in the execution environment, in particular changes in resource conditions. And second, many adaptation actions involve adapting the lower-level services used by an application, which are easy to be managed and adapted at the middleware level. For these reasons, a number of middleware level adaptation approaches have been proposed in the last few years. The target of adaptation for the approaches listed in this section is invariably the lower-level services used by an application.

Some of the earliest middleware level adaptation approaches involved adapting the quality of data for multimedia-rich applications (e.g. audio or video streaming), in response to a change in the availability of network resources. The motivation for these approaches was that multimedia-rich applications typically have high demand for network resources, which may not always be met. On the other hand, the required adaptations to meet variations in network resources can be carried out easily at the middleware level (e.g. by adapting the frame-rate or resolution of a video at the middleware level). Some other earlier middleware level approaches have tried to adapt the communication services in response to changes in the execution environment, e.g. ACE [Sch94], Ensemble [VBH+98] etc. ACE uses service wrappers and C++ dynamic binding to support adaptable inter-process communication and event handling services. Ensemble provides a layered architecture that allows an application to select a particular communication protocol.

However, in the later middleware level approaches, the scope of adaptation has been much widened, to include, for example, adaptation of data encryption, caching, or even fault tolerance etc.

TAO (The ACE ORB) [SLM98] extends ACE to provide a CORBA-compliant real-time ORB. TAO enhances the standard CORBA event service to provide real-time dispatching and scheduling required by real-time applications. TAO employs the Strategy design pattern [GHJV95] for encapsulating different aspects of ORB internals, such as IIOP protocols, concurrency, scheduling, connection management etc. These strategies are specified in a configuration file, which is parsed by TAO to load the required strategies. However, TAO is configurable only at start-up time. Dynamic TAO [KRL+00] extends TAO by using computational reflection to provide a dynamically adaptive version of TAO. That is, using dynamic TAO, the relevant strategies can be reconfigured dynamically, thereby carrying out middleware level adaptation.

Another popular use of middleware level approaches is in intercepting remote methods invoked by an application, and redirecting or modifying these invocations dynamically as a part of dynamic adaptation. Examples of these approaches include QuO [ZBS97, VZL+98], ACT [SM04, MSKC04a], Orbix [Orb07] etc.

QuO (Quality Objects) [ZBS97, VZL+98] is a middleware level approach for controlling quality of service by adapting communication between a client and a remote object. In QuO, CORBA stubs and skeletons are wrapped with functional dele-

gates. These delegates are able to intercept client requests and server replies. QuO employs the concept of contracts for defining acceptable QoS regions. System condition objects are used for monitoring system status. Delegates consult the contract at runtime, and modify the client requests and server replies according to the current system status monitored by system condition objects.

ACT (Adaptive CORBA Template) [SM04, MSKC04a] is another CORBA based adaptive middleware for intercepting remote method invocations. ACT relies on CORBA portable interceptors for intercepting the remote invocations. Portable interceptors serve as generic hooks that a composer can use at runtime to integrate adaptive components. Rule-based interceptors allow a request to be redirected to a different target, or to a proxy component that performs adaptation.

Orbix [Orb07] is a CORBA compliant ORB used for providing adaptive fault tolerance to applications at the middleware level. This is achieved by integrating an object replication mechanism inside the CORBA ORB. Orbix provides support for configuring CORBA object replicas at startup time. *Adaptor objects* in Orbix are used to allow the modified ORB to use the services provided by a reliable multicast. Additionally, Orbix provides support for loading optional pluggable protocols defined in the configuration files.

Jorgensen et al. [JTMJ00] present an approach for customizing ORB implementations according to application-specific preferences, as a part of the Lasagne/J project. In this approach, the ORB implementations provide support for non-functional behavior of an application. The ORB implementations can be reconfigured on the basis of application-specific preferences, which are defined as independent policy objects. However, the policies are defined before runtime, and are translated into code and complied with the application. This implies that the adaptation policies cannot be changed at runtime.

Some of the middleware level adaptation approaches are reflective in nature. That is, these approaches support external regulation of their adaptation strategy. As discussed in Chapter 4, a basic requirement for an adaptive middleware system to be integrated with the CASA framework is that it should be reflective in nature.

Odyssey [NSN+97, Nob00] is one of the most flexible reflection-based adaptive middleware systems. We have already discussed Odyssey in Chapters 3 and 4 as a potential adaptive middleware system for integration with the CASA framework.

CARISMA [CEM03] is another popular reflection-based adaptive middleware

system. The behavior of the middleware can be customized according to the adaptation requirements of an application. In particular, the behavior of the middleware is described as a set of associations between the services that the middleware can adapt, the policies that can be applied to deliver these services, and the context that must be valid for a policy to be applied.

Even though the QuO middleware [ZBS97] described above does not allow an application to control the adaptation behavior at runtime, it can be suitably extended to support reflection. The concept of contracts used in QuO for defining acceptable QoS regions provides a possibility to control the runtime adaptation behavior of the middleware.

7.2 Application code level adaptation

Several approaches have been proposed for dynamic adaptation of application code, mostly as a part of research on dynamic evolution or incremental development of software applications. Below we describe some of the important approaches for carrying out runtime changes in the application code. Most of these approaches provide a runtime system for modifying the application code dynamically, and some of these approaches are supported by certain special languages or language constructs for their operation.

However, from the point of view of the CASA framework, most of the approaches proposed for dynamic adaptation of application components have a common deficiency: they do not allow replacing a currently running component, without discarding its current execution. That is, they require that either a component is not running at the time of replacement, or its current execution is discarded as a result of replacement. Though some approaches have been proposed recently for allowing a change in the code of a running component, they are still not flexible enough to allow resuming the execution from an arbitrary execution point within the changed code of the component. That is, although they allow re-executing a method whose code has been redefined, they do not allow resuming the execution from a point within the method body.

