Dynamic Upgrade of Distributed Software Components

vorgelegt von

Dipl.-Ing.

Marcin Solarski

aus Kraków

von der Fakultät IV –Elektronik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing. –

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr.-Ing. Stefan Jähnichen
Berichter:	Prof. Dr. Dr. h.c. Radu Popescu-Zeletin
Berichter:	Prof. Dr. Miroslaw Malek

Tag der wissenschaftlichen Aussprache: 26. Januar 2004

Berlin 2004
D 83

Zusammenfassung

Die Aktualisierung von komplexen Telekommunikationssystemen, die sich durch die ihnen eigene Verteiltheit und hohe Kosten bei System-Nichtverfügbarkeit auszeichnen, ist ein komplizierter und fehleranfälliger Wartungsprozess. Noch stärkere Herausforderungen bergen solche Software-Aktualisierungen, die die Systemverfügbarkeit nicht beeinträchtigen sollen. *Dynamic Upgrade* ist eine Wartungstechnik, die das Verwalten und die Durchführung von Software-Aktualisierung automatisiert und damit den Betrieb des Systems während der Wartungszeit nicht unterbricht.

In dieser Arbeit wird das *Dynamic Upgrade* als ein Sonderfall der Bereitstellung und Inbetriebnahme (*Deployment*) von Software betrachtet, in dem Teile der einen Dienst repräsentierenden Software durch neue Versionen im laufenden Betrieb ersetzt werden. Die Problemstellung des *Dynamic Upgrade* wird anhand einer vom Autor erarbeiteten Taxonomie erläutert, die die Entwurfsmöglichkeiten für ein System zur Unterstützung von *Dynamic Upgrade* hinsichtlich dreier Systemaspekte klassifiziert: *Deployment*, Evolution und Zuverlässigkeit (*Dependability*). Mit Hilfe dieser Taxonomie lassen sich auch andere Systeme zur Unterstützung von *Dynamic Upgrade* miteinander vergleichen. Aufbauend auf einem ausführlichen Vergleich über existierende Ansätze zur Unterstützung von *Dynamic Upgrade*, wird in der vorliegenden Arbeit eine Lösung entwickelt und dargestellt, die *Dynamic Upgrade* in verteilten komponentenbasierten Software-Systemen ermöglicht.

Ausgehend von der Problemanalyse wird mit Hilfe des *Unified Process* ein als *Deployment and Upgrade Facility* bezeichnetes Modell entwickelt, das sowohl die benötigten Leistungsfähigkeiten eines *Dynamic Upgrade* unterstützenden Systems als auch Eigenschaften von aktualisierbaren Software-Komponenten beschreibt. Dieses Modell ist Plattform-unabhängig und einsetzbar für mehrere unterliegende Middleware-Technologien. Das Modell wird in einem Java-basierten prototypischen Rahmenwerk programmiert und um plattformspezifische Mechanismen auf der Jgroup/ARM Middleware erweitert. Das Rahmenwerk umfasst allgemeine Entwurfslösungen und –muster, die sich für die Konstruktion einer Unterstützung für *Dynamic Upgrade* eignen. Es erlaubt die Kontrolle der Lebenszyklen von Aktualisierungsprozessen und ihre Koordination im Zielsystem. Darüber hinaus definiert es eine Reihe von Unterstützungsmechanismen und Algorithmen für den dynamischen Aktualisierungsprozess, der gegebenenfalls mit unterschiedlichen Zielsetzungen und unter verschiedenen Randbedingungen erfolgen soll. Insbesondere wird ein Aktualisierungsalgorithmus für replizierte Software-Komponenten dargestellt.

Das entwickelte Rahmenwerk wird zwecks Plausibilitätsprüfung der dargestellten Ansätze und zur Auswertung der Auswirkungen der *Dynamic Upgrade* unterstützenden Mechanismen im Hinblick auf Systemperformanz in mehreren Experimenten eingesetzt. Diese quantitative Evaluierung der Experimente führt zu einer Spezifikationen eines einfachen Bewertungsmaßstabs (*Benchmark*), der sich zum Vergleich von *Dynamic Upgrade* unterstützenden Systemen eignet.

Stichwörter: *dynamic (online) upgrade*, Dienstbereitstellung, Verwaltung von Systemänderungen zur Laufzeit, Hochverfügbarkeit, Komponenten-Technologie, verteilte objektorientierte Systeme, *middleware*

Abstract

Upgrading complex telecommunication software systems, characterized by their inherent distribution and a very high cost of system unavailability, is a difficult and error-prone maintenance activity. Even more challenging are such software upgrades that do not compromise the system availability. Dynamic upgrades is a technique, which automates performing and managing upgrades so that the software system remains operational during the upgrade time.

In this thesis, the dynamic upgrade is considered as a special case of software deployment, in which a running service has to be replaced with its new version. The problems of dynamic upgrades are introduced using a novel taxonomy that classifies the design issues to be solved when building support for dynamic upgrade with regard to three system aspects: deployment, evolution and dependability and provides a reference to comparing other systems supporting dynamic upgrades. An extensive and thorough survey of existing approaches to dynamic upgrades follows and, furthermore, is as a starting point to designing a solution supporting dynamic upgrades in distributed component-based software systems.

Derived from the problem analysis, a model called Deployment and Upgrade Facility describing the capabilities needed for managing and performing dynamic upgrades as well as properties of upgradable software components is developed using the Unified Process approach. The model is platform independent and can be used with a range of underlying middleware technologies. The model is implemented in a Java-based prototypical framework and extended with platform specific mechanisms on top of the JGroup/ARM middleware. The framework captures common design solutions and patterns for building a support for dynamic upgrade. The framework allows for controlling life-cycle and coordination of upgrade processes in the system. It also defines a number of supporting mechanisms and algorithms for the upgrade process. A special attention is drawn to an upgrade algorithm for replicated software components for achieving a synergy of replication techniques and dynamic upgrade .

The developed framework is used to validate the feasibility of the approach and to measure the overhead of the mechanisms supporting dynamic upgrade with regard to the performance of the system being upgraded in a number of practical experiments. This quantitative evaluation of the experiments leads to a specification of a simple benchmark for systems supporting dynamic upgrades.

Keywords: dynamic (online) upgrading, service deployment, runtime change management, high availability, component based technology, distributed object systems, middleware

Acknowledgements

Hereafter I would like to express my great thanks to Prof. Radu Popescu-Zeletin (Technical University of Berlin) who has supervised my thesis and helped me with his comments and hints on improvements of the thesis text. I appreciate the support of Prof. Miroslaw Malek from the Humboldt University Berlin who has contributed with his reviews and discussions especially on the evaluation of the experimental part of the work.

At this point, I would like to mention my colleagues at Fraunhofer FOKUS that has been my office for the time of writing the dissertation. Many of them supported me with their expertise and the practical know-how that was of a great value to me. Dr. Eckhard Möller, the Head of the PLATIN Competence Center, as well as Gerd Schürmann were the ones encouraging me to progress, Dr. Klaus-Dieter Engel and Dr. Klaus-Peter Eckert were investigating my attempts thoroughly and giving their feedback. I also appreciate the support of Thomas Becker who served me with his Java expertise and Robert Joop who friendly assisted me with configuring and tuning the Linux-based testbed for my experiments. Prof. Emanuell Mendes from University of Sao Paulo in Brazil visiting the FOKUS institute gave me a hand expressing his critics in the early stages of writing this thesis. Last but not least, Jürgen Dittrich and Oliver Fox would always express their enthusiasm about the intermediate results and give me lots of pragmatic advice.

This work would not be what it is now without the support of Prof. Bjarne Helvik from NTNU. Hein Meling has supported me with his expertise in the Jgroup toolkit that I used to implement a part of the software solution resulting from this thesis. At the same time, I am grateful to Prof. Oddvar Risnes, the director of the Telenor R&D Department in Telenor, Norway for providing me with his support during my scholarship at Telenor. Sune Jackobson and Erik Berg had a few fruitful discussions at that time as well.

Finally, I owe great thanks to my family: my dad and mum who have been always encouraging me to carry on with my education and who enabled many of my previous achievements that led me to successful finalization of my doctorate; my sister Justyna who mobilized me to be an example to her. Last but not least, a big *beso* for my lovely wife Iliana for her patience during all that time I had to be away on conferences and other events related to my work and a nice *ambiente* that she created at home.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

The Basics

1 Introduction

Change management is indispensable in most distributed software systems which are continually being modified throughout their life cycle. Managing the changes at runtime in distributed systems is a complex task that requires much expertise. In highly-available systems, performing software upgrades is even more challenging because of strong requirements on upgrade process dependability and hard constraints on the system downtime. Dynamic upgrades, a technique that allows for introducing changes to a running software system, deals with these challenges. Generally speaking, it allows for replacing pieces of software in a running system with new ones.

Dynamic upgrades is the central topic of this thesis. It is investigated in the context of distributed systems and component-based middleware technology. The rest of this chapter is structured as follows: in section 1.1 the motivation for investigating these technique is given. the goals, scope and approach of this thesis are described in sections 1.2, 1.3 and 1.4 correspondingly. The structure of the rest of the thesis is given in section 1.5.

1.1 Motivation

This thesis is motivated by three well-known facts about the software systems:

- *Evolution of software systems is inevitable***.** Software systems change constantly and need to be upgraded a couple of times in their life time. The spectrum of software change is wide and ranges from introducing program corrections or performance improvements of existing system components to complex changes of the overall functionality and/or structure of the system necessary to adapt the system to new user requirements.
- *High availability and evolution mismatch***.** Conventional software upgrade mechanisms are not good enough for high available systems as they introduce maintenance breaks which reduces the system availability. In a conventional approach, the system runtime has to be interleaved with maintenance breaks in which the necessary changes are manually applied to the system. This approach, however, is not suitable in distributed systems that have to be high available.
- *Software upgrades are critical to the operation of the system***.** In many existing systems, human interaction is required to upgrade a system. However, upgrading a software system introduces a high risk of an error during the upgrade or potentially after the upgrade is complete. Moreover, managing changes to a running system in systems may require high skills that makes the upgrade process expensive.

The technology of dynamic upgrades is not application domain specific. There are many application domains, in which dynamic upgrades can be profitable. In most cases, these applications belong to a wider class of software systems having strong requirements on high availability and reliability. Typical application domains include:

- *Infrastructure***:** Telecommunications, electricity supply.
- *Electronic Business and Government***:** Online shopping systems serving several time-zones, mission-critical nation-wide governmental facilities (e.g. tele-voting system).
- *Banking***:** ATMs, online banking.
- *Industrial process control***:** Furnaces or oil refineries operating around the clock.
- *Medical applications***:** Life support systems.

Moreover, many researchers and domain experts are aware of the need of investigating techniques enabling upgrading systems on the fly. For instance, in the telecommunication domain, Schulzrinne states in a workshop on the IP telephony [100]: *"Probably the major challenge faced by Internet telephony is moving from the current Internet reliability of about 99% or 99,5% to 99,999, i.e. no more than five minutes of unavailability per year. This requires a different mindset, not just protocol and technology fixes, as upgrades have to be done while the system is running and every component has to engineered to have a hot stand-by"*.

Many applications from various industry domains require some support for dynamic upgrades. This support is, thus, not application domain specific and should be provided as a *generic service* to the domain specific applications. In the distributed systems, generic services are part of a middleware platform. That is why, dynamic upgrades are investigated in the context of middleware technologies in this thesis.

Furthermore, the thesis is motivated by the arguments resulting from the survey of the state of the art:

- *Weak support for dynamic upgrades for standard distributed software technologies*. The standard middleware has only recently identified the problem of dynamic upgrades and started related initiatives. The ongoing or first versions of the standards provide solutions that are focused on one specific way of performing upgrades.
- *A more general approach is required.* The existing solutions are proprietary and the standard solutions to come are specific to the underlying software technology. There is a need to handle the dynamic upgrades at a higher level of abstraction for the sake of the solution's reusability and adaptability to the underlying technologies.

1.2 Goals

The primary goal of this thesis is to develop a technology-independent model describing functionalities and capabilities supporting upgrades of distributed systems at runtime. The model should:

- ***P1.*** describe the capabilities needed to support dynamic upgrades in distributed software systems,
- ***P2.*** define the properties of distributed software components enabling their upgradability,
- ***P3.*** be general enough to abstract the specifics of the component application domain and possibly the idiosyncrasies of a certain programming environment,
- ***P4.*** be applicable to the currently available middleware technologies,
- ***P5.*** be expressed in a well-known notation to be easily understandable and reusable.

The second goal of this thesis is to develop a system compliant to the model above. The solution should:

- ***P6.*** support performing and managing dynamic upgrades of distributed software components in compliance with the model defined above,
- ***P7.*** be practically validated as a proof of concept using, and extending if necessary, one of the existing middleware technologies providing the runtime environment and deployment support for distributed software systems,
- ***P8.*** be extendable by various supporting mechanisms for performing and managing dynamic upgrades,
- ***P9.*** be as much reusable and portable as possible.

P10. be evaluated with regard to the promised increase in system availability using quantitative tests.

1.3 Scope

This work addresses miscellaneous issues relevant to building distributed software systems capable of being upgraded on the fly. The scope of investigation is primarily targeted at systems built with distributed object approach, distributed component technology extensions and its existing middleware realizations. The investigation scope is limited to issues regarding the application and the middleware platform that the distributed applications are running on. The questions that are to be answered in this thesis are:

- What deployment and runtime mechanisms does a platform, on which distributed application run, need so that applications can be dynamically upgraded?
- What deployment techniques are required for such applications?
- In how far is it possible to upgrade software on the fly without any breaks in its operation?
- When and under what conditions is software upgrade possible on the fly?
- How to perform an application upgrade so that it is transparent to the users? What kind of transparency issues have to be addressed?
- How does software replication and other high availability techniques relate to dynamic upgrade?
- Is it possible to develop applications without regarding the dynamic upgrade capability? If not, how does the software have to be extended for this reason? What consequences does it have on the application development?
- What is the overhead of introducing the dynamic upgrade capability to the original software system?

On the other hand, other aspects of distributed systems are **not** taken into consideration, including:

- *Specific hardware support needed to upgrade applications on the fly.* It is one of the requirements that the upgrade support should be realized in software and run on general purpose hardware. Moreover, the software solution should be easily portable.
- *Features needed by existing analysis and design processes, notations and programming languages to support building upgradable software.* Even though the software engineering aspects are investigated in this thesis, it is assumed that the upgradable component-based software is developed using existing and main-stream object-related and related development methodologies and tools.
- *Mechanisms allowing for validating component implementations and their substitutability in particular.* Validation and other issues related to software evolution are not investigated in this thesis. The solution presented is mainly concerned with deployment and partly dependability related issues when performing dynamic upgrades. These mechanisms, however, impact the freedom of changes supported by the dynamic upgrades and therefore evolutionary aspects are mentioned wherever it is necessary.

1.4 Approach

In this work, the problem of dynamic upgrades is investigated from the deployment

perspective in that upgrade is a special case of software deployment. As a result, the starting point of constructing a model for dynamic upgrades is a broader in scope deployment model for distributed software. The model for service deployment and upgrades is then constructed and validated in the projects the author has been participating. An important part of this model is a component based service model that describes the life cycle of the service and the way the service is structured.

The deployment model is then elaborated and focused with regard to dynamic upgrades. It forms a framework that captures design solutions and patterns for building a dynamic upgrade support.

The model is complemented by the dynamic upgrade support mechanisms. Selected mechanisms supporting dynamic upgrade are designed and implemented. As a support for dynamic upgrades is a generic service, the middleware platform is found suitable for placing this support. In the thesis, the model and the supporting mechanisms are proposed to extend available middleware platforms. The extensions are called Deployment and Upgrade Facility and are implemented as a series of prototypes as a proof of concept.

On the other hand, this work does not intend to extend other elements of the development and runtime environment for the distributed software systems. In particular, the programming languages and corresponding development tools, like compilers, linkers or interpreters. Neither are extensions of the operating systems underlying the middleware layer investigated. Moreover, the approach presented in this thesis assumes using standard hardware platforms and typical programming environments to provide a solution that has a potential to be used by in the main-stream and commodity distributed applications

1.5 Thesis Structure

In this chapter, the introduction to the topic of dynamic upgrades was given. After the motivation leading to this research including some real world applications was presented, the thesis goals and the approach was put forward. The rest of the chapter was dedicated to a brief but wide-angled survey of research topic related to dynamic upgrades.

In the subsequent chapters of this thesis, the technology of dynamic upgrades is mostly investigated from the *deployment perspective*. An upgrade is shown as a deployment step in the software life cycle.

The thesis is divided into two parts. *The Basics* and *The Solutions*. The first part presents some introductory and background information needed to comprehend the rest of the thesis. It also includes a comprehensive survey of the previous work on dynamic upgrades as well as presents dynamic upgrade support of some mainstream middleware technologies. Part *Solutions* describes my proposal of an approach to dealing with dynamic upgrades.

The remaining chapters of the introductory part of this thesis are structured as follows: Chapter 2 gives the background information on the context of this thesis: distributed systems and the component-oriented software engineering paradigm. Chapter 3 introduces the basic concepts and problems related to dynamic upgrades.

Chapter 4 presents the support for dynamic upgrades in the mainstream middleware platforms and discusses the needed extensions. It also investigates a range of dynamic upgrade support systems, DUSS for short, and concludes with a feature comparison between them. It is also a starting point to define the requirements for the solution presented in the next part of this thesis.

Part *The Solution* contains the following chapters. Chapter 5 specifies a model for deployment and dynamic upgrades for distributed component-based software systems. The model is

determined by a list of requirements for a solution supporting service deployment and dynamic upgrade, in particular. This model includes two parts: (1) a use case model which defines the capabilities needed for service deployment and upgrades in component-based distributed software systems constructed and (2) a component model, which defines how component-based distributed system is structured. Chapter 5 also contains some implementation details of the model.

The remaining chapters investigate the key aspects of the DUSS fundamental to this thesis: dynamic upgrade management and algorithms. Whereas upgrade management is covered in chapter 6, a variation of upgrade algorithms as well as a number of dynamic upgrade support mechanisms are presented in chapter 7. A practical evaluation of the implementation of the presented solution is the topic of chapter 8. Chapter 9 concludes the main part of the thesis and gives a summary of the thesis contributions as well as some proposals for future work.

Attached to the thesis is the list of acronyms, a glossary of terms and a list of references.

2 Distributed Component Technology

This chapter introduces the context of this thesis, namely distributed systems and the component oriented software technology. In section 2.1 the key characteristics of distributed systems in general are briefly presented. The focus of description is put on dependable distributed systems, in which dynamic upgrades is of particular importance. Then, a brief overview of object oriented and component oriented paradigm and its application to distributed systems is given. A number of concepts and terms related to these technologies, like object, type, component or container, are introduced that are used throughout the thesis.

2.1 Characteristics of Distributed Systems

In this section provides some background information on distributed systems. The basic characteristics of distribution systems in general is given first. Then, some introductory information concerning the dependability of computer systems is presented. This information is required to understand the challenges of supporting dependable dynamic upgrades in a distributed environment.

2.1.1 Distribution Complexity

Design and development of distributed system is quite an intricate task that involves understanding many aspects of system complexity introduced by their distribution. Some of the problems that have to be considered when developing dependable distributed systems are listed below [67].