Some languages explicitly provide support for dynamic recomposition of components, such as CLOS and Python. Some other languages have been extended to supporting dynamic recomposition of components. For example, Open Java, R-

Java, and Adaptive Java all extend Java to include new keywords and constructs for supporting dynamic recomposition. [MSKC04b] provides a detailed discussion of language-based approaches for dynamic recomposition of application components. Below we briefly discuss some of the language-specific approaches for adapting application code.

Ensink et al. [ESA03, EA04] provide a programming framework for design and development of adaptive applications. The Program Control Language (PCL) consists of a small set of extensions to a base language, such as C++ or Java, to allow programmers to write high-level code for controlling the behavior of a target program at runtime. PCL uses Adaptors to wrap and adapt original application classes. Adaptors can change the behavior of a class by changing its variables and methods.

Dynamic Java classes [MPG+00] provide a generic approach to support evolution of Java programs by changing their classes at runtime. However, with this approach active classes, i.e. classes that have any of their methods running, cannot be replaced. Similarly, the approach of dynamic C++ classes [HG98] allows a version change of a running C++ class. However, with this approach, once the version of a class has been changed, only the new instances created after the version change belong to the newer version. The already created instances belonging to the older version are either allowed to continue till they expire normally or they are destroyed abruptly, while no attempt is made to replace these instances with ones belonging to the newer version. Clearly, such an approach is not suitable for our purpose.

The work by Hicks [Hic01] presents an approach for relinking a C program once a part of the program is updated, so that existing references to the old version of the code are linked to the new version of the code. The application developer is required to provide the code needed for a transition from the old version to the new version in a *dynamic patch* file. The actual relinking of the code is performed by extending the implementation of a dynamic linker in C. Therefore, this approach requires customized support at the programming language level. A similarity to our approach is that this work also requires specifying the timings (i.e. execution points) when an update can be performed explicitly in the updateable code.

Soules et al. [SAH+03] present an approach for dynamic reconfiguration of operating system components. This work describes an implementation for the K42 operating system by IBM, which is built using C++ objects. This approach allows an operating system component to be dynamically replaced with a new version.

However, it requires the existing calls executing on the old component to expire on their own (by keeping a track of the *generation count* of threads), before any new calls can be invoked on the new component. Since the operating system calls are usually short-lived, this approach is reasonable for replacing operating system components. The same assumption, however, does not apply to CASA, as a call executing on a replaceable component in CASA can run for an arbitrarily long time.

The HotSwap [Dmi01] technology provided in the Java HotSpot JVM allows replacing classes of a running application dynamically. With this technology, methods of a class can be redefined at runtime. However, in the current implementation, active methods of a class cannot be replaced. That is, the calls currently executing on an old method cannot be redirected to the redefined method, and are allowed to complete their current execution within the old method.

OSGi [OSG07] provides a framework for managing the life cycle of Java components. Cervantes and Hall [CH04] provide an OSGi based framework for dynamic adaptation of applications. Using this framework, Java components can be dynamically added or removed, and the bindings between these components can be dynamically changed. However, OSGi does not provide support for replacing the currently running components.

Below we discuss some more approaches that provide a runtime system for actualizing a dynamic change in application components.

The work by Hofmeister [Hof94] presents one of the earliest approaches for dynamic reconfiguration of applications. The approach adopted in this work is to extend the Polylith software bus (which is a software interconnection system) to allow adding, removing and replacing distributed components of an application at runtime. A similarity with our approach is that this approach also requires an application developer to specify reconfiguration points within a component where the component can be replaced. However, this approach is restricted to using an extension of the Polylith system for interactions between software components.

The work by Kon [Kon00] requires every component to explicitly specify its dependencies on other components. These dependencies are resolved at runtime by a component configurator attached to every component. The problem of runtime replacement of components (involving state transfer etc.) is not addressed in this work.

David and Ledoux [DL03] present an approach for runtime adaptation of ap-

plications by activating and deactivating certain meta-level components associated with the normal application components, in response to changes in the execution environment. However, the scope of adaptation is restricted here, as the meta-level components are usually limited to doing some kind of pre or post processing that can be adapted dynamically.

The Chisel adaptation framework [Kee04] allows dynamically associating Iguana/J metaobjects [RC02] with application objects. The Iguana/J metaobjects are implemented using custom metaobject protocols (MOPs). The behavior implemented by a metaobject is used for wrapping the original behavior of the base object to which this metaobject is attached. However, the adaptation achieved by using dynamic metaobject facilities is limited in scope, and imposes considerable overhead in accessing the base objects' data because of the computational costs associated with using reflection.

Rasche and Polze [RP03] present an approach for dynamic reconfiguration of component-based applications for the Microsoft .NET platform. This approach uses a transaction-based component model to decide the appropriate timing and order for reconfiguration. However, dynamic reconfiguration here implies adding and removing components, changing connections among components, or changing component attributes, while it does not provide means for dynamic replacement of components involving state transfer etc.

The Accord framework [LPH04] enables a dynamic change in application behavior according to the rules associated with every application component. However, with this approach, the interactions between application components need to be defined in terms of rules associated with the corresponding components, in order for these interactions to be changeable at runtime by changing the corresponding rules. Since the number of potential interactions between application components can be quite large, the number of possible rules can be exponential, making the rule management quite complex and inducing performance overhead due to execution of all these rules at runtime.