- *Interconnection.* In distributed systems, different components have to communicate and interoperate with each other in a consistent, reliable and efficient way. These components are often developed independently from each other in various isolated environments. This heterogeneity of components makes system integration and reliable component interworking at system runtime difficult.
- *Interference.* System components, developed in isolation and having functionality as designed may yield unwanted effects when combined in a system. Developing a reliable system involves considering explicit and implicit dependencies between components and potential interference of their behavior in the system.
- *Partial Failures.* Components of distributed systems usually run in different runtime environments that are independent from each other in terms of occurring failures. Consequently, it may happen that one part of a distributed system fails whereas the rest of the system is capable of providing its services. This characteristics of distributed systems, on one hand, requires applying mechanisms that allow the system to properly work in spite of the system's partial failures. On the other, the fact that only part of the system has failed allows applying mechanisms that would adapt the system to the new execution conditions and make use of the currently available system resources.
- *Propagation of effect*. In contrast to faults in centralized systems, faults in distributed systems cannot be always easily localized, i.e. assigned to one specific component. Effects of such faults, failures, usually propagate and are detected in some other system component. This characteristics of failures is typical for software bugs, also called Heisenbugs in bibliography [6], which are the most common sources of system crashes in current complex distributed systems. Generally, their occurrence is neither repeatable nor deterministic (does not depend on the system inputs) what makes their tracking down and elimination even more difficult.

2.1.2 Dependability

This section describes the dependability and related concepts as it is understood in this thesis. The definitions are then used when discussing the characteristics of upgrade algorithms.

Some information processing system are mission-critical in terms that the user depends on the services offered by the system. A system failure may be very costly with regard to [32]:

- Loss of production (e.g. manufacturing applications),
- Loss of clients (due to lack of user confidence),
- Loss of human life (life-critical applications).

Because all applications are susceptible to hardware and software faults, both accidental and intentional, some mechanisms have to be provided to support system dependency.

In the context of distributed systems, *dependability* is a collective term to describe availability and reliability. The aspects are not totally independent from each other and their definitions and interdependencies will be discussed below.

- *Reliability* is the likelihood that a middleware platform provides the middleware services in a well specified way, even in the case of middleware or application failures. Reliability services should be offered without unacceptable performance loss.
- *Availability* is the fraction of time a middleware platform is operational (e.g. 7x24) without unacceptable loss of performance due to non-functioning parts of the middleware platform. Non-functioning may have different reasons: e.g. failures, system overload or management breaks The availability of an platform with (real-) time constraints may be described as follows: a component is available if a certain percentage (e.g. 95%) of requests are serviced within an required timing constraint. This must be measured over a period of time significantly longer than the timing constraint

2.2 Distributed Objects

The object-oriented approaches in software technology is dated as long as Simula`67. Initially, they have spread in the area of programming languages, like Smalltalk, C++, Java and C#. Later they have been used in a wider scope in computer science: operating systems, databases, distributed processing and object-oriented analysis and design. The idea of representing system parts distributed on a number of computer nodes as objects, and their interactions as exchanging messages between the objects has been appealing to the designer of distributed systems due to:

- *ability to deal with distribution complexities* described in section 2.1.1. One of the fundamental ideas of the object-oriented paradigm is encapsulation. It makes it possible to abstract from certain aspects of modeled reality and to hide it in the internal implementation of an object. The aspects of distribution, including location and failures can be encapsulated to certain degree. This idea has yielded the so called *distribution transparencies* introduced in the Open Distributed Processing[46].

enabling a more flexible design. The first distributed system were designed in the client-server paradigm. To overcome certain limitation of this paradigm, multi-tier architecture had to be developed. The idea of decomposing a modeled system of distributed nature into logical entities interacting with each other in a mesh-like topology enabled a new dimension of design flexibility in distributed systems. The object-oriented paradigm was ideally suited to support this kind of architecture.

straightforwardness of mapping from modeling domain to the design concepts and

implementation artifacts. Building distributed systems in the same paradigm in all the phases of software development process, including system analysis, design and implementation, allowed for defining straightforward stage transitions between these phases and mappings between the results achieved in each of the phase.

In last two decades, object-orientation is the underlying paradigm of many conceptual architectures for distributed object systems, like ISO Open Distributed Processing[45][46], OMG`s Object Management Architecture or TINA[126] specified by TINA Consortium, as well as frameworks and platforms implementing these architectures, including OpenDoc[84], OMG's CORBA[72][73] and Java RMI, which is part of J2EE[119] now.

In spite of the advantages identified above, the object-oriented paradigm did not handle many issues of distributed system suitably. Some of these problems, related to software deployment and upgrades in particular are listed below[123]:

Objects are not deployable. The notion of objects is very appropriate for software design and implementation. It provides a life-cycle model of any entity and a simple and expressive interaction model. However, this is not enough to cope with software during the deployment phase. The object-oriented paradigm does not define any means to structure and package parts of the software system so that it can be deployed in a destination infrastructure.

No implicit notion of composition. The definition of an object does not deal with composition. Its definition focus on other aspects: abstraction, encapsulation of state and behavior, as well as polymorphism. This fundamental characteristics of objects does not permit to use objects as units of software composition in a straightforward way. Furthermore, the object technology does not introduce any gluing concepts to combine objects together in a simple and easy way. This led to building object-oriented software systems and frameworks that were of monolithic character and made assembling and combining objects into distributed applications difficult.

Reuse difficulties in evolving systems. The basic mechanism for reuse in object-oriented systems is class inheritance. However, this approach does not work well in evolving systems. If a class and its subclasses evolve independently, this may lead to problems when using implementation inheritance, a phenomenon called fragile base class problem in the literature.

Economical Immaturity. Except for technical issues concerning object-orientation, a problem of economical immaturity is pointed out. The promoters of object technology are argued to have underestimated the non-technical market-oriented aspects of their solutions in the destination inter-enterprise markets. The software markets, where individual objects, classes, class libraries and frameworks could be purchased off the shelf, did not happen as some supporters of object-oriented solutions had promised.

All these problems have undermined the meaning of the object oriented paradigm in the developments of software engineering. Software researchers and practitioners have been steadily investigating solutions to the drawbacks mentioned and came up with the concepts of components.

2.3 Software Components

Component-oriented systems has received increasing research and industrial interest in recent years[128][123]. It promotes software reuse according to the black-box model. Components are independently developed pieces of application code that encapsulate their internal

implementation and whose functionality can be accessed only through well-defined interfaces like objects. The key point that distinguishes them from objects is that context dependencies of components are explicitly expressed. Thus, component developers may avoid some maintenance problems known from object-oriented systems, where undisciplined use of inheritance may reduce the encapsulation degree required for independent maintenance and modifiability (cf. fragile base class problem as described in [123]).

A component model defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application. One of the promises of component technology is a world in which customized business solutions can be assembled from a set of off-the-shelf business objects. Software vendors could produce numerous specialized business components, and organizations could select the appropriate components to match their business needs.

The behavior of components is specified in terms of interfaces, like it is done for objects. One way of specifying component interfaces is called contractual specification. The interfaces are specified as contracts. Such contract describes what the client needs to do in order to use the capabilities provided by a component. It also states the service or capability provided by the component. A contract captures two parties: the component offering some service, playing the role of a provider, and its environment, which plays the role of a client. The client has to meet the precondition before interacting with the component to comply to the contract. The provider can then rely on that precondition. On the other hand, the provider has to fulfill certain postcondition so that the interaction can be correctly completed. Finally, contracts can describe both the functional and non-functional requirements

Additionally, software components are also defined by their properties. The list of component properties as understood in this thesis is presented below:

- *Unit of Abstraction.* A component provides some functionality that can be accessed by means of a contract which consists of interfaces providing some services and interfaces that a component requires to provide its interfaces. A component usually hides its implementation, state and resources, which are needed to perform its tasks, from the component clients.
- *Unit of Composibility.* A component is a reusable software building block: a pre-built piece of encapsulated application code that can be combined with other components and with hand written code to rapidly produce a custom application.
- *Unit of Deployment.* A component can be deployed independently of other components with which it will be assembled in a system. It means that it can be packaged and delivered independently from other software components. Furthermore, it can be installed onto the nodes of the target infrastructure and it can be loaded into and prepared to execution in its execution environment (container).
- *Unit of System Management.* Each of the components can be individually managed and configured in terms of their quality of service, such as its service performance, provided security or fault tolerance.

The statements above are summed up in the following definition of the software component, which is then used throughout the text of this thesis:

Definition 1.

> *A software component is a piece of software with contractually specified interfaces and explicit context dependencies only. Context dependencies are specified by stating the required interfaces and the acceptable execution platform(s). Additionally, the component has the following properties: it is a unit of abstraction, composition, deployment and management.*

Software components are deployed in a execution environment to provide their functionalities specified using the interfaces. An execution environment of a software component is called container and typically is a single addressed space, where the components code is installed and executed.

Definition 2.

> *A container is an execution environment in which software components may be deployed and their code can be executed according to the underlying execution model.*

A containers is located on host that provides with computational resources, including the processing power and the memory as well communicational resources. The later are needed for components deployed in multiple containers to interact. In a distributed environment, a host is connected to other hosts through a data network, which allows for inter-host communication.

Once deployed, the code of software components is executed in a container. A software component being executed is also called to be instantiated and its runtime appearance, the result of instantiation, is called runtime instance. When referring to runtime instances of a software components in the context of runtime phase of a software system, the term component or component instance is used.

Runtime instances of components communicate with each other during the runtime. In case of components deployed in different containers, some remote communication means is used. Such (runtime instances of) software components are said to be distributed.

Definition 3.

> *Distributed software components are components that are deployed in containers located on multiple hosts, interconnected with a data network. Runtime instances of distributed components communicate with each other using a given remote communication paradigm.*

There are multiple communication paradigms that are supported by the existing distributed and networking operating systems as well as the middleware technologies. The communication paradigms and the technologies that support them are a topic of the next section. This kind of components is of primary interest of this thesis.

In this work, distributed components are implemented using object-oriented technology. From the implementation point of view, component runtime instances are then sets of co-located objects intercommunicating with each other and providing the contracted component services as described by the component contract. The technology supporting deploying and executing of such components is called middleware and is briefly described in the next section.

In the context of this thesis, a service is seen as a number of related software components and interacting with each other provide a value-added functionality to the users. This set of software components providing a well-defined functionality to a given user is called a service. Services will be considered mainly from the deployment point of view. When referring to a service deployment, it will be understood as deployment of all software components that form

a service.

2.4 Middleware

Middleware has been defined in many way in the literature [5][17][24][84]. From the communication perspective, the middleware can be seen a set of services that facilitate parts of a software system to communicate in a distributed environment. The middleware services are traditionally considered to reside *in the middle* above the operating system (OS) and networking software and below the distributed applications[5]. On the upper level, the middleware provides its services through a set of APIs to the applications. The applications can access the middleware services in a uniform way so that the distribution of applications and its parts is encapsulated. On the bottom layer, the middleware makes use of APIs offered by the underlying instances of the operating system residing on the nodes of the distributed system. A model of middleware from this perspective is given in Figure 1.

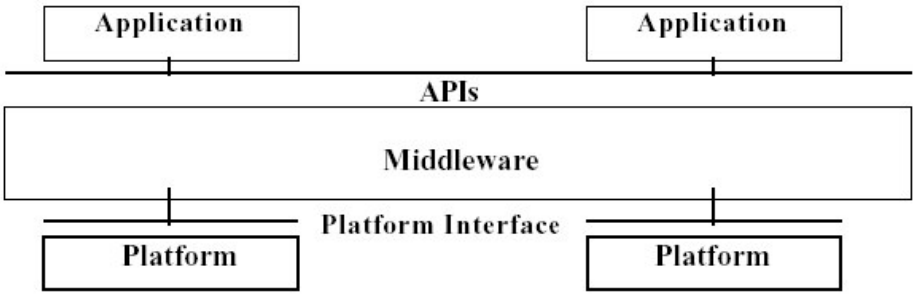


Figure 1. Middleware: Layer Model

2.4.1 Middleware Features

Middleware facilitates deploying and running applications in a distributed environment covering the following aspects of the complexities of distributed systems identified in section 2.1.1:

- *Generic Services.* Middleware can be used in the context of many various application. It offers so called horizontal services that are common to various application domains and independent of the specifics of a concrete application. The generic services include:
  - advanced communication services, like RPC, event-oriented communications, or group communication,
  - service for discovering services, like directory/naming or trading, and
  - coordinating interactions of services: transaction-processing support,
- *Heterogeneity support*. Middleware typically works on a number of hardware platforms and operating systems.
- *Distribution transparency.* Middleware allows for an exchange of information among distributed application parts without the application programmer having to handle intricate details of
- *Interoperability*, the ability of two or more systems or components to exchange information and to use the information that has been exchanged[44].

- *Portability*, which is the ability of two or more systems or components to exchange information and to use the information that has been exchanged[44].
- *Flexibility*, which is the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed[44].

The aforementioned features of middleware make it easily distinguishable from applications. The latter are designed to be specific to some domain and solving a concrete domain problem at hand. On the other hand, middleware can be distinguished from the operating system in that it does not provide any technology-specific services. The middleware API is supposed to encapsulate the idiosyncrasies of the underlying hardware and low-level software as well as provide a unified way of accessing communicational and computational resources in a heterogeneous environment.

2.4.2 Taxonomy Of Middleware

There is a range of different middleware systems that have been developed throughout the last decades. The approaches have been classified below according to the underlying computational and communicational model in [10]. The classification has been extended here by adding the class of component-oriented middleware and mobile agent middleware.

The following classes of middleware can be distinguished:

- *Message-oriented middleware,* MOM for short, is a middleware that typically supports asynchronous calls in the client and server architecture. Message queues provide temporary storage when the destination program is busy or not connected. MOM reduces the involvement of application developers with the *complexity* of the master-slave nature of the client/server mechanism. MOM. The messages can contain formatted data, requests for action, or both.

- *RPC-based middleware* is a middleware using the procedure-oriented paradigm. It allows for invoking procedures or functions on remote objects in the same way it is done in the local environment. This middleware is used seldom nowadays and most of solutions converted to object-oriented approaches. The most famous RPC-base middleware products are Distributed Computing Environment, DCE[95] standardized by Open Group and SunRPC[8], a proprietary solution from Sun Microsystems.

- *Distributed transaction processing*, DTP for short, is a middleware environment oriented toward handling transaction semantics over a network. It extends the MOM or RPC-oriented middleware by adding some transaction support. Some known products include Tuxedo or Encina.

- *Database access middleware* is a class of middleware that is specialized to support interoperable access to the data base products. An example of this kind of middleware is the Open Data Base Connectivity, ODBC standard defined by the Open Group (previously known as X/Open). It defines a programming-language and data base vendor independent API for accessing and processing data stored in the data base. Many of the data base vendors, including Oracle and Sybase, provide an implementation of this standard for their products.

- *Object-oriented middleware.* This class of middleware can be seen as an extension of the RPC-middleware. It allows for distributed parts of the application , which are modeled as distributed objects (cf. section 2.2) to exchange information in terms of object messages. The mainstream middleware technologies available on the market

currently are:

- *Common Object Request Broker Architecture,* version 1.0 to 2.4.3 [73] specified by the Object Management Group, an industry forum gathering companies and research organizations from the whole world. The CORBA is an open standard with many commercial, e.g. Borland's Visibroker[130], or open-source implementations, like OpenORB[82].
- *Java technology* defined and implemented originally by Sun Microsystems Inc. and further standardized by "an open, inclusive organization of active members and non-member public input" in the Java Community Process[120]. The specification includes among others, the Java language[118], Java Virtual Machine as a runtime environment, Java RMI[122] as the communication platform.
- *Component Object Model* and its most recent successor *.NET* platform provided by Microsoft Corp. COM[84] and its subsequent versions, DCOM and COM+, has been offered as part of the Microsoft Windows operating system since 1993. This technology mentioned for completeness but is only very roughly investigated in the thesis because of space and time limitation.

- *Component-oriented middleware.* This class of middleware has appeared relatively recently as the main stream of middleware products. Compared to object-oriented middleware, the basic unit of distribution, management and deployment is component whose semantics is explicitly defined in all these aspects of component life cycle. Component-oriented middleware supports, in particular, component assembling, i.e. combining 3rd party software components together at deployment time. The mainstream middleware technologies available on the market currently are:
  - *Enterprise Java Beans*[119] defines a component model for the Java technology. The model is based on Java Beans[117] published in 1996 and extended by support for different classes of components.
  - *CORBA 3.0* [73] is a component extension of the object-oriented CORBA 2.X middleware. It defines a fully-fledged component model based on the EJB approach and extended to the requirements of the heterogeneous multi-programming language interoperable distributed CORBA environment.
- *Mobile-Agent Technologies.* Mobile objects [50] and mobile agents are concepts that evolved from the object oriented approaches combined with work on mobility aspects of information systems. They support migration of running instances of programs including its computational state from one execution environment to another one so that the computations can be seamlessly continued at another location. The standardization forums concerned with mobile agents include FIPA[26]. A few examples of mobile agent platforms include Aglets[51], Voyager[71] and Grasshoper[3].

In the remaining chapters of this thesis, the term of middleware is equivalent to the object-oriented middleware and its component-oriented extensions, if appropriate.

2.4.3 Middleware-based Distributed System Architecture

The distributed system using object oriented middleware consists of the following elements:

- *Platform,* is the part of middleware which supports interactions between runtime representations of software components in a distributed system. The execution supports the ODP distribution transparency[46]. Because of the way the object-oriented middleware evolved towards the component-oriented middleware, the platform is further subdivided by some authors in the literature [18][48] into:
    - *Distributed Processing Environment*[126] supports interactions between software components using mechanisms to provide the ODP distribution transparency[46]. Examples of a DPE include OMG's CORBA-compliant Object Request Brokers or Remote Method Invocation implementation of the Java platform.
    - *Component-Support Platform*, which makes use of the Distributed Processing Environment and provides additional support for component deployment and execution. With respect to deployment support, it offers mechanisms to assemble components, install and configure them, whereas the execution support includes explicit factory component life cycle Some real world examples of a Component-Support Platform include *Enterprise Java Beans*[119] and *CORBA 3.0*[73].
- *Container* is an execution environment in which components are deployed and run providing their services to the service users.
- *Horizontal Support Services* are generic services used by the application components to access application domain independent functionality. Some example of these generic services have been listed in section 2.4.1. These services in the CORBA specification are called CORBA Object Services, COS, in Java they belong to the packages of the standard runtime environment.
- *Components* is software which provide some functionality specific to some problem domain. To provide the functionality, they use the capabilities of the platform described above. The components can be further classified with respect to the degree of their applicability. Two classes can be distinguished:
    - *Vertical Support Components*, called also *Service Platform Components*, are components that provide common capabilities and useful functions for a given problem domain. Some examples of such components are:
        - TINA Service Architecture[126] specified by the TINA Consortium which defines a set of generic services useful when developing of telecommunication services.
        - IBM's San Francisco[43] which provides a number of components for building E-Commerce applications.
    - *Application Components*. These are software components which encapsulate application-specific functions. They make use of the Vertical Support Components for the generic application domain functions, the Horizontal Support Services for the generic distributed system support services and the DPE for interactions with other system components.

The architecture of a distributed system based on component-oriented middleware is depicted in Figure 2. The sample distributed system contains two hosts running different operating system, shown as columns below the platform. The platform is presented as a even surface

hiding the heterogeneity of the underlying operating systems and hardware architectures. It provides a unified access to the communication and computational capabilities of the hosts. Containers with components running in them are located above the platform symbolizing their client role with respect to the platform. The components presented in the figure comprise both vertical support components and application components. The components can interact with each other by exchanging information in terms of operation calls, asynchronous messages (events) and data streams. These interactions are supported by the platform.

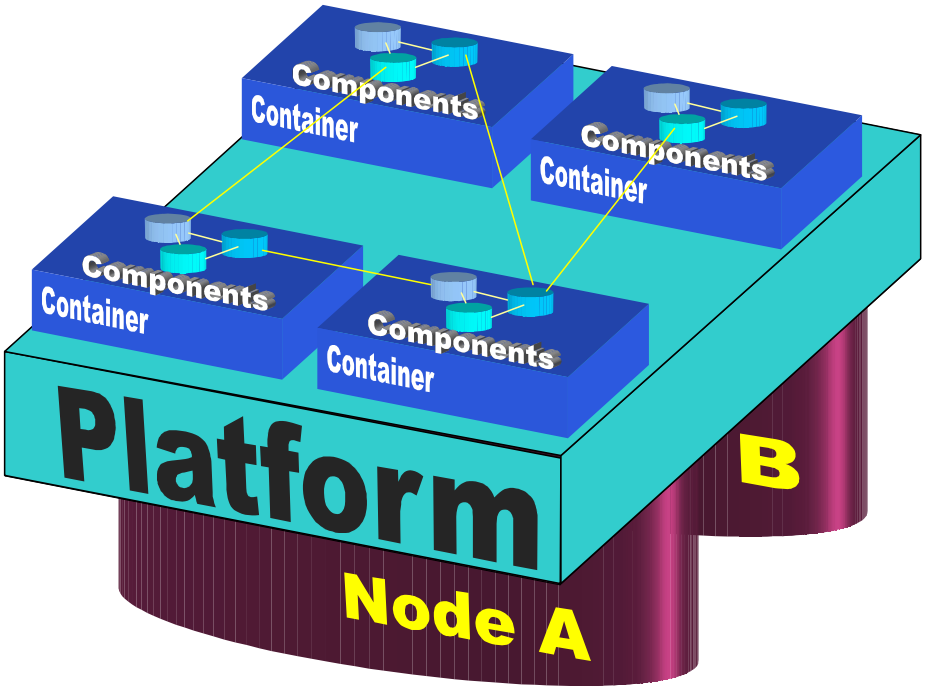


Figure 2. Reference Model for Component-based Middleware.

2.5 Summary

This chapter includes background information needed to understand the context of the core investigations of this thesis. The text refers to the fundamental properties of distributed systems and, in particular, dependable distributed systems and gives a very short concept overview of object-oriented and component-oriented software modeling and development paradigms. The chapter also presents introductory information on the middleware technology and its classification. The focus of this middleware presentation is put on object-oriented and component-oriented middleware platforms, which are technological target of investigations and are used as software development tools when building the prototype.

3 Dynamic Upgrades

This chapter introduces the terminology, basic concepts and problems related to upgrading component-oriented software and building systems supporting dynamic upgrades . First, a set of definitions of basic concepts, including the dynamic upgrade, are introduced in section 3.1. Some of these definitions are elaborated and concretized in the next chapters of the thesis. In section 3.2, it is argued that software components are a natural unit of upgrade. Section 3.3 follows with description of three key aspects of the problems related to performing dynamic upgrades in an distributed environment. The aspects include: system evolution, high availability and deployment and are a basis for defining a problem taxonomy for the dynamic upgrade presented in section 3.4. Section 3.5 motivates implementing the dynamic upgrade support functionality as part of a middleware platform. Finally, an overview of related research topics is given in section 3.5.

3.1 Basic Definitions

This software life cycle can be seen as is a software transition from a state that it is implemented and tested by the software provider to a state, in which it is usable in a given environment. This transition is enabled by releasing the software so that is available in a given environment or infrastructure and followed by a deployment process.

Definition 4.

Releasing software is a process of making a piece of software available for deployment in the given target infrastructure.

As the context of the thesis is distributed systems, the infrastructure is distributed *per se*. In a distributed environment, the infrastructure consists of multiple interconnected hosts, each of which is capable of providing computational and communicational resources to the components. The hosts may be heterogeneous in that they provide the software with different computational platform determined by the execution model and the access means to the host resources.

Definition 5.

Target infrastructure is a infrastructure where the given software is released. In a distributed environment, it consists of a set of hosts, each of which is capable of providing computational and communicational resources to the deployed software.

In general, software deployment is the a process in which software released are placed into the target execution environment and made ready for execution. The deployment process involves various activities, including code distribution, installation, configuration. Thus, the following definition of deployment process is coined for the purpose of this thesis:

Definition 6.

Deployment process is a process involving actions aimed at making a released piece of software ready for execution in a given target execution environment.

Whenever a piece of software is released it a target environment, it is available for deployment. This particular software released is called a software version or a software release and is identified by a version number, called also a release identifier, which differentiates one software version from another software version offered under the same software name provided by the same software provider.

Definition 7.

A software version is a piece of software available for deployment after it is released in the given target infrastructure.

Conventionally, a software version is called newer if it is released after another software version called older. Thus, newer or older attributes of a software version are related to the order of their release times. In the context of this thesis, a software version may additionally be new or old with respect to the upgrade process. The definition of these attributes are given below:

Definition 8.

A new software version refers to the software that is going to replace another software version during a given upgrade process. The other software version refers to a version deployed in the target infrastructure and is called an old software version.

In the process of software upgrade, a new software version is deployed into a target system and replaces the older version of the software code in the given target nodes belonging to the target infrastructure. The replacement or exchange of software versions is one of the key differences when compared to a deployment process as defined in Definition 6. Upgrading a software version means not only distributing, installing and configuring it in the target infrastructure but also configuring the runtime environment so that the new software is used instead of the old software version. In particular, whenever the code of the old software version is to be used, the new software version should be used if possible. Additionally, the upgrade process may optionally leave the old software version in some subset of the infrastructure intact, if there is a need for having multiple software versions deployed in the target infrastructure.

Definition 9.

A software upgrade, called further upgrade, is a process aimed at deploying a (new) software version so that it replaces another (old) software version which is deployed in a given subset of the target infrastructure.

Furthermore, some constraints are imposed on the differences between versions of the code. These constraints define the substitutability relation which determines the degree to which the new code can differ from the old code. Software is upgraded to improve the quality of service of a system or to adapt its functionality to the changing requirements. The substitutability constraints can be defined in a range of ways to cover the spectrum of allowed system changes. At this point, the definition of substitutability constraints is left open for further refinement as its definition strongly depends on the evolution model of the system, which is not in the focus of this thesis. Consequently, a skeleton definition of the substitutability constraints is stated below:

Definition 10.

Substitutability constraints define the allowed differences between the old an new software versions involved in an upgrade process.

To sum up, an upgrade process can be defined as a deployment process focused on exchange of software code so that the substitutability constraints on the code difference hold; this core idea is expressed symbolically in Definition 11.

Definition 11.

An upgrade process is valid if the new and the old software versions involved fulfill the substitutability constraints.

The *dynamic upgrade* is an upgrade process that additionally has a direct influence on the *runtime* behavior of a running software being upgraded. Not only does it deal with exchanging a software version deployed in the target infrastructure but also with exchanging running software pieces, called further *runtime artifacts*. Each runtime artifact is an instance of one software version deployed. An example of a runtime artifact is a component runtime instance as defined in section 2.3.

Definition 12.

A runtime (software) artifact is an abstraction of a piece of software being executed at runtime.

To exchange a runtime software artifact running one (old) software version means to replace it with another runtime artifact that is an instance of another (new) software version released in the target infrastructure. Like in an upgrade process, the old and new software versions of the involved runtime software artifacts have to fulfill the substitutability constraints for an dynamic upgrade to be valid.

Furthermore, the dynamic upgrade process has to ensure that the software system as a whole is operational during the upgrade so that the system user can access system functionality without breaks. These additional constraints set on the upgrade process are called runtime constraints. One example of the runtime constraints is dynamic upgrade transparency, which guarantees that the dynamic upgrade process does not influence the behavior of the runtime artifacts being upgraded software which is seen by other runtime software artifacts in the target infrastructure. Another example of runtime constraints is a requirements on the dynamic upgrade algorithm not to compromise the dependability of the distributed system, whose part is being upgraded. Taking into account these runtime constraints influences the design and the complexity of the algorithms to perform dynamic upgrades.

As the process of a dynamic upgrade deals with exchanging runtime artifacts and has an impact on the runtime behavior of the system both during and after the upgrade, it is of much more dynamic nature compared to the upgrade process. To stress this upgrade aspect, it is called *dynamic upgrade*. Definition 13 sums up the core idea of the dynamic upgrade.

Definition 13.

A dynamic upgrade is an upgrade process aimed at replacing given runtime software artifacts with ones running a new software version so that certain runtime constraints hold during the dynamic upgrade process.

Both the upgrade and dynamic upgrade process are performed to exchange some software artifacts (and their runtime instances in case of the dynamic upgrade). These software artifacts consist a part of the distributed system which the process is applied to and are called hereafter upgrade target.

Definition 14.

An upgrade target is the object of the given upgrade process. For a upgrade process it contains a set of deployed software artifacts to be replaced in the distributed system. For a dynamic upgrade process, it contains additionally a specification of runtime instances to be replaced in the distributed system.

The upgrade target may be compound, in general, as it may consist of a number of software artifacts. The smallest piece of a distributed system that can be an upgrade target is called a unit of upgrade.

Definition 15.

A unit of upgrade is the smallest software artifact that can be upgraded independently.

3.2 Component as Unit of Upgrade

In section 2.3, the component was defined by its properties. A component that has been developed independently from other components, may be then independently deployed and managed . As the application evolve and the components are modified by their developers, working possibly independently for multiple software houses, and deployed into the target system, the system is upgraded. If additionally, this upgrade process is required to by dynamic in the sense of Definition 13 in section 3.1, the component appears to be suitable as a unit of dynamic upgrade. In other words, its properties predispose a component to be a natural unit of upgrade. Therefore, it is argued that a component can have another property derived from its properties listed in section 2.3:

- *Unit of Dynamic Upgrade.* A software component is the smallest software artifact in a component-based distributed software system which can be an object of a dynamic upgrade process.

It is worth of noting that dynamic upgrade is not an orthogonal property compared to the previous ones. The activity of dynamic upgrading is a related to component deployment and (life-cycle) management. A dynamic upgrade is a special case of deployment as defined in section 3.1 and a special case of software life-cycle management, in which the deployed code of the component is replaced with a new version and, consequently, runtime instances of this software component are replaced with instances running this new code. Lastly, an upgrade is a result of component maintenance activity that deals with code changing due to debugging and adopting component code to new system requirements.

In the further investigation of this thesis, the component is considered as a unit of upgrade. All the definitions in section 3.1 related to dynamic upgrades abstracted from a concrete software artifact. From now on, the definitions are treated as if the abstraction of “software artifact” is replaced with a “software component“ and “runtime software artifact” with runtime instance of a component.

Note that this property postulates a component to be the smallest software artifact that can be independently upgraded in the system. This definition does not allow system upgrades of software artifacts of a finer level of granularity, such as a single programming language statement, a single class or a function being just a part of a software component.

On the other hand, this component property allows for dynamic upgrades having multiple software components as an upgrade target. These upgrades are called compound upgrades.

Definition 16.

A compound (dynamic) upgrade is a (dynamic) upgrade process with an upgrade target consisting of multiple software components.

3.3 Aspects of dynamic upgrades

The technology of dynamic upgrades can be considered from three viewpoints:

- *System Evolution.* Dynamic upgrades can be considered as a method to handle change management in evolving software systems at runtime.

High availability. Dynamic upgrades can be seen as a means to increase system availability by shortening the downtime during the maintenance breaks.

- *Software Deployment*. Dynamic upgrades can be viewed as a special case of service deployment, in that new version of some software has to be deployed replacing some other software already deployed in a system.

The subsequent subsections discuss each of the aspects.

3.3.1 Evolution

Dynamic Upgrades can be considered as an enabler technology for managing runtime changes in evolving software systems. Runtime change, in the context of distributed applications, includes the following aspects [42]:

- **Replacing or upgrading the application components** that define the system functionality. The change is further referred to as component upgrade and involves replacing one implementation of a system component with a new one. This aspect of the runtime change is elaborated in the rest of the paper.
- **Modifying the logical structure** (topology) of the system, which defines which components are in the system and how they are interconnected at runtime. Configuration changes are in terms of both adding and removing the system components and the connections between them.
- **Altering the physical structure** of the system, which defines how the system components are mapped onto the system underlying physical resources, like assignment of components to network nodes.

Dynamic upgrades can be considered as a special case of general change management support systems, in which:

- the logical structure of a software system is much more stable then the internals of the components,
- the scope of allowed changes is limited to replacing a number of software components so that each software component is replaced with a new version of the software component.

3.3.2 Dependability

Dynamic Upgrade can be considered a method to improve system availability [35], which is one of the characteristics of the system dependability[32]. Traditionally, the techniques for increasing availability have been based on masking hardware failures [32][115]. Software failures which are found to be more often sources of system unavailability have been addressed only recently. The formula characterizing system availability is often expressed as follows:

$$\frac{\text{MTTF}}{\text{MTTF+MTTR}}$$

Thus, availability can be enhanced by increasing the Mean Time To Failure, MTTF, or decreasing the Mean Time To Repair, MTTR. Dynamic Upgrade increases software availability by reducing the MTTR parameter of the system. A component that is malfunctioning can be replaced on the fly. Component is replaced without stopping the system

so the time to repair is much shorter then time to repair required to traditionally reinstall a part of the system. In this reasoning, it is assumed that the MTTF is not diminished hereby, that is, the system failures occur at most with the same frequency as before the upgrade. On the other hand, if upgrade contributes to improved system reliability as a new better and less buggy component replaces its older version, the system availability is enhanced even more.

Additionally, dynamic upgrade is a complementary mechanism to commonly used cost effective fault tolerant techniques applied to highly available systems. These techniques are based on software or temporal redundancy. The first ones comprise active replication, whereas the latter – a variety of passive replication schemes or rollback and restart. Since these methods mask only failures and do not deal with the origin of the failures, i.e. the faults, the faults remain in the system and will probably cause other failures in the future. In worst case, if the failure is not transient, the system will not successfully recover and these fault tolerant techniques will fail. Dynamic Upgrade alleviates this drawback by giving a support for removing faulty components.

3.3.3 Deployment

Software upgrades are part of the software life cycle. Dynamic upgrades, that is upgrades of running parts of a software system, are a special case of upgrades at runtime. This section explains the relationship of upgrades, and in particular dynamic upgrades, to other phases of software system life cycle.

In Figure 3, the full lifecycle of a software system is presented in a form of a graph. It comprises four main phases the *specification*, *development*, *deployment* and *runtime*. After a software system is specified according to the user requirements and with regard to results of the functional analysis, the software system is designed, validated and adapted to the specifics of a concrete middleware platform.

As a result of the development phase, the software is formed out of programming language constructs, called *artifacts*, like functions or classes and the so called *resources* containing required information to configure and initialize the programming constructs in the deployment phase. The programming constructs are divided into groups and packaged together in *software components*.

Then software component can be deployed into a target environment by delivering (or distributing) them and installing at their destination. If needed, the component is configured according to the deployment criteria of the local runtime environment and the preferences of the local system users. After that, the component is ready to activate. In conventional component lifecycle, the components have to be deactivated to be upgraded. Finally, if the old version of a component is not needed any more, it is withdrawn from the system.

Dynamic upgrade can also be considered as a special case of deployment. Upgrading software comprises deploying a new version of some service and optionally removing the old version if it is not needed any more. A dynamic upgrade support system can be then seen as an extension of a deployment system. To support software upgrades it uses the capabilities provided by the deployment system, like releasing new version of software in the distributed system, delivering software components to their destinations, allow for installation and uninstallation of software components. Additionally, some additional mechanisms are needed compared to a deployment system that does not support upgrades. These mechanisms include storing the state of the old version components and transferring it to the new ones. Even more challenging are dynamic upgrades that involve ensuring upgrade transparency.

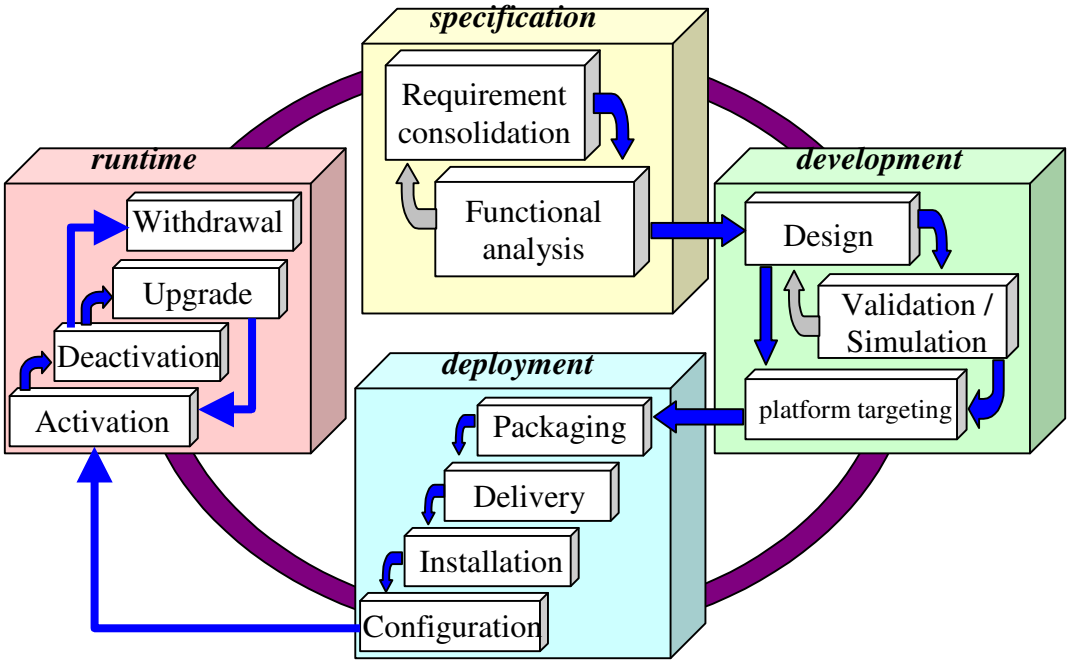


Figure 3. The phase model of software lifecycle.

In each phase, different methods, tools and technologies can be utilized in order to successfully and efficiently produce the desired output. For distributed systems, such technologies, methods and tools have been surveyed, explored or introduced by relevant research and development projects. It is not within the scope of this work to delve in all these phases and provide explicit descriptions for the applicable actions. Instead, the thesis focuses mostly on the component upgrade, particularly on its dynamic version, which belongs to the component runtime phase. It is worth noting, however, that investigating dynamic upgrade requirements on software technology, requirements may be expressed concerning other phases of the component lifecycle, like component development or deployment.

From the component developer perspective, the component's capability of transparent dynamic upgrade can be seen as a quality of service parameter of this component. This extra feature of the component has to be added to the component functionality. The upgrade transparency requires that the component developer be not burdened with caring about this component feature when developing their component. It means, however, that some development support tools have to be provided that automatically enhance the component implementation with some dynamic upgrade mechanisms. The component runtime environment, i.e. middleware platform may also be required to enhance.

3.4 Problem Taxonomy

This section describes the major issues that have to be addressed while dealing with upgrading software components systems on the fly. These issues are proposed to be classified in a hierarchical taxonomy. The taxonomy is based on the aspects of the dynamic upgrades introduced in section 3.3.

Figure 4 shows the proposed taxonomy. It consists of three trees, each of which classifying problems related to one of the dynamic upgrade aspects, including: evolution. dependability and deployment. The following subsections present the problems related to each of the aspects. Some of the problems are also symbolically illustrated in Figure 5.