LuckyJ [Ori07] provides an approach for dynamic recomposition of components, where all communication between different components is mediated by a centralized coordination manager. The components requiring a certain service search for provider components using the coordination manager, which performs matching between the requester and provider components. The role of the coordination manager

here facilitates recomposition of components. However, this approach is suitable for recomposition at a very high level of component granularity, because of the high overhead incurred in managing components discovery, matching and interactions using the coordination manager.

In the last few years, a number of approaches have been proposed for dynamically adapting crosscutting aspects [KLM+97] of an application. These approaches are commonly referred as dynamic AOP approaches. Dynamic AOP presents a powerful mechanism for dynamic adaptation of applications, as quite often an adaptation of application behavior implies a corresponding change in some crosscutting concerns, such as access control, QoS, persistence management, fault tolerance etc.

As discussed earlier, the CASA framework relies on PROSE [NA05] for dynamic weaving and unweaving of aspects. We decided to use PROSE because of the flexibility and runtime efficiency it offers. However, there are some other dynamic AOP approaches developed over the last few years, which we briefly mention below.

TRAP/J [SMCS04] uses a two-step approach for dynamic weaving of aspects. In the first step, an aspect weaver inserts generic interception hooks into the application code at compile time. In the second step, a composer dynamically weaves new aspects into the application at runtime, and a meta-object protocol uses reflection to forward intercepted operations to these aspects.

JAsCo [VS04] uses the JVM debugger interface for dynamic aspect weaving. JAsCo introduces an aspect-oriented extension of Java for defining aspect beans and connectors. Aspect beans define join points and the corresponding advice code in an abstract and reusable manner. Connectors are used for deploying one or more aspect beans within a concrete context. JAsCo uses Java HotSwap technology in such a way that only those join-points where aspects are applied upon are trapped.

7.3 Software architecture level adaptation

A number of software architecture level adaptation approaches have been proposed in the last few years. In these approaches, software architecture models play a central role in reasoning about adaptation decisions and actions. Even though the adaptation actions are eventually executed at the middleware or application code level, the focus on the software architecture models for reasoning about adaptation decisions and actions distinguishes these approaches from the approaches discussed in the pre-

vious two sections which were primarily concerned with runtime implementation of adaptation actions. In other words, the primary focus of these approaches is to facilitate reasoning about the adaptation logic, irrespective of how the corresponding adaptation actions are carried out practically. However, most of these approaches also provide guidelines and runtime implementations for carrying out the adaptation actions.

Kramer and Magee [KM90] have presented one of the earliest software architecture based approaches for dynamic adaptation of distributed systems. The adaptation is achieved through dynamic recomposition of components. However, in this approach, a component needs to be in a quiescent state in order to be replaced. This approach provides an architecture-based dependent transaction model, which allows discovering dependencies between transactions, and thereby between components. The information provided by the transaction model helps in reasoning about the quiescent state of a component to be replaced. The application components affected by a change in application configuration, and the components directly adjacent to these components (according to the dependency relation) can be requested by a reconfiguration manager to reach the quiescent states, before the change in configuration is carried out.

Oreizy et. al. [OMT98] provide a software architecture-based approach for runtime software evolution, and discuss dynamic recomposition of application components at the architecture level. This approach is mainly focussed on the C2 architectural style [TMA+96], and requires all component interactions to be mediated through explicit *connectors*. The use of connectors makes it possible to alter a component composition by changing the component bindings of the connectors at runtime. The role of connectors here is similar to the role of Handle components in CASA. However, in this approach, all component interactions are mediated through connectors, while in CASA only the dynamically replaceable components need to be accessed through Handle components. This approach requires an accurate model of an application architecture to be available at runtime (as the reasoning about the application adaptation is carried out at the architecture level), and the mappings between the architecture model and implementation modules to be explicitly defined (these mappings allow the architecture level adaptation decision to be executed on the application). This imposes additional overhead for the application, and, moreover, induces scope for potential ambiguities and errors in defining the mappings

between the architecture model and implementation modules. This approach does not provide a standard for defining the adaptation policy, as it is targeted to aid the application architect in reasoning about and carrying out modifications at the application architecture level for runtime software evolution.

Rainbow [GCH+04, GCS03] presents another architecture-based approach for dynamic adaptation of software applications. In this approach, software architecture models are referred for monitoring an application and guiding dynamic changes to it. The adaptation strategies are defined at the architecture level in the form of constraints on the architecture. Since the centerpiece of this approach is the use of software architecture models, these models need to be provided to the adaptation infrastructure for monitoring and adapting an application. The adaptation infrastructure is divided into *system*, *architecture*, and *translation* layers. Monitoring of the execution environment and application adaptation are carried out at the system layer. The adaptation decisions are taken at the architecture layer, based on the architecture model of the application and the constraints specified on the model. Since the system layer and the architecture layer operate at different levels of abstraction, the role of the translation layer is to bridge the gap between these two different abstraction levels by mapping the information exchanged between system and architecture layers. However, this mapping may not be straightforward always and can be potentially error-prone, besides imposing additional overhead.

Kephart and Chess [KC03] envision autonomic computing systems to be able to deal with increasing software and environment complexity, thanks to the self-managing characteristic of these systems. Such self-managing systems are considered to be self-configuring, self-healing, self-optimizing and self-protecting. Recently some approaches have been proposed with the aim to turn this vision into reality. White et. al. [WHW+04] propose an architectural approach to developing autonomic systems. In this approach, an autonomic system is made up of autonomic elements, where each autonomic element is self-managing in its own behavior as well as in its interactions with other elements. However, this work presents the approach at an early stage. This work describes the broad requirements to be satisfied by autonomic elements and systems, and recommends certain design patterns for self-managing properties, while the details of carrying out self-management are not given.