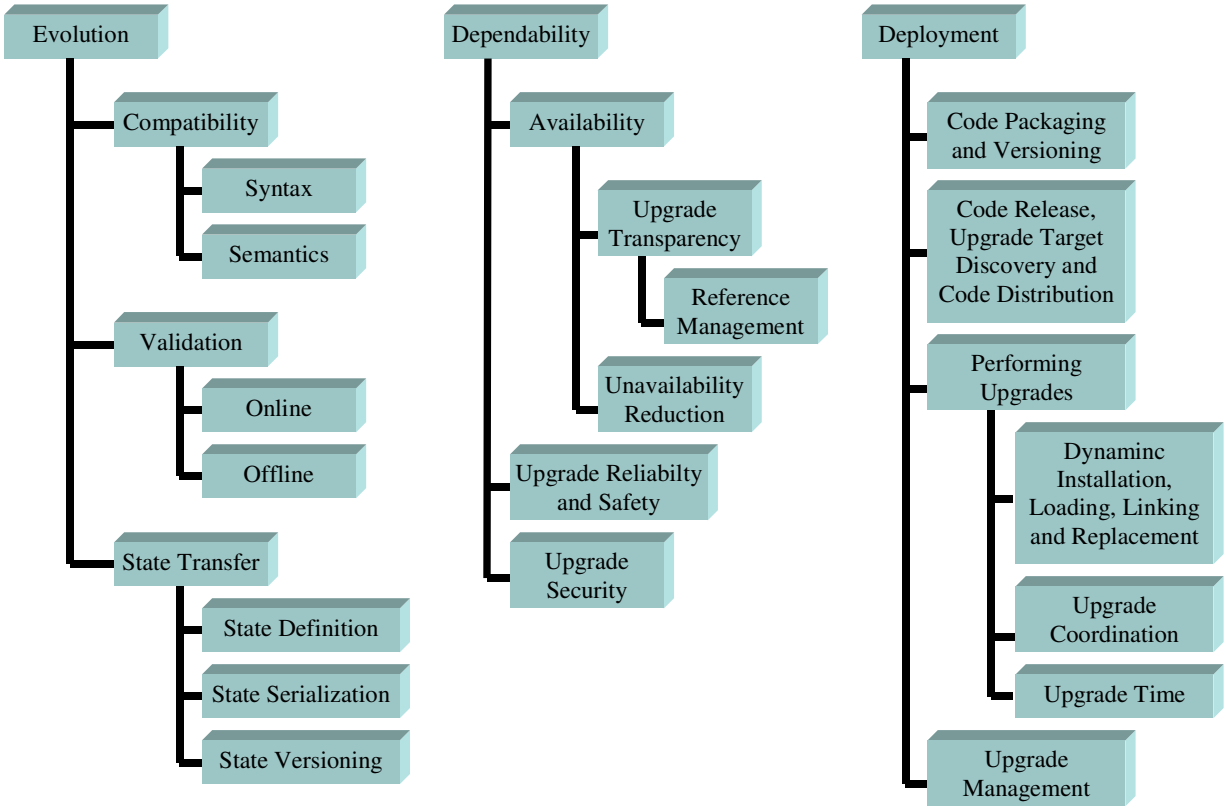


Figure 4. The taxonomy of the dynamic upgrade problems.

3.4.1 Evolution

The first tree of the taxonomy depicted in Figure 4 includes problems related to the aspect of system evolution, which was introduced in section 3.3.1. The related issues include system compatibility and its validation as well as state transfer.

3.4.1.1 Compatibility

One of the most important goals to meet when upgrading some running software is to preserve its consistency. Although the system consistency is described in terms specific to the functional model of the system, there are also some general issues on consistency preservation that are to be addressed in every component-based system.

Consistency preservation is a multi aspect issue. From the system evolution perspective, it can be investigated with respect to compatibility specification and upgrade validation. The first issue is concerned with specifying to what extent the versions of a component can differ from each other, i.e. what is the component substitutability [105]. Regarding a component as an entity with a number of well-defined interfaces that allow for accessing the component functionality and with some code implementing the functionality, the compatibility of implementations as well as interfaces is to be considered. Both of the concepts are related with the notion of subtyping [55].

3.4.1.2 Upgrade validation

Upgrade validation is concerned with checking that the upgrade is valid in the sense of Definition 11 and has terminated successfully so that the system consistency is preserved after the system is upgraded. This includes checks that the system upgraded behaves according to the expectations, that is the changes introduced do not make the system incompatible to its old version.

Two upgrade validation types can be distinguished:

- *Online validation* is performed during the upgrade process and immediately after the upgrade is done for some limited period of time. During this time, the new system version may be run in a testing mode in which it performs a possibly constrained set of its functions and the results of its computations are compared against the old version's results. If the checks are positive, the new version may be fully replace the old version which can be then deactivated. Otherwise, the upgrade is not valid and has to be rolled back so that the old version of the system can be resumed.
- *Offline validation* is done before the actual upgrade is carried out in a real system. The new version of the code is validated and tested in an environment simulating the real system. On a successful validation, the new version is considered compatible to the old version and the test upgrade is valid. The new version can be used to upgrade the target system.

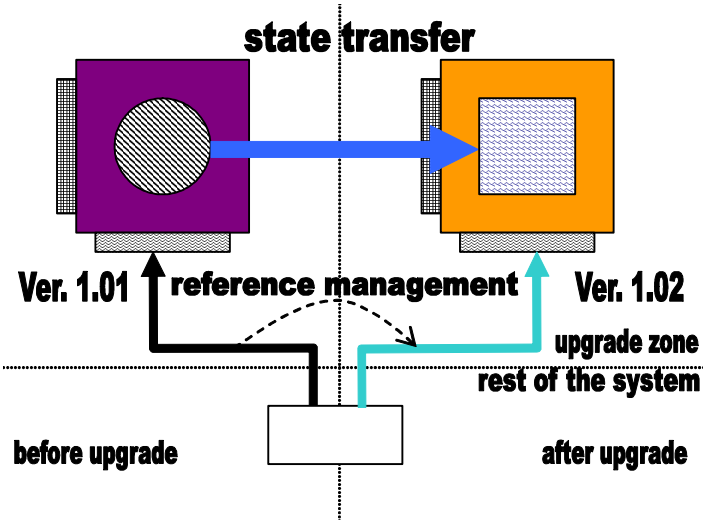


Figure 5. Component upgrade issues.

3.4.1.3 State transfer

Another problem that has to be solved regards component state transformation between the old and the new version of the component to upgrade. In order to preserve consistency of the system, the current state of the component to upgrade, i.e. its computation results and the whole context of the computations, have to be transferred to the new version of the component (see Figure 5). This means that the computational state of a component has to be stored somewhere temporarily and subsequently restored by the new component. In this way, the new version of the component can correctly carry on the computations from the point they were stopped for the upgrade break. A well-defined protocol is required for transferring the state at least from the old version of a component to the new one. Optionally, the protocol

may allow for transferring the component state from a newer version of the component back to the old one in case the old version has to be restored in the target system. An example of such a standardized protocol is GIOP[73] or the one supported by Java serialization mechanism[118].

Except for transferring the state between the component versions, an additional mechanism is needed that maps the state of the old component version to the new one. In other words, the destination version has to correctly interpret the state transferred and set itself into an equivalent state to the one transferred. In general this mapping cannot be performed automatically. Manual and semi-automatic mechanisms have been proposed in proprietary solutions [42] and [132]; no standardized mechanisms are available.

3.4.2 Dependability

The second tree found in the proposed taxonomy comprises problems related to the aspect of system dependability introduced in section 3.3.2. The problems identified are presented in the following subsections.

3.4.2.1 Availability

System availability is one of the characteristics of system dependability. It is defined by the time fraction in which the system is operational in a given period of time. The problems in this category are then related to minimizing the system unavailability during a dynamic upgrade process.

3.4.2.1.1 Upgrade transparency

Usually only some part of the whole distributed system has to be upgraded. To make the system operational during the upgrade process, this process in terms of unavailability of parts being upgraded should not be visible to the rest of the system. This property is called upgrade transparency and is related to issues of reference management and unavailability reduction explained in the subsequent sections. Some work that attempted to address this problem include [33] and [114].

3.4.2.1.1.1 Reference Management

A closely related issue is reference management. In our model, a system is made up of components interoperating with each other by means of connectors. The connectors are abstractions of lower-level constructs called bindings or references. To preserve the consistency of the whole system, an upgrade process is required not to affect the integrity of the component connections. On a lower abstraction level, it means that references that components use to send requests to other components must be upgraded respectively, as well. It is the upgrading policy that is decisive in how the references are managed. Assuming the upgrading policy permits only one (the newest) version of a component to exist in the upgrading zone, all the distributed references to the component to upgrade have to be rebound from the old to the new version of the component. Thus, after a component has been upgraded, all the references to the old version are now pointing at the newest component version so that the replacement of the component is transparent to the component clients. Other upgrading policies may require different reference management to keep the references between the system components consistent.

3.4.2.1.2 Unavailability reduction

One of the main reasons for dynamically upgrading systems is to reduce the unavailability of the system that would result from traditional maintenance breaks. During dynamic upgrades, special take care is taken to minimize the breaks in operation of the system component upgraded. One solution to this in systems without replication is to coordinate the upgrade process so that the upgraded parts of the system are as small as possible. In this way, at any time during the upgrade only a small part of the system is unavailable while others are operational. In distributed systems with replication support, the solution to this problem is based on using replication mechanisms to run both the old and the new versions during the upgrade in parallel and to perform the version switch instantaneously.

3.4.2.2 Upgrade Reliability and Safety

Distributed systems are characterized by inherent partial failures as described in section 2.1. For this a system supporting upgrades should be designed in a reliable and safe way. The safety of the upgrade process means the system ability of tolerating failures occurring in distributed systems (without running upgrade processes) as well as of recovering from new types of failures introduced by the upgrade itself. The latter ones may be caused by running new code that behaves in a different though compatible way. Some solutions to the upgrade reliability problem include *self-stabilizing upgrade algorithms* and a *recovery support* that enables a system to fall back to the state before the upgrade took place. As a consequence, the upgrade algorithm should be designed according to the fault tolerance principles ([6],[12] and [90]).

3.4.2.3 Upgrade Security

Upgrading of software at runtime is a critical activity. Malicious or wicked attempts to upgrade a running part of a system may have a disastrous consequences to the system. The upgrade support system should permit to perform upgrades to only authorized persons, playing the role of the system (upgrade) manager. Activities related to performing upgrades should be dealt with in a similar way like other deployment or maintenance activities. A special case is one a software component autonomously determines a need for an upgrade. The dynamic upgrade support system may have to determine whether the upgrade is permitted. The decision process can be determined by a rule engine prescribing which components and possibly under what additional conditions are allowed to instantiate an upgrade process.

3.4.3 Deployment

The last problem tree in the taxonomy presented in Figure 4 is concerned with problems of deploying the code in the distributed system. An overview of the related problems is the contents of the sections below.

3.4.3.1 Code Packaging and Versioning

The software components that are to be deployed have to be packaged prior to transfer and delivery to the target nodes. The package must contain the software components in terms of the code and a meta-data description of the system including its requirements and dependencies.

Packaging scheme depends on the component model. An example of packaging is that used for software downloaded via the Internet Explorer. The package consists of a compressed file (file.cab) which includes installation and registering instructions in an information file and an open software description file [129] specifying dependency data (e.g. DLL files) and the software component(s). A proposal of the new CORBA Deployment and Packaging Model, a part of the OMG's CCM specification[73], and its proposed extensions [20] defines another packaging scheme. The CORBA component model provides the concept of the 'descriptor' that is based on extended XML elements to describe a packaged software system. Yet another packaging method is described in the FAIN documentation on the Active Service Provisioning[23]. The descriptor is defined using a XML schema and describes the characteristics, including dependencies, of the associated service component.

Additionally, versions of the component implementation have to be distinguishable in order to perform a system upgrade. The mechanisms supporting this is called versioning. Versioning of components is usually done assigning a version number to each component version released by the software provider. There are a number of versioning schemes applied in different development and deployment environments. The version numbering schemes can be as simple as monotonously increasing integer numbers whenever a new component version is released[74]. A component version number can also be defined as a n-tuple of integer numbers, each of which is increased with regard to a different aspect of the system that has been changed in the new software version[94]. More complex versioning schemes are based on description of component set-valued features to model relationships between component versions sets[133]. In any case, version numbering is related to the software release and should be orthogonal to other aspects of component, like naming of runtime instances of the component or naming scheme for the internal software artifacts comprising the component implementation. Additionally, because the interface and code of a component implementation can evolve independently of each other, these two versioning mechanisms are frequently separated.

3.4.3.2 Code Release, Upgrade Target Discovery and Code Distribution

A system upgrade is possible as soon as the new upgraded code is available. The process of making the code available to the system is called code release. The code release may be as simple as notifying the system manager, or the software running on behalf of him, of the available code and providing the information on how to access the code. Code release may also be followed by an automatic code download so that it is available locally on the nodes belonging to the distributed system. For the downloading system to work efficiently, it is necessary to discover where to download the code to. This requires discovering of the so called upgrade target, which consists of running instances of the code to be upgraded.

3.4.3.3 Performing Upgrades

The very core of deploying upgraded code in the target system are the issues related to performing the upgrade process. These issues are presented in the following subsections.

3.4.3.3.1 Dynamic Installation, Loading, Linking and Replacement

To be able to upgrade a piece of running code in a software system, the execution platform (for instance an operating system or a runtime environment like Java Virtual Machine) has to support installation of new code at runtime. The installation comprises system management actions at the nodes of the distributed system aiming at preparing the code to be executable in

its target instance of the execution platform (typically an operating system process). Example installation steps may include unpacking the component packages, copying the code into the suitable directories, updating the appropriate system variable and registry services to make the execution platform aware of the newly installed code modules. Not only has the code to be installed but it has to be loaded into a running instance of an execution environment (dynamic loading) and linked with other code already running on the platform. Finally, the runtime artifacts running the old version of code to be upgraded have to be replaced with corresponding artifacts running new code.

3.4.3.3.2 Upgrade Coordination

Performing an upgrade depends much on what to is be upgraded, that is what is the upgrade target is. Some examples of upgrade targets include a single running instance of a component, a set of runtime instances implementing a component, like an actively-replicated server, or all runtime instances of a set of components. Obviously, the running code instances may be interacting with each other and upgrading all of them at once may reduce the system availability. For handling upgrades of multiple targets, which are potentially running on a number of hosts in the system, the upgrades have to be coordinated. The coordination mechanism computes a sequence in which the runtime artifacts are to be upgraded, enforces this sequence and takes care of additional requirements on the upgrade process, including transactional property.

3.4.3.3.3 Upgrade Time

Another germane matter that needs investigating is focused on when a component can be upgraded. An ideal case would be if a component could be upgraded directly after an upgrade request is sent. However, this is not always the case. Let us imagine a component running a method's code as a consequence of a client request. Intuitively, the component should not be stopped at any moment of the request execution but the upgrade should be postponed to the time when the component is ready, e.g. when the request has exited and the component is inactive again. One approach to the problem has been proposed in [49]. A component is only allowed to be upgraded when it is in a quiescent state, i.e. the component is passive and has no outstanding transactions which it must accept and service.

3.4.3.4 Upgrade Management

In a large distributed system, the complexity of software including the dependencies between components, the distribution scheme for software deployment is fairly high. Furthermore, the system management and supervision may be also done in a distributed way, possibly by multiple system administrators. To correctly and reliably upgrade such a system, a automated support is needed. The main issues in the upgrade management concern:

- coordination of multiple upgrade processes and resolution of conflicts that may occur if upgrades are triggered independently.
- support for upgrade planning and enforcing these plans so that the upgrades are performed efficiently from the perspective of the whole distributed system.

3.5 Upgrade support location

Considering the issues that have to be dealt with when supporting dynamic upgrades of distributed applications, on one hand, and the capabilities that the different elements of the distributed systems as presented in section 2.4.3 on the other hand, this thesis argues that the

platform is the most suitable element in the middleware-based distributed system architecture for localizing the functionalities supporting dynamic upgrades. The following rationale supports this claim:

- *Dynamic upgrade is a horizontal support mechanism.* As dynamic upgrades are special case of deployment mechanism, which is then one of the generic domain-independent services, it is straightforward to design a facility for supporting dynamic upgrades of distributed applications in the platform. As a dynamic upgrade support facility has to cover a number of issues described in section 3.4, which are related to many aspects of the platform, the facility may have to extend both the DPE as well as the component support platform.
- *Dynamic Upgrade transparency.* Upgrading a component on the fly is required to occur transparently to other application components. This transparency may be considered as another distribution transparency as defined in the ODP terminology. It is a inherent feature of middleware platforms to implement these transparencies.

The middleware support for dynamic upgrades is further called Deployment and Upgrade Facility. The Facility, the platform and a set of tools supporting development of dynamically upgradable software components will be further called a dynamic upgrade infrastructure. Figure 6 shows a middleware-based architecture of a distributed system extended by the support for dynamic upgrades. It focuses on the elements of the dynamic upgrade infrastructure.

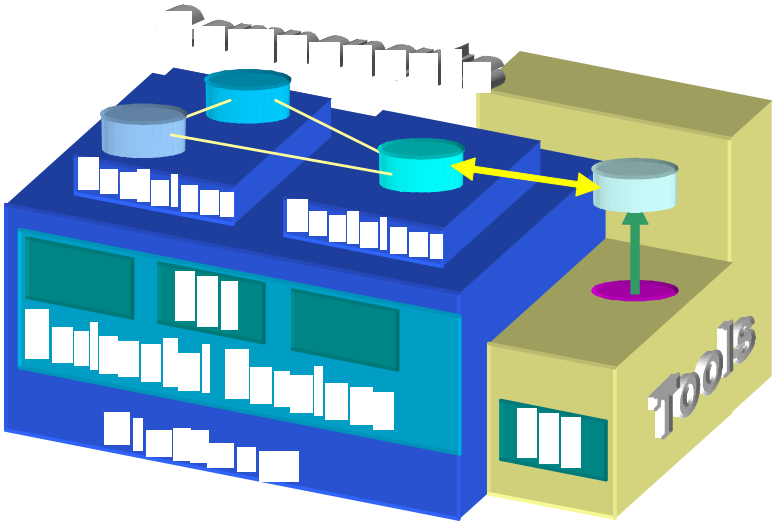


Figure 6. The infrastructure to support dynamic upgrading.

The infrastructure elements are described as follows:

- **Deployment and Upgrade Facility (DUF)** provides basic mechanisms enabling component replacement at system runtime. The mechanisms are embedded in a concrete platform and use other services of the platform, like dynamic loading of components, component state storing/restoring or inter-component reference management.
- **Platform** is based on the existing middleware platforms for distributed systems, like Microsoft DCOM[85], Java RMI[122], OMG ORB[73] or TINA DPE[126]. Extended

with the DUF facility as one of the horizontal services, the platform forms a complete execution environment for distributed software components capable of being dynamically upgraded. The platform is the key dependency of the implementation of the model developed in this thesis but is not part of the result of this thesis.

- **Deployment and Upgrade Tools (DUT)** form a development support environment for the software developers and providers. The tools support the programmer in developing deployable and dynamically upgradable components. Their aim is to minimize the effort of programming software components with dynamic upgrade capability by semi-automatic generation of code related to deployment and dynamic upgrade. Development of the tools, though, is not part of this thesis.

3.6 Related topics

Dynamic upgrade has been a research topic for a long time now. It was most frequently discussed in the context of runtime change management systems, such as CONIC[49] or POLYTH[41][42]. Upgrading a system without stopping it was also an relevant issue in the context of high available and dependable systems [32] in investigation of methods to reduce the planned system downtime [66]. Some aspects of dynamic upgrade problem was addressed in the research on migration and mobile code. In the next subsections, the problem of dynamic upgrade from the viewpoint of the related research areas is briefly presented.

3.6.1 Fault tolerance

Dynamic Upgrade binds two aspects of software systems: evolution and availability [32]. The latter is also a goal of fault tolerant systems. Because of some similarities in concepts and their realization techniques, the techniques can be used complementary in many critical systems. This section relates the notions of dynamic upgradability and fault tolerance and focuses on the points distinguishing the techniques.

The principle objective of the dynamic upgrade is to support the evolution of software systems with a requirement that the system's availability is maximized. Systems made up of components can be modified by replacing the components with their new versions. Instead of applying the changes to the system by shutting the system down, the components are replaced on-line, so that the maximal availability of the system's functionality is reached. The changes are applied so that both the scope of the system that has to be stopped and the time of the system maintenance are minimized.

Dynamic upgrades as a method to cope with software evolution running fault-tolerant middleware have been considered in our work [109][110] and [17]. Below the main differences between Dynamic Upgrade Support Systems and Fault Tolerance Systems are sketched.

Fault tolerant systems are designed with the objective to cope with, and recover from, different sorts of failures of the system and its environment, and additionally, to ensure system's high availability. The failures in the context of fault tolerant systems [115] traditionally refer to a loss of processing resources or a loss of logical and physical communication paths. However, if component upgrades are also considered as system failures, then software systems that are dynamically upgradable can be regarded as a special case of fault tolerant systems. A temporary loss of processing resources is inherent to an upgrade of a component since a component has to be stopped and their new version started after the upgrade. Additionally, the new version of the component has to be consistently initialized with the current state of the whole system in that the stored state of the old version

of the component is used to compute the initial state of the new version of the component. Analogously, if a failure of a processing resource, such as an operating system process, happens in a fault tolerant system, it is handled by restarting another instance of the process and initializing it with the state of the old instance of the process that has recently been checkpointed This comparison allows for finding further analogous like upgrade request – failure, upgrade – recovery, dynamic upgrade facility – recovery mechanism, component state saving – checkpointing.

Two significant differences between dynamic upgrading and fault tolerance have to be mentioned, however. Firstly, the initialization time of an upgrade can be controlled, whereas real failures occur unexpectedly and have to be handled immediately in most cases. In spite of the asynchronous nature of the upgrade requests, the start of the upgrade execution can be put off until the system is ready to handle the upgrade since a relatively short postponement of the upgrade is usually acceptable. On the other hand, critical failures have to be dealt with immediately in fault tolerant systems. Secondly, component upgrades always involve two versions of the component: the one to be upgraded and the one that will replace the running version. In fault tolerant systems, on the other hand, the recovery mechanisms deal with only one version of the component affected by the failure. Thus, recovering from a system failure typically requires restarting the same version of the component that has crashed for whatever reasons. Upgrades have an additional dimension when compared to conventional component recovery in fault-tolerant systems: the component evolves during the upgrade. This needs an upgrade facility to have appropriate mechanisms that will correctly insert the new type of the component into the slot in the system where the old version of the component has been executing. This can comprise: validating the conformity of the new version of the component with the old one, transferring and mapping the component state from the old to the new version.

In [134], the maintenance events have been classified with regard to their impact on the system availability. Class 1 describes Non-recoverable Maintenance events, that is events requiring the system to be unavailable for the entire duration of the maintenance activity; Class 2 is thought to be for Recoverable Maintenance events, which need the system to be unavailable during part of the maintenance activity; Class 3 is concerned with Transparent Maintenance, meaning events that do not require a system outrage. The technique of dynamic upgrades can be then viewed as a method to eliminate events of class 1 and 2 from the system life cycle.

3.6.2 Code Mobility and Mobile Agents

Systems supporting code mobility that has been investigated and developed in the last years address similar issues that are dealt with when upgrading software on the fly. This section describes the similarities and the differences in the dynamic upgrade and mobile code problematic.

Migration has been a research topic heavily investigated since the late 1970s in computer science. Process migration systems, like Condor[9], Sprite[13] or TUI[106], and object migration systems like Emerald[116] enable process or objects, correspondingly, to move between different execution environments at runtime. Execution environments are usually located on different hosts connected by a network. The key problem that has to be solved to allow for code migration at runtime is to transfer the current state of a process or an object from the original machine to the destination. Whereas the process state consists of all the variables within the process, the object state means a set of internal variables encapsulated by the object implementation forming just a part of the runtime execution environment or

process. The problem of state transferal is also central to upgrade systems; e.g. a new version of a component has to continue the component's computation from the state the old version of the component had just before the upgrade. In both classes of systems the common point is that a component state has to be prepared for transfer somehow, and translated back to a form that can be used for component for further component computations after an upgrade or migration process is over.

The details of the transfer process are however different. State transfer in migration systems is concerned with the remoteness of the original and destination execution environments, whereas state transfer in dynamic upgrade systems is a means to deal with different versions of the component implementation and its state definitions. In a migration system, the state is prepared for transfer by being marshaled so that it can be sent on the wire to a remote destination address space. After the state has been transported to the destination it can then be demarshaled. If a heterogeneous distributed system is involved, a migration system has to define a well-defined state exchange format that is independent of the hardware and underlying operating system so that the object state running in the original execution environment can be correctly interpreted in the destination execution environment. On the other hand, in an upgrade system, the state transfer involves mapping the state of the old version of the component to an equivalent state of the new version of the component. To sum up, the state transfer in upgrade systems deals primarily with differences in the component architecture itself, whereas the state transfer in migration systems deals with differences in component execution environment.

The mobile code concepts have been further elaborated in the mobile agents systems, like KQML[25], General Magic's Odyssey [29], Open Space's Voyager [71] or IKV++ Grasshopper [3]. To sum up the concept of a mobile agent, a short description of mobile agent technology after [93] is cited hereafter: *"mobile agents are agents that can physically travel across a network, and perform tasks on machines, called agencies, that provide agent hosting capability. This makes processes possible to migrate from computer to computer, for processes to split into multiple instances that execute on different machines, and to return to their point of origin. Unlike remote procedure calls, where a process invokes procedures of a remote host, process migration allows executable code to travel and interact with databases, file systems, information services and other agents"*.

Despite many similarities of dynamic upgrade and mobility support in agent systems, the application areas of these two approaches differ. On one hand, applications that need to be upgraded on the fly have typically strong availability requirements, i.e. even a short planned maintenance break would unacceptably decrease the system availability. Such systems are usually designed in a way that does not permit changes of the system architecture but enables for predicting the time characteristics of the mission critical system components. On the other hand, typical application areas for mobile agent technology do not have such high availability requirements but allow for building flexible systems that can be reconfigured and extended. To upgrade such an agent-based system, it is enough to send the new version of the agent software to a corresponding agency and upgrade the agent off-line.

3.7 Summary

This chapter presented an introduction to the dynamic upgrade topic. After the basic concepts are concisely defined in section 3.1, three aspects of the dynamic upgrade problem are presented in section 3.3: system evolution, high availability and a software deployment. These aspects serve as a starting point when classifying problems related to dynamic upgrade in the problem taxonomy 3.4. The problems identified in the taxonomy are further analyzed and

serve as input to form a set criteria for the comparative study of the state-of-the-art Dynamic Upgrade Support Systems (chapter 4), or DUSS for short. The deployment aspect is stressed in the further chapters of this thesis and the dynamic upgrade is seen as a special case of the deployment process.

In this chapter, it is claimed that the most suitable element of the distributed system architecture for localizing the dynamic upgrade support is the middleware platform (section 3.5). Consequently, in the subsequent chapters of this thesis, middleware platforms are investigated with regard to the support for dynamic upgrades and extensions of these

Additionally, the software component is proposed to be the basic unit of dynamic upgrade in the component-based distributed systems. It is suitable to be a unit of upgrade because of its other properties as pointed out in section 3.2.

Chapter completes with a short discussion with related research topics in section 3.5

4 State of the art

This chapter describes the state of the art in support for dynamic upgrades both in the main stream middleware products and in research prototypes extending the standard programming environments for distributed systems. These systems are analyzed and evaluated in terms of a set of comparison criteria. The analysis and evaluation of the state-of-the-art systems help to identify their drawbacks and missing features.

4.1 Introduction

The problematic of managing runtime changes in distributed systems has been heavily investigated. One trend of this research is focused on formal describing software architectures which has resulted in plethora of *Architecture Description Languages, ADL.* The system architectures are often described in terms of system components, connectors and their designed configurations (e.g. Darwin [57]) or planned configuration changes (e.g. Rapide [56]). A newer class of these formalisms are *Architecture Modification Languages*, which are extensions to ADLs and provide means to describe unplanned changes to architecture of a system, see C2's AML [83].

Much work that has been done combines the problem of describing the changes of the software architecture with the dynamic upgrade of architectural components. The theoretical foundations of a distributed component system capable of being dynamically changed have been developed during work on Regis/CONIC project [49]. Some more technical results have been presented in papers on POLYLITH [41], a framework for dynamic reconfiguration of distributed systems. This approach is based on the formalisms elaborated in the CONIC project and focuses on introducing runtime change at the granularity of an operating system process. Research on CONIC and related projects concerns upgrading only the so-called *quiescent* (not active) components, i.e., informally, which are being not involved in any communication with either other components or their environment.

Other category of research done on upgrades is focused on the low-level details of the development environment including the programming language and the operating system that the solution is developed for. These approaches propose reengineering the development tools, including compilers and assemblers and tackle issues regarding techniques for dynamic code loading and linking [53][87][39], defining the state of upgraded software, transferring it to the new version [2][7][21][31][39][51][132] and reference management [33].

Some research that is focused only on dynamic upgrade aspects of runtime change management can also be found in literature. Different design and programming artifacts at various levels of abstraction and of different granularity have been investigated to be a unit of system upgrade, see Figure 7. As an example, Gupta et. al [34][35][36] consider introducing changes to software systems at the level of an imperative language statement. A procedure implementation as an upgrade unit is used in DAS [31] and PODUS [102]. Fabry [21] and Bloom [7] (in his work on ARGUS extensions) investigates software replacement on the level of program modules. Other systems, like C2 [83], DCUP [89] support dynamic upgrading of components at the architectural level. Still another approach is represented by the DRASTIC [16] project in which upgrading software is examined on the level of autonomous sub-domains, called zones, which form a distributed system. A comparative survey of some of the systems supporting dynamic upgrade can be found in [103].

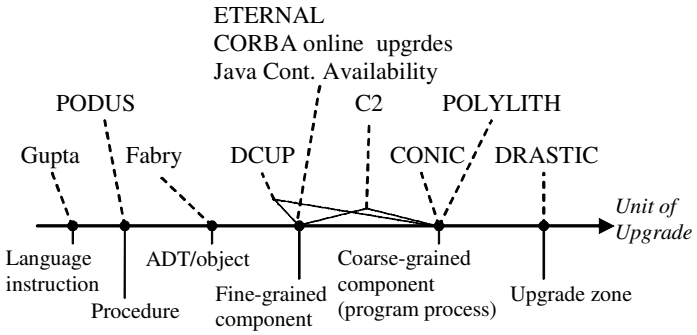


Figure 7. Classification of different approaches with respect to the granularity of the units of upgrade.

4.2 Investigated Systems

The investigated systems include both standard middleware platforms and their propriety extensions specialized to provide support for dynamic upgrades.

Two standard middleware are investigated in this chapter: CORBA (sec. 4.2.1) and Java (sec. 4.2.2) technology. These are the main stream technologies used both in the industry and research community. Additionally, this work on this thesis has been carried out in the context of projects aiming at developing distributed systems for purposes of telecommunication applications, which these two technologies have been mostly applied in.

A number of systems enabling dynamic upgrade is presented in more detail, to provide the reader with better understanding of the dynamic upgrade issues better. Because of the focus of this thesis, the survey is limited to systems that :

provide a pure software solution, i.e. do not require on special hardware support,

support dynamic upgrade of distributed applications.

The subsequent sections describe the proprietary dynamic upgrade support systems. They include: C2 (sec. 4.2.3), CONIC (sec. 4.2.4), DRASTIC (sec. 4.2.5), Eternal (sec.4.2.6), PODUS (sec.4.2.7), POLYLITH (sec. 4.2.8), SOFA/DCUP(sec. 4.2.9), STL (sec. 4.2.10) and CHORUS (sec. 4.2.11).

4.2.1 CORBA

Common Object Request Broker Architecture is an open standard for distributed object computing, defined by the Object Management Group, OMG. The current version of the CORBA specification at the time of writing this text is 3.0 adopted by OMG and published in a formal document [73]. Further references to CORBA include: [72][84][85][97].

The Object Management Group has been active in the area of upgrading CORBA systems at runtime for some time.[77][76]. The standard defines a first portable and interoperable framework to facilitate upgrading of CORBA objects. The framework:

- allows for upgrading of implementations of individual CORBA objects only, i.e., external object interfaces cannot be upgraded.
- provides a means to ensure objects-to-be-upgraded to reach a quiescent state, whereas a quiescent state is defined as not executing any methods.
- defines interfaces to enable state transfer of CORBA objects.

- ensures upgrade transparency in terms of:
    - *consistent message delivery* during the upgrade process, i.e. the messages are not lost, disordered or processed twice,
    - *referential integrity* in that clients continue using the same object reference after the upgrade they used before the object has been upgraded.
- provides a capability to roll back an upgrade before the new implementation becomes operational.
- allows for reverting from the upgrade in a given period of time.
- provides some basic transactional support for upgrades of object collections.
- supports upgrades of both replicated and non-replicated CORBA objects can be upgraded.

To sum up, the current release of the CORBA standard does not support dynamic upgrades. The OMG is currently working on a future CORBA extension adding rudimentary support for dynamic upgrades. The main drawbacks of this future extensions are:

- Limitations in the dynamics of change, i.e. in how far the new version of code can change from the old version. Only implementation changes are supported.
- Support for upgrades of simple upgrade targets. The specification does not enable to upgrade a number of CORBA objects during one upgrade process.
- Weak upgrade management. Even though the specification supports two kinds of predefined upgrade initiation schemes, the push and pull upgrades[1], no other upgrade management schemes are considered.
- Inflexibility of specification. The specification proposes one upgrade algorithm and defines interfaces and an interaction protocol to support this algorithm. Other algorithms are not supported and the specification does not address framework extensibility, such as a possibility to add new upgrade algorithms.
- The specification deals basically only with runtime aspects of upgrades. It does not addresses issues of deploying new versions including managing the code, such as installation of new code, uninstallation and discarding the old one. It assumes that a new version is available at a given node and is deployed so that instances of the implementation can be created on demand.
- Weak support for versioning and evolution tracking. The specification does not address how to relate different versions of a object implementation. There are no mechanisms that support storing the information about the old and upgraded code.

4.2.2 Java

Java [1] is one of the most frequently used object oriented programming languages. It was released by Sun in 1995 and quickly has become very popular in the area of Internet applications thanks to its applets, i.e. Java programs that can be dynamically downloaded in the compiled form (as byte code) either from a local file system or through a network e.g. from a web page, allowing code mobility. The code mobility offered by Java is limited to one hop, i.e. sending the code from a web server to the clients. The code is executed in the Java Virtual Machine, JVM, in the so called sand-box model to minimize security problems of the

[1] The push upgrade is initiated by a client of the upgrade framework who triggers an upgrade request. The pull upgrade is initiated by one of the application objects to upgrade.

foreign code running at the client side.

Java platform supports distributed computing by the remote method invocation (RMI) service an the serialization service. The first allows for sending and processing requests by remote objects. The latter allows for serializing a graph of Java objects into byte stream and deserializing objects from a byte stream.

Another important service that Java offers it reflection. This service allows for inspection of programming artifacts, like classes or packages for their features (class methods or package classes), instantiating objects, accessing or modifying the values of fields of objects or classes, and invocation of methods on objects and classes.

Java Beans is a component architecture for the Java platform. The architecture allows for defining beans, which a components consisting of a set of Java classes and resources. The specification distinguishes two phases of a component life cycle.

- Design time, when beans are customized by modifying their attributes, connected to other beans and assembled into compound beans.
- Runtime, when the beans can be used, i.e. when Java programs or other beans invoke operations on the beans, the beans send or receive events.

Depending on the phase of the bean lifecycle, the behavior or appearance of a bean may be different.

Java with its JVM, distributed object model (RMI) and a component model Enterprise Java Beans can be considered as a middleware technology. It offers a similar functionality as CORBA or DCOM but is limited to only one programming language, Java. Because of JVM availability for most mainstream platforms, the Java middleware can be used in heterogeneous systems.

In the context of the support for dynamic upgrade of software components, the following features of the Java language and its environment are relevant:

- Java serialization. With this technology, a Java component can save their state and send it to a new version of the component.
- Class loading [53]. Java allows for loading new classes to be loaded into the JVM at runtime. This functionality is similar to dynamic libraries known from DLLs in Microsoft Windows or Unix shared libraries and supports the following features:
    - *lazy loading,* i.e. classes can be loaded on demand at run time. class loading is delayed till the class is to be used.
    - *type-safe linkage,* i.e. both link-time and runtime type checks are applied to the loaded code to guarantee type safety.
    - *user-defined class loading policy*, code is loaded by the so-called class loaders so that the loading process may be fully controlled by the programmer.
    - *multiple namespaces,* i.e. class loaders provide separate namespaces for different software components, and thus class of the same name may be treated as separate types in the same JVM.

  This feature can be used to load new version of software during runtime to upgrade components in the system.

Product Versioning. To cope with evolving software, Java allows for versioning of specification and implementations. In Java, the following entities have their versions: the Java Virtual Machine both in terms of the language specification as well as of the implementation , Java Runtime and the core classes, and Java packages. Java provides

reflection mechanisms that allow to query Java programming artifacts, like packages or classes for their characteristics. Among others, the Java reflection API provides information about Java entities versioning.

Java Packages are the basic unit of Java software in terms of independent development, packaging, verification, upgrade and distribution.

The Java Language[118] specifies the notion of binary compatibility that corresponds to the general notion of component substitutability. The specification states that a change to a type is binary compatible with (equivalently, does not break binary compatibility with) preexisting binaries if preexisting binaries that previously linked without error will continue to link without error. Binaries are compiled to rely on the accessible members and constructors of other classes and interfaces. To preserve binary compatibility, a class or interface should treat its accessible members and constructors, their existence and behavior, as a contract with its users. A list of some important binary compatible changes that the Java programming language supports is presented below:

Reimplementing existing methods, constructors, and initializers to improve performance.

Changing methods or constructors to return values on inputs for which they previously either threw exceptions that normally should not

occur or failed by going into an infinite loop or causing a deadlock.

Adding new fields, methods, or constructors to an existing class or interface.

Deleting private fields, methods, or constructors of a class.

When an entire package is upgraded, deleting default (package-only) access fields, methods, or constructors of classes and interfaces in the package.

Reordering the fields, methods, or constructors in an existing type declaration.

Moving a method upward in the class hierarchy.

Reordering the list of direct superinterfaces of a class or interface.

Inserting new class or interface types in the type hierarchy.

The Java programming language is designed to prevent additions to contracts and accidental name collisions from breaking binary compatibility.