Chapter 8

Conclusion and Future Work

8.1 Concluding discussion

In this dissertation, we have presented the CASA framework for enabling development and operation of dynamically adaptive applications. Our work on the CASA framework was motivated by the fact that software applications executing in dynamic computing environments, such as mobile and wireless environments, should be able to adapt to changes in their execution environment dynamically. We have broadly classified the changes in execution environment as: changes in contextual information (i.e. information about the context that may influence the service provided by an application, such as location, time-of-day etc.), and changes in resource availability (i.e. physical infrastructure available to the application for providing a service, such as communication bandwidth, memory, data resources etc.). The adaptation of an application, in response to a change in its execution environment, can be either compulsory for its continued execution (e.g. adjusting resource requirements in response to a loss of certain resources), or desirable for the purpose of improving its functionality or performance (e.g. providing context-dependent service in response to a change in contextual information).

The design of the CASA framework allows separation between the adaptation concerns and business concerns of an application. This separation plays an important role in the framework for providing a runtime system for handling adaptation concerns. Together, the separation of concerns and the CASA runtime system, considerably facilitate development of dynamically adaptive applications.

The adaptation policy of an application is defined in the application contract.

The application contract is external to the application, and is specified in an application-independent format. The contract-based adaptation policy plays a key role in separating the adaptation concerns of an application from its business concerns. Another important benefit of the contract-based adaptation policy is that it allows modifying the adaptation policy at runtime. Runtime changes to the adaptation policy are helpful in customizing the adaptation policy according to user's needs and preferences, as well as evolving the adaptation policy to integrate new adaptive behaviors or to handle new execution environment conditions.

The CASA framework provides a comprehensive adaptation solution to meet the wide range of adaptation requirements of a broad and diverse set of software applications executing in dynamic computing environments. In particular, the CASA framework supports a number of adaptation mechanisms for adapting any constituent of an application configuration. We classify the adaptable constituents of an application configuration as: application components, crosscutting aspects, application attributes, and lower-level services. The ability to adapt any of these constituents implies flexibility for the application developer to comprehensively meet the adaptation needs of a given software application.

For dynamically adapting lower-level services, the CASA framework relies on external reflection-based adaptive middleware systems. A number of such systems have been developed in the last few years. Some of these systems are specific to certain application domains, while others are more general. We have discussed integration of the CASA framework with a general and flexible reflection-based adaptive middleware system called Odyssey.

For dynamically adapting crosscutting aspects, the CASA framework relies on a flexible and efficient dynamic AOP system called PROSE. PROSE allows dynamic weaving and unweaving of aspects into / from an application at runtime, and is characterized by its flexibility and efficiency. We have discussed integration of the CASA framework with PROSE.

For dynamic recomposition of application components, we have developed an indigenous approach. Dynamic recomposition of components involves dynamic addition, removal and / or replacement of components. Our approach provides mechanisms for ensuring that the consistency of the application is not compromised as a result of dynamic recomposition. The details of this approach have been discussed in this dissertation. For a dynamic change in application attributes, the CASA frame-

work requires the concerned application to provide appropriate callback methods that are invoked by the CASA runtime system.

We have implemented a prototype system based on the CASA framework, details of which are presented in this dissertation. Performance evaluation of the prototype, mainly to evaluate our components adaptation approach, indicates that the overhead incurred is reasonable for most practical applications.

8.2 Future directions of work

We have presented the foundational work on developing a framework for dynamically adaptive applications. However, in order to realize the full potential of an enabling framework for dynamically adaptive applications, progress in several related research directions needs to be made. Below, we briefly describe some of the important directions for future work.

Ensuring completeness and correctness of alternative configurations: We have presumed that ensuring the completeness and correctness of alternative configurations specified in the application contract is the responsibility of the application developer. This involves verifying that a specified configuration is appropriate for the corresponding execution environment conditions, and all dependencies of the constituents of the configuration are well satisfied. Ensuring the completeness and correctness can be a non-trivial task, especially for large applications. An important direction of future work will be to develop suitable tools for aiding the application developing in carrying out this task effectively and efficiently.

Identifying safe points in dynamically replaceable components: In our discussion on dynamic replacement of components, we have presumed that the "safe" execution points (i.e. the execution points where the component replacement can be carried out without compromising the consistency) are defined in the component body by the application developer. Identifying such safe points is clearly a hard problem. As Gupta et al. [GJB96] have discussed, the problem of identifying the safe points is in general undecidable, and no generalized solutions exist for identifying such safe points. However, more work should be carried out in developing appropriate heuristics for identifying safe points (based on data flow analysis techniques etc.) for specific component types, in order to aid the application developer in this task.

Monitoring the execution environment: In this dissertation, we have given only a cursory treatment to the subject of monitoring the execution environment. More work clearly needs to be done in this direction. The swiftness in detecting changes in the execution environment has an obvious direct impact on the total response time in adapting the application to these changes. On the other hand, the ability to monitor a wide range of execution environment parameters allows flexibility to meet adaptation requirements of diverse applications. Therefore, efforts need to be made in developing efficient mechanisms for monitoring a wide range of environmental parameters. The developed mechanisms should be ideally easy to extend and evolve, in order to allow flexibility in monitoring new environmental parameters, or integrating more efficient mechanisms for monitoring existing environmental parameters.

Security issues: In this dissertation, we have not paid attention to ensuring the secure behavior of an application configuration, as security issues and measures are considered independent and orthogonal to the field of dynamically adaptive applications. We have presumed that the application developer verifies the secure behavior of alternative application configurations at the time of defining these configurations and specifying adaptation policy. However, the CASA framework also allows modifying the application contract at runtime. A runtime modification of the adaptation policy may involve adding new types of components, aspects or lower-level services dynamically. Such runtime modifications may in turn impose new security concerns. Future work may involve developing stringent security measures for ensuring the secure and predictable behavior of an application, even in the wake of runtime adaptation or changes to the adaptation policy.

General improvements of functionality and performance: General improvements may be carried out with an aim to improve the functionality and performance of the CASA runtime system. The design of the CASA framework is very modular, with only a loose coupling between different entities of the framework. Therefore, any of these entities can be evolved independent of the other entities of the framework. Improvements of the CASA runtime system may involve developing more efficient adaptation mechanisms to replace the existing mechanisms.

Bibliography

[ACKM04] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, 2004.

[API07] OWL-S API. Maryland Information and Network Dynamics Lab Semantic Web Agents Project (MINDSWAP), http://www.mindswap.org/2004/owl-s/api/. 2007.

[APK+03] Agarwala, S., Poellabauer, C., Kong, J., Schwan, K., and Wolf, M. System-level resource monitoring in high-performance computing environments. *Journal of Grid Computing*, 1(3):273–289, 2003.

[Bol99] Bollert, K. On weaving aspects. In *Proceedings of the ECOOP'99 Workshop on Aspect-Oriented Programming*, June 1999.

[Bro96] Brown, P.J. The stick-e document: A framework for creating context-aware applications. *Electronic Publishing*, 9(1):1–14, September 1996.

[CEM03] Capra, L., Emmerich, W., and Mascolo, C. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[CH04] Cervantes, H. and Hall, R.S. A framework for constructing adaptive component-based applications: Concepts and experiences. In *Proceedings of the 7th Symposium on Component-Based Software Engineering (CBSE 2004)*, May 2004.

[Dij82] E.W. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.

[DL03] David, P. and Ledoux, T. Towards a framework for self-adaptive component-based applications. In *Proceedings of the 4th International Conference on Distributed Applications and Interoperable Systems (DAIS 2003)*, 2003.

[Dmi01] Dmitriev, M. Towards flexible and safe technology for runtime evolution of Java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution at OOPSLA 2001*, October 2001.

[EA04] Ensink, B. and Adve, V. Coordinating adaptations in distributed systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[ESA03] Ensink, B., Stanley, J., and Adve, V. Program control language: A programming language for adaptive distributed applications. *Journal of Parallel and Distributed Computing*, 63(11):1082–1104, November 2003.

[GCH+04] Garlan, D., Cheng, S., Huang, A., Schmerl, B., and Steenkiste, P. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[GCS03] Garlan, D., Cheng, S., and Schmerl, B. Increasing system dependability through architecture-based self-repair. *Architecting Dependable Systems*, Springer-Verlag, 2003.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJB96] Gupta, D., Jalote, P., and Barua, G. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2), 1996.

[Gyg04] Gygax, A. *Studying the Effect of Size and Complexity of Components on the Performance of CASA.* Internship Report, Institut für Informatik, University of Zurich, http://www.ifi.unizh.ch/req/ftp/papers/casa-perf.pdf, 2004.

[HG98] Hjalmtysson, G. and Gray, R. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, 1998.

[Hic01] Hicks, M. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[Hof94] Hofmeister, C.R. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, CS-TR-3210, Department of Computer Science, University of Maryland, January 1994.

[JPr04] JProfiler. EJ Technologies, http://www.ej-technologies.com/products/jprofiler/overview.html. 2004.

[JTMJ00] Jorgensen, B.N., Truyen, E., Matthijs, F., and Joosen, W. Customization of object request brokers by application specific policies. In *Proceedings of the Middleware 2000*, April 2000.

[KC03] Kephart, J.O. and Chess, D.M. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.

[Kee04] Keeney, J. *Completely Unanticipated Dynamic Adaptation of Software*. PhD thesis, Department of Computer Science, Trinity College Dublin, October 2004.

[KLM+97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., and Irwin, J. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, June 1997.

[KM90] Kramer, J. and Magee, J. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), November 1990.

[KMM+98] Kenens, P., Michiels, S., Matthijs, F., Robben, B., Truyen, E., Vanhaute, B., Joosen, W., and Verbaeten, P. An AOP case with static

and dynamic aspects. In *Proceedings of the ECOOP'98 Workshop on Aspect-Oriented Programming*, July 1998.

[Kon00] Kon, F. *Automatic Configuration of Component-Based Distributed Systems*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

[KRL+00] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L.C., and Campbell, R.H. Monitoring , security, and dynamic configuration with the dynamicTAO reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000)*, April 2000.

[LMK+03] Lowekamp, B., Miller, N., Karrer, R., Gross, T., and Steenkiste, P. Design, implementation, and evaluation of the Remos network monitoring system. *Journal of Grid Computing*, 1(1):75–93, 2003.

[LPH04] Liu, H., Parashar, M., and Hariri, S. A component based programming framework for autonomic applications. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, May 2004.

[Mey92] Meyer, B. Applying "design by contract". *IEEE Computer*, 25(10):40–51, October 1992.

[MG03] Mukhija, A. and Glinz, M. CASA – A contract-based adaptive software architecture framework. In *Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2003)*, July 2003.

[MG04] Mukhija, A. and Glinz, M. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *Proceedings of the ICDCS 2004 Workshop on Distributed Auto-adaptive and Reconfigurable Systems*, March 2004.

[MG05a] Mukhija, A. and Glinz, M. The CASA approach to autonomic applications. In *Proceedings of the 5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN 2005)*, June-July 2005.

[MG05b] Mukhija, A. and Glinz, M. Runtime adaptation of applications through dynamic recomposition of components. In *Proceedings of the 18th International Conference on Architecture of Computing Systems (ARCS 2005)*, March 2005.