In the JCP community there is ongoing activity on continuous availability support for Java applications running on top of the J2EE platform[121][65]. The work is aimed at defining standard APIs for availability-related support transparent to application components and includes online upgrades as one of the mechanism considered in addition to support for Field-Replaceable Units (FRU). The focus of the online upgrade work is put on collaboration between the J2EE platform and the application during the online upgrade process. The authors propose an EJB, Enterprise Java Bean to be an upgrade unit and define a two-phase-commit upgrade algorithm with a rollback support. A rollback of an upgrade process may be triggered by the operator before an upgrade is committing. The specification defines an interface that upgradable objects have to implement for starting and committing an upgrade process on a bean class. The interface is used by the J2EE server, which provides an execution environment for beans, to control the upgrade run. Upgrades are triggered by the Operator or other components of the J2EE Platform. State transfer is controlled by an auxiliary platform component, the Resource Manager, and is based on the Java serialization mechanism. The old version of software modules, the EJB classes, are removed (unloaded) from the execution environment after an upgrade terminates successfully.

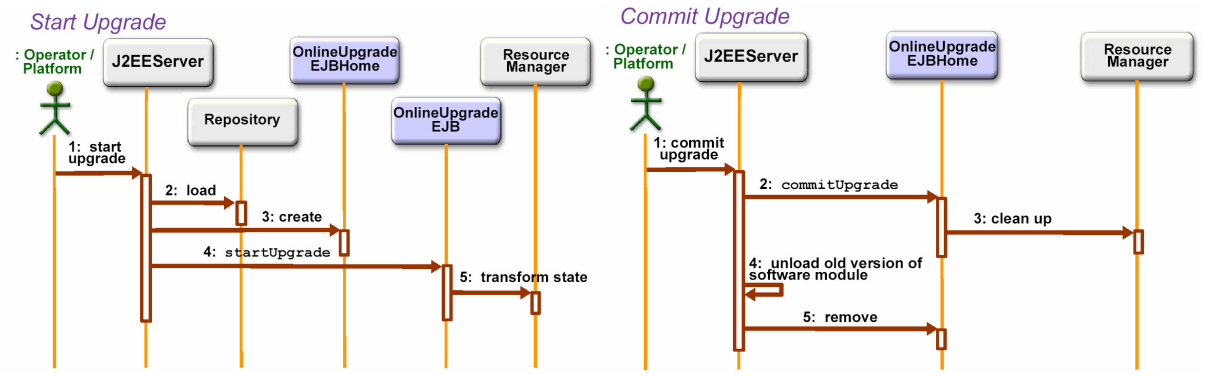


Figure 8. Sequence diagrams showing a two-phase dynamic upgrade algorithm in the proposed J2EE solution.

4.2.3 C2

Oreizy et al. [83] presents an approach to runtime software evolution on the architectural level. They investigate software systems having the so called C2 architecture. Such systems consist of components and connectors that are explicit architectural entities. Components communicate via connectors so that component interdependencies are minimized and computation is separated from communication. The key benefit of such an approach is:

- high abstraction level of runtime change handling, making it possible to make decisions based on an understanding of application requirements and its semantics;
- separation of decisions concerning change from application-specific behavior.

They distinguish four aspects of change management:

- application policy defines how the change is applied, such as instant or gradual component replacement,
- scope is the extent to which the system is affected by the change,
- separation of concerns regarding system functional behavior and runtime change
- level of abstraction at which changes are described.

It is investigated how dynamic changes to system at run time, like adding, replacing or removing system components, and reconfiguring connectors binding can be managed so that the system integrity is preserved.

The authors also discuss what requirements components and connectors should fulfill to support runtime change. Components must be dynamically loadable and able to rebind to connectors. Connectors which encapsulate components interactions and localize decisions regarding communication policy and mechanism, must be implemented as discrete entities so that component binding reconfiguration is possible. They support different aspects of change management: they can provide various change policies, facilitate to localize change

4.2.4 CONIC

Kramer's and Magee's [49][57] work on dynamic change management is focused on distributed application structure changes like component addition, deletion and reconfiguration of component connections. Their approach, which is based on a separation of issues regarding the application's architecture from those regarding application components' functionality, allows specifying and introducing structural changes so that the application state

does not need to be considered to preserve the consistency of the application during and after the system upgrade.

Their results regard a model of a distributed system presented in their paper. A distributed system is a directed graph of processing nodes bound with each other by means of directed communication paths starting in the connection initiator. Each node can initiate and service transactions that model an exchange of information between nodes. The transactions concern always only two nodes and consist of one or more message exchanges between the connected nodes. The model assumes that the transactions complete in bounded time and their initiators are aware of the transaction completion. The consistency of the system is determined on the global and local levels. On the global level, some constrains in terms of system node states have to be preserved. Nodes are said to be locally consistent if there are no partially complete transactions at the node.

The central property an application node must have so that the system consistency could be preserved during change, is identified and defined as quiescence. A node is *quiescent* if:

- It is not currently engaged in a transaction that it initiated,
- It will not initiate new transactions,
- It is not currently engaged in servicing a transaction, and
- No transactions have been or will be initiated by other nodes which require service from this node,

This property ensures that the node's state is consistent (does not contain the results of partially completed transactions) and frozen (the application state will not change as a result of new transactions.

A model for dynamic change management has been prototyped in CONIC. It is based on general structural rules for making alterations at the configuration level without the need to consider the application state nor the specification of component actions. The changes can be applied in a way as to leave the modified system in a consistent state and without disturbing the unaffected part of the operational system (Figure 9).

The model tries to meet the following objectives:

1) Changes should be specified in terms of the system structure.
2) Change specifications should be declarative. That means that the needed changes are separated from how they are carried out.
3) Change specifications should be independent of the algorithms, protocols and states of the application.
4) Changes should leave the system in a consistent state.
5) Changes should minimize the disruption to the application system. This refers to the fact that the management system should, from the change specification, be able to determine a minimal set of nodes which will be affected by the change, leaving the rest of the system to continue its execution normally.

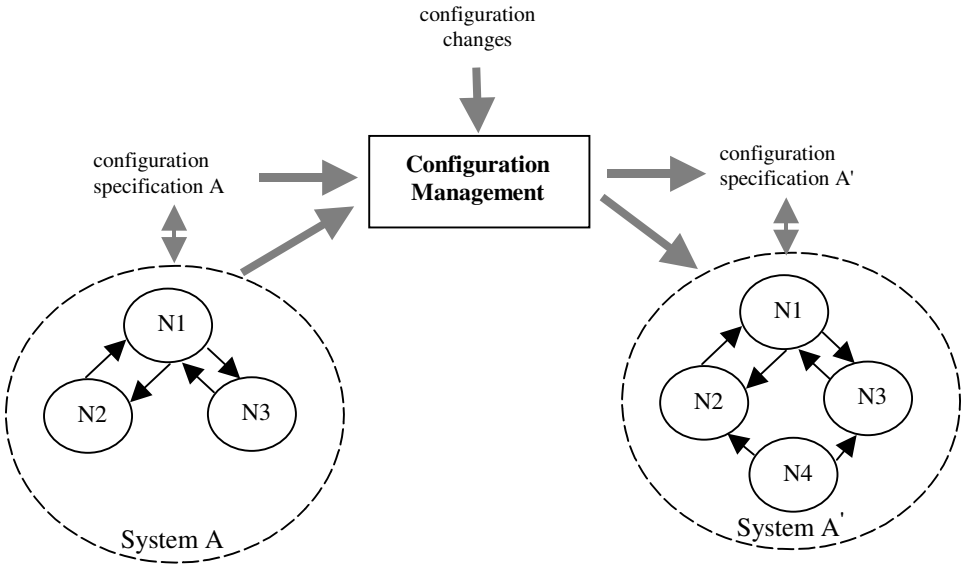


Figure 9. Configuration management in CONIC

In order to maintain a consistent application state during dynamic changes, the management system should interface with the application in order to direct it towards the appropriate state of reconfiguration. Furthermore, the management system must be able to confirm that the application has reached this state. The configuration states along with their transitions are outlined in Figure 10.

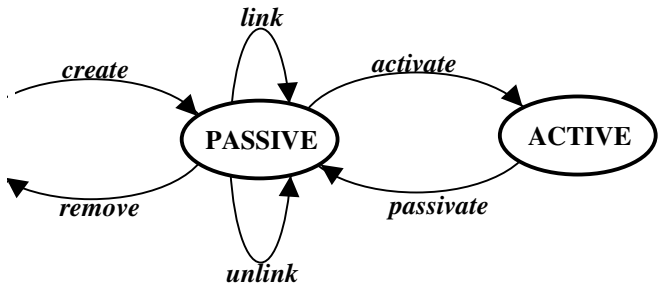


Figure 10. State transition diagram of a CONIC application

A component in a passive state can continue to accept and service transactions but it must not initiate any new transactions except as a result of accepting or serving transactions. The precondition for either linking or unlinking is that the component in question must be in quiescent state which signifies that all components that can initiate transaction on the target component are passive.

4.2.5 DRASTIC

The DRASTIC [15] project is aimed to build an architecture which explicitly addresses run-time change evolution of heterogeneous distributed systems. The architecture should allow distributed systems for changing component types and their implementations, as well as the system configuration. DRASTIC systems are organized in a number of disjoint autonomous zones that are logical collections of processes distributed over a network. The processes

consist of interoperating objects. A zone is the finest unit of evolution in that all the objects within a zone are considered to evolve as a single unit. A zone is also a unit of change encapsulation i.e. it provides some means to constrain propagation of the change so that other parts of the system that have been not directly affected by the change do not have to be adapted to the new system configuration. These constrains are explicitly expressed with contracts that are defined as pair-wise agreements between zones. The contracts describe which types can be transferred between the zones and the necessary transformation of the types. In the DRASTIC architecture contracts have to be provided by the software developer.

As a proof-of-concept of the DRASTIC concepts, a series of prototype systems is built in various environments. The CORBA specification is investigated as one environment. The project research results in an observation that the CORBA specification lacks support for system evolution. Firstly, CORBA does not provide for tracking type changes. An CORBA object is accessed by its clients through its interface. Interface definitions are kept in interface repositories. Every object interface is given a unique identifier, so called repository id, which is used to look up the interface in an interface repository. When passing an object (reference) from one ORB to another, it is required in CORBA that the repository id be the same in both ORBs. However, every new version of an interface is considered to be other interface and there is no explicit support in CORBA to relate the different versions of an interface. DRASTIC comes up with a contract concept to solve the problem. Secondly, the CORBA specification does not provide a means of constraining the effects of evolution. Operations on an interface repository, like adding a new version of an interface definition, are visible throughout the whole system, permitting other parts of the system to see an incoherent repository. Thus, a means of localizing the system change is needed so that the evolution will be made transparent and changes will not propagated to other parts of the system. To constrain the effects of the evolution, DRASTIC provides zones that are the units of decomposition and evolution. The software in each zone is evolved as a coherent whole, largely autonomously from software in other zones, even though the source code may originally have been shared by components in many zones.

Contracts specify which program types can be exchanged between zones and the transformations that are required should an object move from one zone to another or if a method invocation is made that crosses a zone boundary. Zone autonomy is supported by inserting code supplied by the software engineer at the zone boundary. These software fragments, called change absorbers, handle the transformations indicated above and enforce the zone contract.

The zone contract is a pair-wise argument between two distinct zones. A contract between any two zones, for example, A and B is described in two sub-contracts which consist of three descriptions each (Figure 11). Sub-contracts are seen from the perspective of the exporting zone. Zone A 's sub-contract specifies: which types it will export to zone B; which types zone B will import from zone A; and the transformations required to transform objects from one type to another. The second sub-contract describes the same information from zone B's point of view, although it is not necessarily the inverse of zone A's.

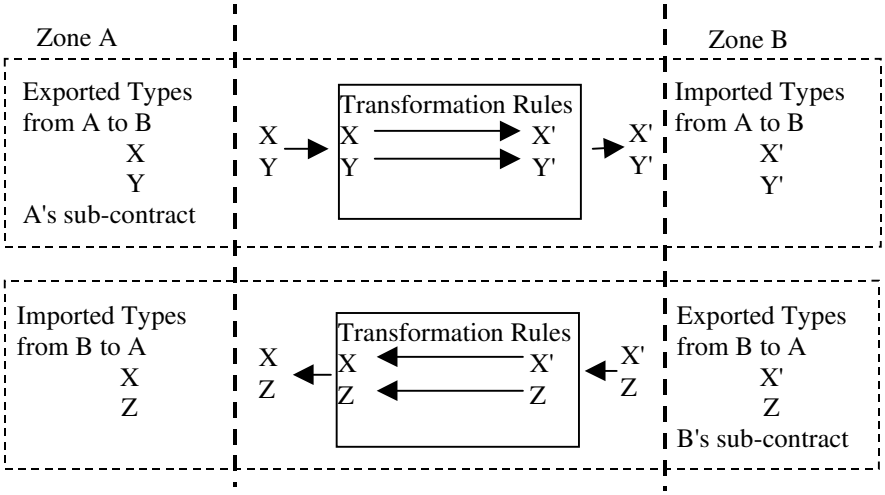


Figure 11. A Contract between Zone A and Zone B

Zone B exports objects of type X' which zone A imports as objects of type X. Only the software that performs the transformation from objects of type X to objects of type X' needs to be aware of this relationship. As type transformation rules can be specified between any two zones, it is possible for one zone to export a type that another is not prepared to directly import, for example, X' and X above. This is the reason why one zone's export and import sub-contracts may not necessarily be the inverse of the other zone's.

DRASTIC allows processes in one zone for their evolution while other processes in other zones can continue their execution, even when these processes hold references to each other. This is done by intercepting application-level object references and placing change absorbers along the reference chain between the invoking object and the object being invoked. In the current DRASTIC design, all software and data in a zone must be evolved at the same time. The objects have a unique identity within the entire system and they can migrate. Persistency is supported using the concept of orthogonal persistence, i.e. the object persistency is added independent of the implementation of the object.

To sum up the DRASTIC approach to system upgrading:

- It does not support a fine granularity evolution. All the components in a whole zone have to be frozen at evolution time.
- Freezing a zone means terminating all the processes in the zone and storing their state on a disk. This is inefficient and cannot be applied to mission-critical systems with real-time constrains.
- The problem of upgrading continuously active components within a zone is not addressed in the project. The programmer is compelled to solve the problem in ad hoc way.

4.2.6 Eternal

Eternal [68][69] extends CORBA with capabilities for fault tolerance based on object replication. It supports both active and passive replication schemes of CORBA objects both playing the role of clients and/or servers.

The more recent work on Eternal extensions [66][124] provides the CORBA application developers with support for *life upgrades*, i.e. software and hardware upgrades on the fly. The target of upgrade is a replicated CORBA object. The upgrade process is incremental so that

the replicas are upgrades one by one while the application is operational. The upgrade process is divided into the following steps:

- *Preparation phase,* in which the new version of the source code is analyzed and compared against its old version. The preparation is aimed to automate the process of upgrading the system. The system with the assistance from the programmer, generates some code to facilitate the upgrade. The code is then compiled and deposited in the CORBA implementation repository.
- *Upgrade phase,* in which the old executable code running is indeed upgraded with it new version. Whereas the upgrade is seen from the application level as a single atomic action, it consists of multiple upgrades of single replicas being upgraded. The upgrade supporting system ensures that at least one replica of an object is operational throughout the upgrade.

The system does not automate upgrading distributed systems fully. Assistance from the application programmer is needed to support state mapping between the old and new code, for instance, when new attributes consisting the object implementation are added.

4.2.7 PODUS

The PODUS [102][103] system, developed by Frieder and Segal at University of Michigan and at Bellcore, represents a procedure-oriented approach, as classified in [103], to dynamic program upgrading.

Programs in PODUS system are distributed programs written in a procedural languages, like C or Pascal, that use remote procedure calls for interprocess communications. An upgrade of a program is done by replacing each procedure of the program with its new version. Replacing a procedure involves changing the binding form the procedure's old version to the new one.

The key features of the PODUS upgrading system are:

- The upgrade policy allows replacing many procedures as an atomic program upgrade. The order of the individual procedure upgrades, as well as the time of the upgrade is automatically computed so that the correctness of the program can be preserved during the upgrade.
- The upgrading systems determines automatically time when each procedure may be upgraded by analyzing syntactic[2] and semantic[3] dependencies between procedures and inspecting the program runtime state to check weather the procedure is active. Only inactive procedures can be upgraded.
- The PODUS imposes two constrains on programs to be upgraded: they must be structured in a top-down manner and all data must be accessed by means of abstract data types.
- It supports changes to procedure signature (interface), internal data structures and implementation code. The procedure interface change is possible by introducing a level of indirection: the user defines so called *interprocedures*, subroutines converting the procedure calls between different versions and *mapper procedures* converting data formats between procedure versions.
- It supports upgrading of distributed programs. The programs are limited, though.

---

[2] A procedure P depends syntactically of another procedure if P can be called from the other procedure.

[3] Procedures P and Q are semantically dependent if they interact but their dependency cannot be determined on the basis of program code syntax.

- Program components communicate by means of RPC and are single-threaded processes,
- semantically dependent procedures are collocated. This constrain enables to avoid considering the dependencies between distributed procedures during the upgrade.
- It supports existence of multiple versions of the same procedure in a program.

4.2.8 POLYLITH

Hofmeister and Purtilo [41][42] presents an extension to POLYLITH, a software interconnection system, that allows management of runtime change in distributed programs without lost of overall system service. Their research is based on results of Kramer and Magee [49] work on Conic with regard to when a distributed program can be reconfigured and in what way. The concerns investigated by Hofmeister focus mainly on externalization, internalization and possible transmission of the process state. Their method can also support process migration and fault tolerance.

Applications in the POLYLITH environment are composed of interoperating components that are system processes. Each process is implemented by a module that includes individual data and program units. Components communicate with each other by binding to module interfaces of other components and thus building communication channels. The structure of the application is determined by its modules and channels binding them.

The following forms of runtime change are distinguished:

- module implementation, replacing an individual module,
- system structure, like adding or removing a process, and
- geometry that defines mapping system parts to physical hosts.

The approach to reconfigure a system is characterized by:

- A component is defined to be a system process that is capable to externalize and internalize its state in terms of abstract data types (ADTs). This capability is application dependent and adequate functions supporting the capability must be supported for each component by the programmer. The components are responsible for capturing and externalizing the whole their state, including the piece of state cached by the underlying operating system, like table of open file descriptors.
- The components have to communicate with each other by means of the POLYLITH bus so that all the communication can be controlled during the reconfiguration process.
- Component upgrade involves atomic rebinding that does not cause the messages in transit sent to the upgraded component to be lost. The neighbor components (that communicate with the components) do not have to be halted during most of the upgrade period because communications channels can buffer messages.
- Component addition, a special kind of structure reconfiguration, requires the component to be resynchronized with the current state of application so that application consistency is preserved. This resynchronization is application dependent. Changing a system part requires freezing the contiguous system elements.

POLYLITH (see [41] and [42]) is a component architecture enabling components implemented in different languages executing on different platforms to communicate across a network. Components (called *nodes* in POLYLITH) may communicate using asynchronous messages or RPC. Both clients and servers must explicitly declare interfaces (*modules* in POLYLITH) with the operations (interfaces in POLYLITH) to use for communication.