[ML98] Mezini, M. and Lieberherr, K. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of the 13th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA 1998)*, October 1998.

[MPG+00] Malabarba, S., Pandey, R., Gragg, J., Barr, E., and Barnes, J.F. Runtime support for type-safe dynamic Java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, 2000.

[MSKC04a] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., and Cheng, B.H.C. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[MSKC04b] McKinley, P.K., Sadjadi, S.M., Kasten, E.P., and Cheng, B.H.C. *A Taxonomy of Compositional Adaptation*. Technical Report MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, USA, July 2004.

[NA05] Nicoara, A. and Alonso, G. Dynamic AOP with PROSE. In *Proceedings of the International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005)*, 2005.

[Nob98] Noble, B.D. *Mobile Data Access*. PhD thesis, CMU-CS-98-118, School of Computer Science, Carnegie Mellon University, May 1998.

[Nob00] Noble, B.D. System support for mobile, adaptive applications. *IEEE Personal Communications*, 7(1), February 2000.

[NSN+97] Noble, B.D., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., and Walker, K.R. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[OMT98] Oreizy, P., Medvidovic, N., and Taylor, R.N. Architecture-based runtime software evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, 1998.

[OOP99] OOPSLA99. Panel discussion: "Are components objects?", http://www.acm.org/sigs/sigplan/oopsla/oopsla99/. 1999.

[Orb07] Orbix. Iona technologies. http://www.iona.com/products/orbix/. 2007.

[Ori07] Oriol, M. Primitives for the dynamic evolution of component-based applications. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC 2007)*, March 2007.

[OS07] OWL-S. OWL-based web ontology language for services, http://www.daml.org/services/owl-s/. 2007.

[OSG07] OSGi. Open services gateway initiative, http://www.osgi.org/. 2007.

[OWL07] OWL. Web ontology language, http://www.w3.org/2004/owl/. 2007.

[PAG03] Popovici, A., Alonso, G., and Gross, T. Just in time aspects: Efficient dynamic weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, March 2003.

[Pet00] Petre, L. *Components vs. Objects.* TUCS Technical Report, No. 370, Turku Centre for Computer Science, Finland, 2000.

[PGA02] Popovici, A., Gross, T., and Alonso, G. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, April 2002.

[PSA+03] Poellabauer, C., Schwan, K., Agarwala, S., Gavrilovska, A., Eisenhauer, G., Pande, S., Pu, C., and Wolf, M. Service morphing: Integrated system- and application-level service adaptation in autonomic systems. In *Proceedings of the 5th International Workshop on Active Middleware Services*, 2003.

[PSDF01] Pawlak, R., Seinturier, L., Duchien, L., and Florin, G. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of the REFLECTION 2001: Metalevel Architectures and Separation of Crosscutting Concerns*, September 2001.

[Qui07] QuickTime. QuickTime multimedia framework, http://www.apple.com/quicktime/. 2007.

[RC02] Redmond, B. and Cahill, V. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, June 2002.

[RP03] Rasche, A. and Polze, A. Configuration and dynamic reconfiguration of component-based applications with Microsoft .NET. In *Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[SAH+03] Soules, C., Appavoo, J., Hui, K., Silva, D., Ganger, G., Krieger, O., Stumm, M., Wisniewski, R., Auslander, M., Ostrowski, M., Rosenburg, B., and Xenidis, J. System support for online reconfiguration. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[Sch94] Schmidt, D.C. The ADAPTIVE communication environment: An object-oriented network programming toolkit for developing communication software. In *Proceedings of the 11th and 12th Sun Users Group Conference*, 1993-94.

[SDA99] Salber, D., Dey, A.K., and Abowd, G.D. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 1999.

[SLM98] Schmidt, D.C., Levine, D.L., and Mungee, S. The design of the TAO real-time object request broker. *Computer Communications, Elsivier Science*, 21(4):294–324, April 1998.

[SM04] Sadjadi, S.M. and McKinley, P.K. ACT: An adaptive CORBA template to support unanticipated adaptation. In *Proceedings of the 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[SMCS04] Sadjadi, S.M., McKinley, P.K., Cheng, B.H.C., and Stirewalt, R.E.K. TRAP/J: Transparent generation of adaptable Java programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 2004)*, October 2004.

[SPS04] Srinivasan, N., Paolucci, M., and Sycara, K. Adding OWL-S to UDDI, implementation and throughput. In *Proceedings of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, 2004.

[Szy98] Szyperski, C. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TG04] Tuduce, C. and Gross, T. Resource monitoring issues in ad hoc networks. In *Proceedings of the International Workshop on Wireless Ad-Hoc Networks (IWWAN 2004)*, 2004.

[TMA+96] Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, pages 390–406, June 1996.

[UPn07] UPnP. Universal plug and play, http://www.upnp.org/. 2007.

[VBH+98] Van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., and Karr, D. Building adaptive systems using Ensemble. *Software – Practice and Experience*, 28(9):963–979, August 1998.

[VS04] Vanderperren, W. and Suvee, D. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In *Proceedings of the Dynamic Aspects Workshop (DAW 2004)*, 2004.

[VZL+98] Vanegas, R., Zinky, J.A., Loyall, J.P., Karr, D., Schantz, R.E., and Bakken, D.E. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 1998)*, September 1998.

[WHFG92] Want, R., Hopper, A., Falcao, V., and Gibbons, J. The active badge location system. *ACM Transactions on Information Systems*, 10(1), 1992.

[WHW+04] White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., and Kephart, J.O. An architectural approach to autonomic computing. In *Proceedings of the International Conference on Autonomic Computing (ICAC 2004)*, May 2004.