Figure 12 shows the POLYLITH MIL (Module Interconnect Language) definitions for a simple application with a server performing integer additions for a client. Observe that the client and server interfaces are bound statically.

```
service "math_server" : {
  implementation: { binary : "/user/x/maths.exe"
            machine : "apollo.widgets.com"}
  function "add" : {integer, integer} returns {integer}
}


service "math_client" : {
  implementation: { binary : "/user/x/mathc.exe"
            machine : "zeus.widgets.com"}
  client "add" : {integer, integer} accepts {integer}
}


orchestrate "math_program"{
  tool "math_server"
  tool "math_client"
  bind "math_client add" "math_server add"
}
```

Figure 12. Polylith MIL definitions for a simple application.

To allow dynamic upgrade of POLYLITH nodes, mechanisms for rerouting communication and transferring node state has been added (cf. reference management in section 3.4.2.1.1.1).

For a *stateless* node, the upgrade process would be to rebind all (external) interfaces bound to the old module to the new module. POLYLITH reconfiguration primitives permits this change to be applied atomically, and in such a way that no messages in transit are lost.

Transferring the state to the new node involves transferring node data and the stack of program counters. Node data is serialized and deserialized by code supplied by the programmer. The programmer must also supply a set of *reconfiguration points* (equivalent to the upgrade points as defined in section 3.1 of this thesis). When a state transfer is requested, execution continues until a reconfiguration point is reached. At that point, local variables (i.e. data in the current stack activation frame) are serialized and sent to the new node along with the identity of the reconfiguration point, and then the function returns. This process continues until all stack frames have been unwound. After restoring global data, the new node will call each function that should be on the stack in the right order, restore the local data and jump to the point where execution was suspended. The state of the old node has then been reconstructed, and execution resumes.

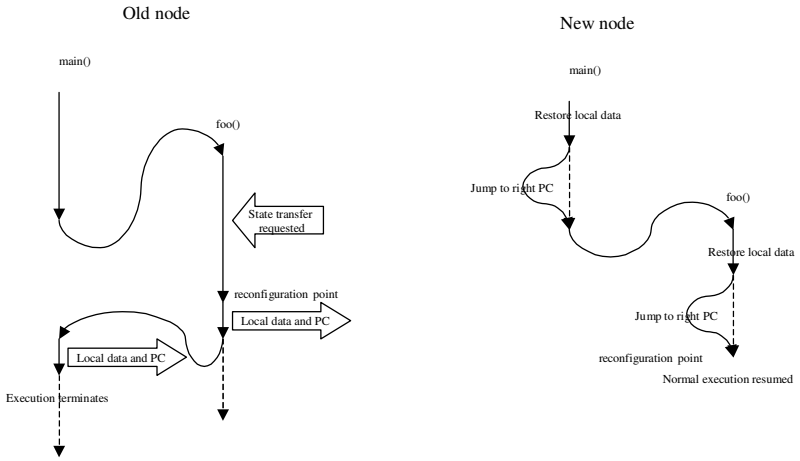


Figure 13. Transferring stack frames.

Figure 13 illustrates how the stack is transferred to the new node. The PC is *not* a physical program counter (memory address), but rather the identity of a label to which the new code does a jump.

The process for unwinding and rebuilding the stack is aided by a special pre-processor (as of yet only available for C), which inserts code that handles everything but the serialization and deserialization of data. Since the stack management is written in high level language and manipulates the stack using normal function calls and returns, it is portable.

4.2.9 SOFA/DCUP

SOFA (Software Appliances)[88][89] is a platform, developed at the Charles University in Czech Republic, which supports off-the-shelf software components to be provided over a network and run applications made of these components. It provides a set of abstractions to model trading with software components over a computer network including the SOFA component model. In particular, the platform addresses issues related to component searching, downloading, upgrading, licensing and billing.

Software components are distributed to clients by means of the network of SOFA nodes. The SOFA network supports a secure and reliable component propagation medium that is independent of the underlying component provider's or client's network.

DCUP, Dynamic Component Upgrading, is an extension to SOFA that allows safe upgrading of software components at runtime. The architecture is implemented in Java environment. An application in DCUP consists of components that form a tree-like structure. A component, which is a group of objects, is composed of a permanent and a replaceable part. The permanent part exists in the application across the versions of a component and provides a indirection layer encapsulating the replaceable part of a component. The replaceable part is changed each time a component upgrade takes place and contains the objects that define the functionality of the component. The permanent part is responsible for component building, upgrading and terminating, as well as for transition of references crossing component boundaries. Wrappers, that belong to the permanent part of the component, mediate access to the target objects exported by a component and prevent from accessing the component during an upgrade. Thus, the upgrade of a component is transparent to other components. The

wrappers also check activity of a component allowing only for an upgrade of inactive components. The DCUP addresses also transition of state from the old to the new version of a component in that every component must be serializable.

To sum up, DCUP does not support upgrade transparency to component developers. Every component has to implement certain interfaces that are needed for the upgrade support system. The current implementation is available for the Java environment and a CORBA port is planned. The approach taken in DCUP is limited as it allows for upgrading components that are inactive only.

4.2.10 STL

STL [132] is a tool supporting state transfer during an online software upgrade in a distributed environment. The tool is able to checkpoint an application state and transfer this state between two versions of the application processes even though the state format is different. The application's state is defined in terms of data structures and can be transferred so that the definitions of data structures in the new version of the application process are different from the definitions used in the old version of the application process.

The proposed mechanism is claimed to be portable and machine independent as the tool uses a data format based on the External Data Representation, XDR, for transferring the state. The format is a standard data representation in remote-procedure call, RPC, mechanism to transfer data between different hardware architectures. XDR is enhanced to transfer complex data structures, like double links, cyclic graphs and other cross referenced data structures.

The following changes in the data structure are allowed by STL:

- **Removal of fields**, when the new version removes some data fields from the old version.
- **Addition of fields**, when the new version adds a few data fields to the old version.
- **Field size change**, when the new version changes the size of existing fields like arrays.

The application state has to be explicitly determined by the application developer by identifying the data structures contained in the state. The identified data structures and their modified versions are collected in a file. The modifications are manually marked by the programmer. Once the specification of the application state for two versions of the software is given, a corresponding procedures for marshalling and demarshalling are generated by the STL tools. Instances of data structures are stored independently from each other. In more complex cases, where the data structures include cross-references, some support from the application programmer is needed to determine which instance of data structures should be stored together.

4.2.11 CHORUS extensions

Hauptmann et al. describes in their article [37] an extension to the Chorus operating system allowing for dynamic upgrade of applications. Their approach does not require modifications to development tools but focuses on adjusting the application code to make it replaceable. The authors provide recommendations so that this adjustment can be supported by development tools. Their results are not general enough and apply to only one operating system, namely Chorus, and one programming language, C++ because dynamic upgrading, or on-the-fly software replacement in their terms, cannot be realized in an operating system- and language-independent way.

4.2.12 Other approaches

Peterson et al. [87] presents a method for dynamically upgrading running code in higher-ordered and typed ("HOT") programming languages. In their work on an Haskell compiler implementation, called Hugs, they mainly investigate issues that are relevant to dynamic upgrade from the language perspective. The first-class entity and the basic unit of upgrade is a higher-ordered function. Functions are defined in program modules which permit control over namespaces, facilitate separate compilation and can be loaded on demand into a running program. In the paper, a strategy of principled dynamic code improvement is investigated which permits to constrain the options to dynamically upgrade the program code. Only the pieces of code that are chosen by the programmer in advance (before the compile time) may be upgraded. The authors do not discuss what compatibility of components means and when one component is substitutable for another one in terms of semantics. The validation of the component substitutability is limited to type checking. In their approach, components, may have their state. The state transfer between the old and the new version of the component is mentioned and an outline of saving and restoring the state is given, however, the definition of the component is not presented.

4.3 Evaluation of previous work

This section presents an analysis of the approaches to upgrading systems on the fly, as described in the previous section. The approaches are compared in terms of system features describing various aspects of the dynamic upgrade identified in section 3.4.

The following paragraphs describe the comparison features in details. The comparison is then summarized in a table. The analysis concludes with a list of the dynamic upgrade requirements that are not suitably met by the systems presented here.

4.3.1 Comparison criteria

The comparison criteria describe various aspects of system support for dynamic upgrade. The set of criteria is based on features originally proposed in [103]. In this thesis, the comparison is adapted to the list of problems and issues of dynamic upgrades that were identified in section 3.4. The list of criteria is also extended by some new features that described more details of the upgrade process. Additionally, the criteria refer only to a software support for dynamic upgrades.

The chosen comparison criteria include:

- **Unit of upgrade (granularity)** is the basic unit that the system allows upgrading. It can be:
  - *Zone* is a logical collection of processes, or containers in a component-related terminology, which are distributed over a number of hosts.
  - *Component* is a piece of software running within a address space that can offer its functionality through its contract to other components, either co-located or placed somewhere.
  - *Module* is a syntactic programming language artifact that comprises a set of procedures related to each other. It is an artifact that may have some state and is a singleton. In object-oriented languages, this concept has be elaborated and comprises to a set of related classes.
  - *Procedure* is a syntactic programming language artifact that offers some functionality available through procedure invocations to other program entities within the program.

- *Code instruction* is the smallest syntactic programming language construct which sequences form a procedure.

The finer the granularity of the unit, the more easily and quickly the system can upgrade program with small, localized changes. On the other hand, the finer the unit, the finer and more complex deployment and dynamic upgrade support have to be.

Programming language concepts, like statements, functions or classes are very fine-grained units of upgrade. Considering dynamic upgrades on this level of abstraction allow for defining precise upgrade validity rules and adequate upgrade validation tools, designing an optimal state transfer mechanism in the given programming environment. The disadvantage of these approaches is that the solution is programming environment dependent and in most cases, it is not generic enough to be applied to other development environments.

A process is an engineering term from the multi-tasking operating systems. A process is a runtime entity, a unit of execution that provides a virtual runtime environment for code execution. Usually, the code running in a process is a self-contained application program or a service offering its functionality to other programs running in the system. A process is also a unit of system resource management, i.e. some resources can be allocated to the process and their usage is controlled by the operating system on the process level. Since the process is a unit of independent execution and resource management it is a tempting candidate for a unit of upgrade. However, a process is a runtime entity and has a different life cycle from software that is the target of the upgrade. It is also one possible engineering realization of software runtime environment. The characteristics of the process are also operating system specific and reduce the application scope of such a unit of upgrade.

It is believed that component oriented paradigm of software development gives a much better way of dealing with software upgrades. A software component is view here as a natural unit of upgrade that allows coping with upgrades throughout the life cycle of software. The component is a single abstraction common in all software life cycle phases starting from design, through deployment and runtime to maintenance. Thus, considering components as a unit of upgrade enables overcoming the scope limitation of the approaches presented above and giving the reader a more complete view of issues related to dynamic upgrade that need addressing in different software life cycles.

- **System-support requirements**. It concerns the development and underlying runtime environment support for upgrading systems on the fly. Some example include: choice of the underlying platform technology, like middleware or the operating system.
- **Language Requirements** describe the programming language supported if the solution depends on some features of the language in which the units of upgrades are programmed.
- **Distributed interprocess communication** describes the sort of and the technology used for interaction of the distributed software parts. This affects the kinds of programs that can be upgraded.
- **Dynamics of change**. This describes the type and scope of the changes supported and permitted by the substitutability constraints of the dynamic upgrade (cf. section 3.4.1.1). Depending on the granularity of the upgrade different levels of changes are possible: for instance, when upgrading a procedure, the signature of the procedure, i.e. its interface does not change, whereas when upgrading a module, new procedures may be added and the existing ones cannot change signatures. The change types can be classified into two main groups with regard to the target of the changes.
    - *Interface changes*, which can be further grouped into:

        - No interface change. Only the encapsulated aspects of the unit of upgrade may change.
        - Monolithically increasing change of the interface. An example of this kind of change would be inheritance in object-oriented programming languages.
        - Arbitrary interface changes.
    - *Implementation changes*, which can be further grouped into:
        - Functional changes such as:
            - monolithically increasing (adding new functionality while the old one is unchanged),
            - replacing functionality (superset of states transitions plus new states), arbitrary changes (a totally different finite state machine)
        - Non-functional changes (performance, security, etc)

- **State transfer**. It describe the extent of the support for state transfer (cf. section 3.4.1.3).
- **Execution Model.** It describes the supported execution model of upgradable components. The model is either single threaded or multiple-threaded.
- **Upgrade process**. The criterion describes the features of the upgrade algorithm controlling the upgrade process. The algorithm's atomicity is described here in terms of number of phases the algorithm performs to terminate. An algorithm may be one-phased if the algorithm supports upgrades that can be made available immediately after the upgrade terminates. Two-phased and many-phases algorithms handle upgrades with higher dynamic of change, in which the additional or changed features of the new software version can be made available only with a delay related to a need of upgrading other software artifacts.
- **Active Target of Upgrade.** This criterion describes whether an upgrade of active runtime instance of software artifact is supported, whereas active means that the software artifact is being used by an running execution thread. In more technical terms, it that an active thread has invoked some code which belongs to the implementation of the software artifact and has not returned from this invocation. In case of an object being a runtime instance of a class, an object is active if a thread has entered one of the member functions (or a method) of the class associated with the object and has not returned from this function. The thread may be executing either the body of this method or other code invoked from the body of this method so that the invocation of this method is stored in the execution stack.
- **Type of Upgrade Target.** It describes the complexity of the upgrade target that can be handled in one dynamic upgrade process. The possible upgrade targets can be: simple or compound. A simple upgrade targets is a single software artifact, whereas a compound upgrade target is one consisting of many software artifacts that are potentially distributed. The compound upgrade target may be further divided into actively and passively replicated software artifact in case of replicated software or interworking if it consists of many artifacts using each other. Upgrading a compound upgrade target during one upgrade process is more difficult to deal with because it requires dealing with passivation of a group of possibly interacting runtime instances and more complex failure scenarios.
- **Upgrade management**. The feature shows in how far an upgrade process can be managed in the system. The upgrade management may be include handling a single upgrade process at a time or multiple upgrade processes. Additionally, it may handle automatic initialization of upgrade processes and recovery from non-trivial failures during the

upgrade process, including conflicts between dynamic upgrade processes.

- **Multiple Versions.** The criterion determines the support of multiple versions of a software entity running in a system at a time. Multiple software versions may be supported on the container/process level (if different version may run in the same container/processes) or system level (if multiple version cannot run in one process but may run in parallel in different containers comprising the distributed system).
- **Deployment support**. This feature describes the system support for deploying the new software. It includes handling software package versioning, dynamic loading and linking new code during the system runtime, unloading (removing from the process or container) the old code, etc.
- **Main limitations**. The main limitations of the system supporting the dynamic upgrades.

4.3.2 Comparison

The solutions supporting dynamic upgrades described in section 4.2 have been divided into two groups for better presentation:

- Systems with the software component as a unit of upgrade. These systems are presented in **Table 1**. The table includes also features of DUF, the solution developed and presented in the next chapters of this work.
- Systems with other software artifacts a unit of upgrade. These systems are presented in **Table 2**.

System / Feature	C2	Future CORBA with Online Upgrades	Future Java with Continuous Availability Spec.	SOFA/DCUP	DUF
Unit of upgrade (granularity)	Architecture-level component	CORBA component	Java Bean	DCUP component (framework of Java objects). Component may contain subcomponents.	DUF component
System-support requirements	C2 framework (Java implementation)	CORBA 3.0	JVM	DCUP architecture	Component-oriented middleware
Programming Language Requirements	Supports dynamic loading	IDL-2-languague mapping must be defined (available for Java or C++)	Java	Java	Object-oriented
Distributed interprocess communication	Arbitrary connector based	CORBA	RMI	Java RMI (planned be extended to CORBA)	Remote object communication (e.g. CORBA, RMI)
Dynamics of change	Arbitrary application architecture reconfiguration and component replacement	Possible component implementation changes. Not determined in the specification. No interface versioning. Evolving versions of a	Binary compatibility. JVM, JRE and package versioning	Component implementation	Open to the upgrade algorithm assumptions. Currently implemented algorithms permit changes of implementation.

		interface are not related to each other. Runtime environment and component implementation versioning is not specified.			and simple interface changes.
Degree of human intervention	Low (change specification on the architectural level; change validation)	Low(preparation of upgradable component implementations)	Low	Low	Very low (automated upgrade management)
State transfer	Not discussed in the work; component specific.	User has to define the component state and use the Externalization service for storing/restoring.	User has to define the component state and use the Java serialization for storing/restoring.	Java serialization applied to chosen "important" objects in the component implementation.	Serialization supported by the underlying middleware platform.
Execution Model	Single threaded	Single- or mutli-threaded.	Mutli-threaded.	Multi threaded	Single- or mutli-threaded.
Upgrade process	One-phase	One-phase	One-phase	Simple component replacement. Upgrades initiated by component provider or the upgrade infrastructure.	Thanks to extendable framework any algorithms can be added. Possible n.phase algorithms.
Active Target of Upgrade	Yes	Yes	?	No	Yes
Type of Upgrade Target.	Interworking	actively or passively replicated CORBA objects	Simple	Simple	Simple, replicated and coumpound.
Upgrade management	Changes specified in C2's AML	No	Out of scope	On demand policy.	Policy-based. New policies can be added.
Multiple versions	No	System-level	Container-level	Container-level	Container-level
Deployment support	Out of scope	CORBA deployment infrastructure	JAR packaging, beans assembling class loading.	Upgrade infrastructure allows for downloading the new software into the application premises (push or pull model).	Yes.
Main limitations	C2-architectures; Java; Simple component replacement	Limited dynamics of change, inflexible specification, weak upgrade management.	Java only, not extendable, no upgrade management.	Only inactive components can be upgraded. Homogeneous components.	

Table 1. Comparison of DU systems with a unit of upgrade which is a component.

System / Feature	CONIC/ POLYLITH	Chorus	Eternal	DRASTIC	PODUS	STL
Unit of upgrade (granularity)	Process	Process	CORBA object	Zone	Procedure	CORBA object
System-support requirements	Conic runtime environment	Operating system support for multi-threading and dynamic process loading	Fault Tolerant CORBA (Eternal)		None	XDR checkpointing libraries
Programming Language Requirements	Conic as ADL, C as programming lang.	C/C++	C++	Supported by CORBA	Procedural (C)	CORBA IDL or C++
Distributed interprocess communication	RPC/asynchronous messages	Message passing	CORBA	CORBA	RPC	CORBA
(Dynamics of change	Architectural changes; replacement of Guardians (service module) implementations	Code, date, interfaces	Object interfaces and implementations	Objects may change their type arbitrarily.	Procedure signatures, internal data structures, procedure implementation	Addition/removal of fields, changing sizes of fields
Degree of human intervention	Moderate	Low	Low (state mapping between different object implementation versions	Change absorbers needed to be written to transform the inter-zone interactions	Low to moderate	Moderate
State transfer	Strong mobility: data (de)serialization and execution stack (un)winding in C		Object state has to be expressed as value object.	Orthogonal persistency		Storing and restoring data structures in a language independent extension to XDR; complex (circular) data structures supported; user-defined mapping between the state format versions
Execution	Single threaded	Multi	Single- or mutli-	Single	Single	Single

Model		threaded	threaded.	threaded	threaded	threaded
Upgrade process	One-phase	One-phase	2 phase (prepare, change)	A zone is frozen; the list of change absorbers is upgraded; objects evolve and the zone is thawed.	One-phase	One-phase
Active Target of Upgrade	No	No	Yes	No	Yes	No
Type of Upgrade Target.	Simple	Simple	actively or passively replicated CORBA objects	Interworking (within an upgrade zone)	Interworking	Simple
Upgrade management		No	No	Out of scope	Out of scope	On demand policy.
Multiple versions	System-level	System-level	Container-level	System-level	Container-level	System-level
Deployment support	Out of scope	Out of scope	No	No	Out of scope	No
Main limitations	Special-purpose language; interprocess communication requires explicit intermodule links	Chorus specific solution: non-standard solutionm	FT support necessary, no upgrade management, no deployment		Requires top-down structured programs; all data as ADTs	Simple component replacement. No support for multiple types of upgrade targetss. Not easily extendable.