[WSA+95] Want, R., Schilit, B.N., Adams, N.I., Gold, R., Petersen, K., Goldberg, D., Ellis, J.R., and Weiser, M. An overview of the ParcTab ubiquitous computing experiment. *IEEE Personal Communications*, 2(6), 1995.

[YN01] Yuan, W. and Nahrstedt, K. A middleware framework coordinating processor/power resource management for multimedia applications. In *Proceedings of the Globecom*, 2001.

[ZBS97] Zinky, J.A., Bakken, D.E., and Schantz, R.E. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

Appendix A

XML Schema of the Application Contract

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:element name="app-contract" type="app-contractType"/>

<xsd:complexType name="app-contractType">
    <xsd:sequence>
        <xsd:element name="context" type="contextType"
                     minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="contextType">
    <xsd:sequence>
        <xsd:element name="params" type="paramsType"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="config" type="configType"
                     minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger" use="required"/>
```

```
</xsd:complexType>

<xsd:complexType name="paramsType">
    <xsd:sequence>
        <xsd:element name="par" type="parType"
                     minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="ontology" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="parType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="unit" type="xsd:string" use="optional"/>
    <xsd:attribute name="value" type="xsd:string" use="optional"/>
    <xsd:attribute name="minv" type="xsd:double" use="optional"/>
    <xsd:attribute name="maxv" type="xsd:double" use="optional"/>
    <xsd:attribute name="enum" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:complexType name="configType">
    <xsd:sequence>
        <xsd:element name="resources" type="resourcesType"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="components" type="componentsType"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="aspects" type="aspectsType"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="callbacks" type="callbacksType"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="llservices" type="llservicesType"
                     minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:positiveInteger" use="required"/>
</xsd:complexType>
```

```
<xsd:complexType name="resourcesType">
    <xsd:sequence>
        <xsd:element name="hw" type="hwType"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="sw" type="swType"
                     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="hwType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="unit" type="xsd:string" use="optional"/>
    <xsd:attribute name="mpv" type="xsd:double" use="optional"/>
    <xsd:attribute name="lpv" type="xsd:double" use="optional"/>
    <xsd:attribute name="value" type="xsd:string" use="optional"/>
    <xsd:attribute name="enum" type="xsd:string" use="optional"/>
    <xsd:attribute name="reference" type="xsd:string" use="optional"/>
    <xsd:attribute name="essential" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:complexType name="swType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="reference" type="xsd:string" use="required"/>
    <xsd:attribute name="essential" type="xsd:string" use="optional"/>
</xsd:complexType>

<xsd:complexType name="componentsType">
    <xsd:sequence>
        <xsd:element name="comp" type="compType"
                     minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element name="binding" type="bindingType"
                     minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
```

```
</xsd:complexType>

<xsd:complexType name="compType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="class" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="bindingType">
    <xsd:attribute name="handle" type="xsd:string" use="required"/>
    <xsd:attribute name="boundto" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="aspectsType">
    <xsd:sequence>
        <xsd:element name="aspect" type=" aspectType"
                     minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="aspectType">
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="reference" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="callbacksType">
    <xsd:sequence>
        <xsd:element name="call" type="callType"
                     minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="callType">
    <xsd:sequence>
        <xsd:element name="arg" type="argType"
```

```
                    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="method" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="argType">
    <xsd:attribute name="value" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="llservicesType">
    <xsd:sequence>
        <xsd:element name="lls" type="llsType"
                    minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="llsType">
    <xsd:sequence>
        <xsd:element name="arg" type="argType"
                    minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="manager" type="xsd:string" use="required"/>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="operation" type="xsd:string" use="required"/>
</xsd:complexType>

</xsd:schema>
```

Appendix B

Resource Allocation Algorithms

B.1 Algorithm for new request

```
New_Request(IN: App A, IN: Context C, OUT: Config)
BEGIN
    Boolean success;
    List q;
    // 'sci' denotes selected configuration index in the list 'q'
    Integer sci;
    // 'Result' is a complex data structure with four attributes
    Result = {Integer numSuspend, Float netPrioritySuspend,
              Integer numReconfig, Float netPriorityReconfig};
    Result res;
    FOR(each Config G in C) /* starting from top to bottom */
    DO
        success = Allocate_Resources(A, G);
        IF(success == TRUE) THEN
            RETURN G;
        ENDIF
    ENDFOR
    q = NULL;
    FOR(each Config G in C) /* starting from top to bottom */
    DO
        res = Must_Allocate_Resources(A, G);
```

```
        IF(res != NULL) THEN
            q.newEntry (G, res.numSuspend, res.netPrioritySuspend,
                        res.numReconfig, res.netPriorityReconfig);
        ENDIF
    ENDFOR
    IF(q == NULL) THEN
        RETURN NULL;
    ENDIF
    sci = 0;
    FOR(Integer n=1; n<q.size; n++)
    DO
        IF(q[n].numSuspend < q[sci].numSuspend) ||
          (q[n].numSuspend == q[sci].numSuspend &&
              q[n].netPrioritySuspend < q[sci].netPrioritySuspend) ||
          (q[n].numSuspend == q[sci].numSuspend &&
              q[n].netPrioritySuspend == q[sci].netPrioritySuspend &&
              q[n].numReconfig < q[sci].numReconfig) ||
          (q[n].numSuspend == q[sci].numSuspend &&
              q[n].netPrioritySuspend == q[sci].netPrioritySuspend &&
              q[n].numReconfig == q[sci].numReconfig &&
              q[n].netPriorityReconfig < q[sci].netPriorityReconfig)
        THEN
                  sci = n;
        ENDIF
    ENDFOR
    q[sci].assignResources();
    RETURN q[sci].G;
END New_Request
```

B.2 Algorithm for allocating resources

```
Allocate_Resources(IN: App A, IN: Config G, OUT: Boolean)
BEGIN
    /* 'PL' is the priority list of all applications, sorted in the
```

```
       order of lowest priority to highest priority */
    List PL;
    G.assignFreeResources();
    IF(G.resourceReqs == NULL) THEN
        RETURN TRUE;
    ENDIF
    FOR (Integer n=0; PL[n].priorityValue < A.priorityValue; n++)
    DO
        App K = PL[n];
        Config currentConfig = K.currentConfig;
        List configList = K.allConfigs;
        Config newConfig = currentConfig;
        FOR(Integer i=0; i < configList.size; i++)
        DO
            IF(configList[i].savingValue > newConfig.savingValue) THEN
                newConfig = configList[i];
            ENDIF
        ENDFOR
        IF(newConfig != currentConfig) THEN
            K.changeConfig(newConfig);
            G.assignFreedUpResources();
            IF(G.resourceReqs == NULL) THEN
                RETURN TRUE;
            ENDIF
        ENDIF
    ENDFOR
    RETURN FALSE;
END Allocate_Resources
```

B.3 Algorithm for compulsorily allocating resources

```
Must_Allocate_Resources(IN: App A, IN: Config G, OUT: Result)
BEGIN
    // 'Result' is a complex data structure with four attributes
```

```
Result = {Integer numSuspend, Float netPrioritySuspend,
          Integer numReconfig, Float netPriorityReconfig};
Result res;
/* 'PL' is the priority list of all applications, sorted in the
   order of lowest priority to highest priority */
List PL;
G.assignFreeResources();
FOR(Integer n=0; n < PL.size; n++)
DO
    App K = PL[n];
    Config currentConfig = K.currentConfig;
    List configList = K.allConfigs;
    Config newConfig = currentConfig;
    FOR(Integer i=0; i < configList.size; i++)
    DO
        IF(configList[i].savingValue > newConfig.savingValue) THEN
            newConfig = configList[i];
        ENDIF
    ENDFOR
    IF(newConfig != currentConfig) THEN
        K.changeConfig(newConfig);
        G.assignFreedUpResources();
        res.numReconfig++;
        res.netPriorityReconfig = res.netPriorityReconfig +
                                  K.priorityValue;
        IF(G.resourceReqs == NULL) THEN
            RETURN res;
        ENDIF
    ENDIF
ENDFOR
FOR(Integer n=0; PL[n].priorityValue < A.priorityValue; n++)
DO
    App K = PL[n];
```

```
        IF(overlap(K.resourceAllocation, G.resourceReqs) != NULL) THEN
            K.suspend();
            G.assignFreedUpResources();
            res.numSuspend++;
            res.netPrioritySuspend = res.netPrioritySuspend +
                                     K.priorityValue;
            IF(G.resourceReqs == NULL) THEN
                RETURN res;
            ENDIF
        ENDIF
    ENDFOR
    RETURN NULL;
END Must_Allocate_Resources
```

B.4 Algorithm for reduced availability of resources

```
Less_Resources(IN: Resources currentAvailability)
BEGIN
    /* 'PL' is the priority list of all applications, sorted in the
       order of lowest priority to highest priority */
    List PL;
    Resources resourceDeficit;
    resourceDeficit = currentAllocation - currentAvailability;
    FOR(Integer n=0; n < PL.size; n++)
    DO
        App K = PL[n];
        Config currentConfig = K.currentConfig;
        List configList = K.allConfigs;
        Config newConfig = currentConfig;
        FOR(Integer i=0; i < configList.size; i++)
        DO
            IF(configList[i].savingValue > newConfig.savingValue) THEN
                newConfig = configList[i];
            ENDIF
```

```
        ENDFOR
        IF(newConfig != currentConfig) THEN
            K.changeConfig(newConfig);
            resourceDeficit = currentAllocation - currentAvailability;
            IF(resourceDeficit <= 0) THEN
                RETURN;
            ENDIF
        ENDIF
    ENDFOR
    FOR(Integer n=0; n < PL.size; n++)
    DO
        App K = PL[n];
        IF(overlap(K.resourceAllocation, resourceDeficit) != NULL) THEN
            K.suspend();
            resourceDeficit = currentAllocation - currentAvailability;
            IF(resourceDeficit <= 0) THEN
                RETURN;
            ENDIF
        ENDIF
    ENDFOR
END Less_Resources
```

B.5 Algorithm for increased availability of resources

```
More_Resources(IN: Resources currentAvailability)
BEGIN
    /* 'PL' is the priority list of all applications, sorted in the
       order of lowest priority to highest priority */
    List PL;
    Resources resourceSurplus;
    resourceSurplus = currentAvailability - currentAllocation;
    FOR(Integer n=(PL.size-1); n >= 0; n--)
    DO
        App K = PL[n];
```

```
        Config currentConfig = K.currentConfig;
        List configList = K.allConfigs;
        /* configList is sorted from the most preferred (i.e. highest
        in current <context> element) to the least preferred */
        FOR(Integer i=0; i < configList.size; i++)
        DO
            IF(configList[i] == currentConfig) THEN
                EXIT; //exit this inner FOR loop
            ENDIF
            Resources reqdResources = configList[i].resourceReqs -
                                      currentConfig.resourceReqs;
            IF(reqdResources <= resourceSurplus) THEN
                K.changeConfig(configList[i]);
                resourceSurplus = currentAvailability -
                                  currentAllocation;
                IF(resourceSurplus == 0) THEN
                    RETURN;
                ELSE
                    EXIT; //exit this inner FOR loop
                ENDIF
            ENDIF
        ENDFOR
    ENDFOR
END More_Resources
```