Table 2. Comparison of DU systems with a unit of upgrade which is not a component.

4.3.3 Conclusions

The presented solutions provide a support for dynamic upgrades to a different degree and for different technologies. Below are some statements summarizing the state of the art.

- **Specialized and main-stream solutions**. The presented solutions are grouped into proprietary and standardized solutions. Many proprietary solutions provide support for a specialized development and runtime execution environment and are not easy to port to other platforms. The support for dynamic upgrades has been only recently considered in the standardization activities of the main-stream middleware platforms (the CORBA online upgrade) or are being worked on (Java Continuous Availability). They (will) provide rudimentary support for dynamic upgrades that can be used by wider spectrum of users.
- **Weak flexibility and extendibility.** Event though the solutions deal with supporting evolution and extendibility of the software systems under upgrade, their own design is not flexible. They provide different degree for dynamic upgrades and do not allow for extending this support. Many solutions are rather monolithic and close in that they provide the dynamic upgrade support in form of a library or a self-contained toolkit. It is not easy to extend the specifications and add new upgrade mechanisms that would

overcome the existing limitations of the solutions. To gain a better reusability and portability, a more modular structuring, such as an open object-oriented framework or a composed-based approach, is needed.

- **No relation to deployment support.** The existing specifications focus on the runtime aspects of the dynamic upgrade process itself. However, the support for dynamic Upgrades should not only address issues related to runtime configuration, i.e. exchanging runtime artifacts in a software system. Additionally, support for deploying new versions of the code, including code distribution, code packaging and versioning as well as discovery of service instances deployed in the system needs to be given. Therefore, the dynamic upgrade support should be integrated with deployment facility provided by the middleware. In particular, dynamic upgrades could be seen a special case of service deployment, in which a new version of a running service is redeployed in lieu of its old version.
- **No management of dynamic upgrades.** Dynamic upgrades belong to maintenance activities that are performed during the system runtime in parallel to usual system operations. The dynamic upgrade process should be managed in the same way other maintenance activities are. In the existing solutions for dynamic upgrades, management support is missing which should include automated initialization and progress control of the dynamic upgrade processes, handling parallel upgrades, support for security of upgrades.
- **Full support for distributed systems**. An upgrade is a deployment or maintenance process that may involves communication and coordination of distributed objects in a system and may take some time to terminate. As the distribution inherently imposes a risk of failures, the process of upgrade should tolerate failures typically occurring in distributed systems. Additionally, running new code during an upgrade may cause some failures that could not happen before the system is upgraded. Consequently, a solution supporting dynamic upgrades should provide upgrade safety and reliability.

This short analysis identifies the main drawbacks of the investigated systems supporting dynamic upgrades in a distributed environment. It is a starting point for specification of the requirements on our solution which enables distributed software components to be upgraded..

The Solution

5 Model for Service Deployment and Upgrade

Software upgrades can be considered as a special case of software deployment in which the software components to upgrade have been already deployed and they have to be replaced with another component implementation as argued in chapter 3. This chapter specifies a model covering deployment of component-based distributed software and defines the dynamic upgrade as one use case. Having in mind a wider picture of activities related to dynamic upgrades, it is easier to:

- identify the mechanisms common for deployment and dynamic upgrade and the ones specialized to dynamic upgrade so that the further work can be better focused on the specialized mechanisms, and
- achieve a better degree of reuse of design and implementation of the available mechanisms in the current middleware platforms when prototyping a solution compliant to the model as requested by property P4.

The model for service deployment and upgrade consists of the following parts:

- *Use case model*, which defines the main actors involved in deploying services in a distributed system as well as the main capabilities to be provided by a system supporting deployment and upgrades, in particular. The model also identifies the main activities involved in providing the system capabilities. This part of the model fulfills the request model property P1 stated in section 1.2 and is presented in section 5.2.
- *Component model,* which specifies the basic notions related to service deployment and upgrade. It is defined in the form of classes and their interrelations. It addresses model property P2 and is presented in section 5.3.

The presentation of this model is preceded with the results of the requirements analysis in section 5.1. The demanded properties of the target system supporting dynamic upgrades is specified with a list of requirements defined when analyzing the drawbacks of the state of the art dynamic upgrade support systems presented in section 4.3. The list of requirements includes both functional and non-functional requirements and is structured into requirement sublists grouping requirements with regard to the central object they refer to, i.e. the dynamic upgrade support system as a whole, the upgrade process, and finally the upgrade management.

The approach taken to specifying the model for service deployment and upgrade is based on the Unified Process as described and interpreted in [52]. This development process involves top-down specification of the system, starting from system capabilities specified using the use cases, through a class-based domain model, design model until the implementation model which includes considerations specific to a specific implementation technology to be used to develop the software based on the model. The process is iterative and allows for coming back the previous phases of the specification and their revising In this chapter except for section 5.4, the model for deployment and dynamic upgrades is presented in a technology independent way and therefore meets model property P3 requested in section 1.2.

With a few justified exceptions, the graphical notation used to describe the model is Unified Modeling Language, version 1.4, a modeling language that is commonly used with the Unified Process and has been standardized by the OMG, following the object-oriented modeling paradigm. This language has been selected to meet the model property P5 stated in section requesting expressing the model using a well-.known notation.

The model presented in this chapter is then used as an underlying model for services to deploy and to upgrade in the next chapters. In section 5.4, more details of the design and

implementation model are given. The model has been developed and validated in the context of the IST FAIN project[23].

5.1 Requirement Analysis

In this section, a list of requirements on a dynamic upgrade support system is presented. The starting point of the requirement analysis process was the result of the survey of the state of the art systems supporting dynamic upgrades presented in chapter 4. The requirements for a system supporting dynamic upgrades presented in this section are stated so that:

- they elaborate and precise the goals of this work presented in section 1.2,
- the deployment aspects of the dynamic upgrade are explicitly taken into consideration,
- they allow for building a more general and extendable solution compared to the existing systems supporting dynamic upgrade.

Furthermore, an explicit requirement is added on the component-based model for the distributed services to convey our belief that components is natural unit of upgrade as argued in section 3.2. This requirement analysis also attempts to handle the complexities of distributed systems presented in section 2.1.1, by stating some non-functional requirements on the DUSS and the support mechanisms, including dependability, heterogeneity and portability.

The requirements are divided into six categories for the sake of better readability. Section 5.1.1 specifies general functional requirements and section 5.1.2 non-functional requirements on our solution. Because the algorithms for actual performing dynamic upgrades are a central point of interest to this thesis, requirements concerned with mechanism and algorithms for dynamic upgrades have been stated in a separate group of requirements: the functional requirements on dynamic upgrade process itself are stated in section 5.1.3 and the non-functional ones in section 5.1.4. Another requirement group are related to the management support for dynamic upgrades in a distributed environment and are enumerated in section 5.1.5. Finally, a list of requirements on the component upgradability to be supported by the solution complete the list in section 5.1.6.

5.1.1 General functional requirements

These requirements define the general functional properties of the system to be modeled, designed and developed in this work. When referring to a class of system fulfilling these requirements, a term Dynamic Upgrade Support System, or DUSS for short, is used. This group of requirements include:

R1. **Basic Deployment Capabilities.** A DUSS, as a deployment system should provide the basic deployment capabilities. In particular, it should be possible to make a service available in a system, to deploy the service to a suitable target environment, to remove it from this environment, and finally to withdraw a service from the system.

R2. **Support for distributed services**. A DUSS should handle distributed nature of components which may have been deployed in a number of containers and hosts of a distributed system. In case of the upgrade of a distributed service, a DUSS should identify and localize all the targeted service components installations in the distributed system and then coordinate an upgrade of these components.

R3. **Support for co-existence of multiple versions.** It is required that multiple version of system components may coexist in a system. Different versions of a service component may be released in a system, may be deployed in a target environment and may be running in the system at the same time.

5.1.2 Non-functional requirements

This list of requirements define non-functional requirements on the DUSS. They are as follows:

R4. **Extensibility**. An dynamic upgrade support system is required to be easily extensible. It should be possible to add both policies to manage dynamic upgrades in the system and mechanisms supporting these policies.

R5. **Portability.** An dynamic upgrade support system has to be so realized that it is easy to port the system to other middleware platform, by means of some middleware or its extension so that the solution is portable across hardware architectures and operating systems.

5.1.3 Functional requirements on dynamic upgrade process

These requirements are concerned with the functional properties of the algorithms for dynamic upgrades. They are:

R6. **Automated Upgrade Process.** The component deployment and upgrade, in particular, must be carried out as much automatically as possible. However, some human intervention may be allowed or sometimes necessitated, e.g., enabling the system deployer to support the system to recover from deployment/upgrade failures.

R7. **System consistency preserved during the upgrade**. An upgrade of a system must not transform the system into a inconsistent state so that the further processing will result in a system failure. What the system consistency means is inherent to the concrete application.

5.1.4 Non-functional requirements on dynamic upgrade process

The requirements presented here state non-functional characteristics of the algorithms controlling dynamic upgrades of distributed software components.

R8. **Minimizing the loss of the system functionality during the upgrade process.** The change has to be localized so that the functionality of the minimal part of the system affected by the change is degraded.

R9. **Minimizing the unavailability periods** in which the upgraded parts of the system are not able to provide their functionality

R10. **Dependability of the upgrade.** An upgrade should be an atomic action. Either the system is upgraded successfully and new version replaces the old one, or the system should fallback to the state before change. For all actions triggered by the component upgrade mechanisms there must exist reliable counter-actions that may cancel and bring to their backward state all impacted areas.

R11. **Upgrade transparency.** System components that have not been directly influenced by the system runtime change should not be aware of the changes during the system reconfiguration.

5.1.5 Upgrade Management

The management and coordination of dynamic upgrade processes in distributed systems is the central issue to the following requirements:

R12. **Automated Upgrade Management.** It should be possible to determine when the upgrade is to take place. Possible scenarios are: immediately after the upgrade request is delivered; when a component to upgrade is idle, or at a given time. An upgrade can either concern an instance of a component's type, some well-defined group of components (e.g. a management zone) or all the instances of the evolving type.

R13. **Support for multiple simultaneous upgrades.** Since delays in realizing services are inevitable in distributed systems, an upgrade may take some time to complete. In large scale distributed systems, where management is decentralized, there may be several sources of upgrade request. Thus, upgrades are often performed simultaneously and some upgrade synchronization is needed.

5.1.6 Upgradable Components

Upgradability can be seen as an attribute of a component implementation, similar to persistency or multi-threaded-ness. This section describes requirements on the DUSS concerning its impact on the way the software system as a whole, and a software component, in particular, has to be designed and implemented if a component is supposed to be capable of dynamic upgrades, i.e. *upgradable*.

R14. **Orthogonal Upgradability**. The upgrade capabilities of a component, which belong to the system management issues, should be separated from other aspects of the component, especially the component's business logic.

R15. **Simplicity of development of upgradable components.** The component developer should put as little as possible additional effort to provide components with dynamic upgrade capability. Thus, any programming artifacts that are needed by an upgradable component should be automatically generated by the development tools and transparently used by the system components.

R16. **Minimizing the set of constraints on the system** that an infrastructure supporting the dynamic upgrade should impose on the development process and the system itself. In particular the infrastructure should not constrain the system's architecture, nor programming style.

R17. **Heterogeneous service support.** A DUSS should support upgrades of heterogeneous services, i.e. services that may consist of components requiring various software technologies, i.e. programming languages and execution models. The component deployment mechanisms must be able to successfully install and activate a component, as well as upgrade a running component with a new version of the component. A new version of a component implementation may use another software technology and may need to be installed in a different container.

5.2 Use case model

This section presents the use case model for deployment of services in distributed systems. The modeled system is called hereafter *Deployment and Upgrade Facility*, or *DUF* for short. The system under discussion is briefly analyzed with regard to its required capabilities and its boundaries in section 5.2.1. In the subsequent sections, the system is described using the UML concept of actors and identified functionalities of the system with the help of use cases.

Thus, the use case model defines:

- main actors interacting with the system, whereas *an actor is anything with behavior, including the system under discussion when it calls upon the services of other systems* [92]. The actors are introduced in section 5.2.2.
- use cases which represent the set of all functionalities a system supporting deployment should offer. These functionalities can be considered as detailed specification of requirement R1 stated in section 5.1.1. Furthermore, the use case model identifies the main activities needed to realize the specified functionality as well as presents the interrelations between specified functionalities. The use cases are presented according to a use case description template defined in [92] section 5.2.3.

5.2.1 Capabilities

Deployment and Upgrade Facility should provide the basic deployment capabilities as stated in requirement R1 in section 5.1.1. At a greater detail, it means that the following capabilities have to be provided:

1. **Releasing and withdrawing services in the system**. The *Service Deployers* are provided with a capability to release their services in the distributed system. The service is released by registering its name and some deployment information (a list of required service component descriptors) with the network service registry, and uploading the service code including all the dependent code into the network-wide service repository. A service may also be withdrawn from the network when a service provider does not want to offer it to its customers.
2. **Deploying and removing services.** After the service is released in the network, the *Service Deployer* may want to deploy his service to a specific target environment, which is most suitable for the given user requesting access to the service. Because of the distributed nature of services, as stated in requirement R2, a target environment is formed by a set of active nodes on which the code modules of the active service are deployed. The *Service Deployer* may also remove the installation of the service from the given target environment if it is needed, e.g. if the service should not be offered in the system any more.
3. **Redeployment** or an **upgrade of service** is a specific case of deployment in which a service already deployed in the service is replaced with a given service. Additionally, it is required as stated in requirement R3 that a number of versions of a service may coexist in the system.
4. **Dynamic Upgrade of a service.** It is considered a special case of a service upgrade in which a runtime instance (or a group of them) of a service are replaced by another version of the service so that a number of constraints hold as defined in section 3.1.

5.2.2 Actors

The main actors communicating with the DUF system are:

- **Service Provider** is an entity who makes the service available to the target system. It is also the Service Provider that decides to withdraw his service from the system so that the service cannot be deployed in the system any more.
- **Service Deployer** is an entity who possibly initiates and coordinates the process of service deployment and in particular service upgrade. Traditionally, the Service Deployer's role is taken by the human system administrator. However, this role can be taken by a software system in order to automate the deployment and upgrade

processes. In the solution proposed in chapter 6, it is a specialized dynamic upgrade management framework that will play the role.

- **Target System**, which represents the distributed system which is the target of the deployment operations. The target system includes both physical infrastructure and the software running on top of it. The new services to be deployed or new versions of the running services become part of the Target System itself after the deployment or upgrade process, correspondingly, succeeds.

5.2.3 Use Cases

All above capabilities have been modeled as UML use cases and are depicted in Figure 14. This use case diagram presents the actors interacting with the use cases as well as the use case relationships. The following use cases of the model can be identified:

- **Release service**. It describes the capability of the DUF system to make a service available for deployment in the target system. The details of this use case are presented in section 5.2.3.1.
- **Deploy service**. It describes the capability of the DUF to deploy a released service in a target environment in the target system. The details of this use case are presented in section 5.2.3.2.
- **Upgrade service**. It describes the capability of the DUF to upgrade a deployed service in the target system. Service upgrade is considered as a special case of service deployment as argued in section 3.1. Additionally, service upgrade may include activities that a part of the service removal process. The details of this use case are presented in section 5.2.3.3.
- **Dynamically upgrade service**. It describes the capability of the DUF to dynamically upgrade a deployed service in a target system. It is modeled as a special case of a service upgrade. Except for the activities needed to perform an upgrade, additional means have to be taken to ensure that the dynamic upgrade process is performed with the given real time constraints. The details of this use case are presented in section 5.2.3.5.
- **Remove service**. It describes the capability of the DUF system to remove a service from a target system. The details of this use case are presented in section 5.2.3.6.
- **Withdraw service**. It represents the capability of the DUF system to withdraw a service from a list of available services in the target system. The details of this use case are presented in section 5.2.3.7.

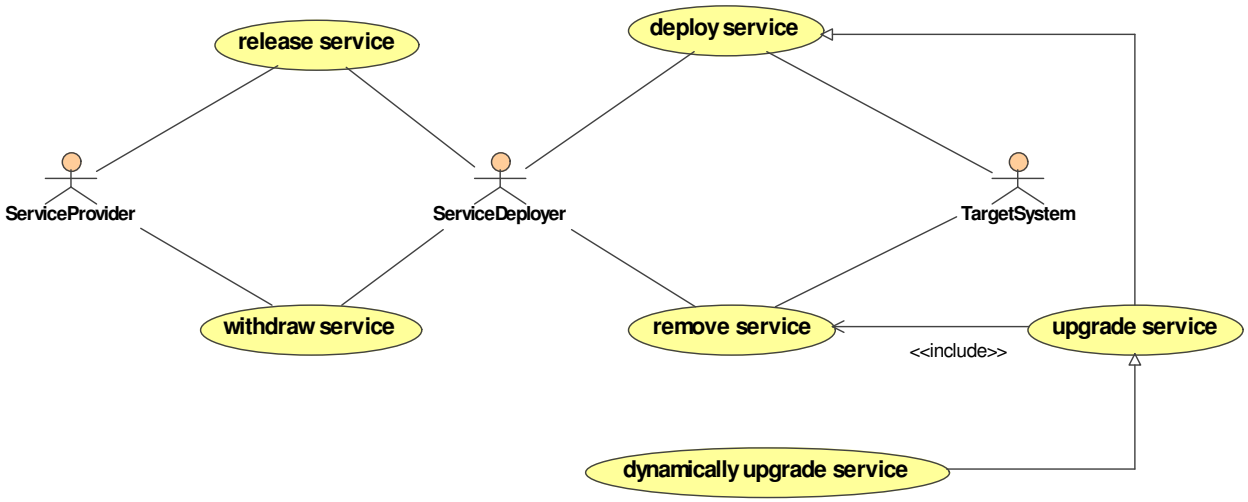


Figure 14. Main Use Case Diagram of the DUF.

The DUF system capabilities represented by the main use cases are related to each other in that there is a valid sequence in which they occur for a given service. The activity diagram in Figure 15 depicts this sequence of activities.

- First, a service is **released** by the Service Provider in the target system. This means that the Service Provider makes the service available to the users by announcing or registering the service in the target service. The information about the service are storied in the system and available for the deployment purposes.
- After having been released, a service may be **deployed** in the target system. It is the Service Deployer that initiates this process by interacting with the DUF and provides some additional information determining the preferred target environment for the service. The deployment process considers this information as well as the deployment requirements of the service itself related to its design. On a successful service deployment, the service is ready to use by its users. It is the responsibility of the Service Deployer to grant service access to the users.
- Once a service is deployed, the service may need to be **upgraded**. It is the Service Deployer (which may be also the service itself) to trigger the upgrade process. The target of the upgrade process is a deployed service. There are no requirements on the availability of the service so the old running version of the service may be shutdown and a new one started instead. If there are some additional requirements on the availability of the service during the upgrade, the process is called dynamic upgrade.
- A deployed service, i.e. a service installation, may be **removed** from the target environment it has been deployed to, if needed.
- Finally, the Service Provider may withdraw a service from the target system. From this moment, the service is not available for deployment anymore.

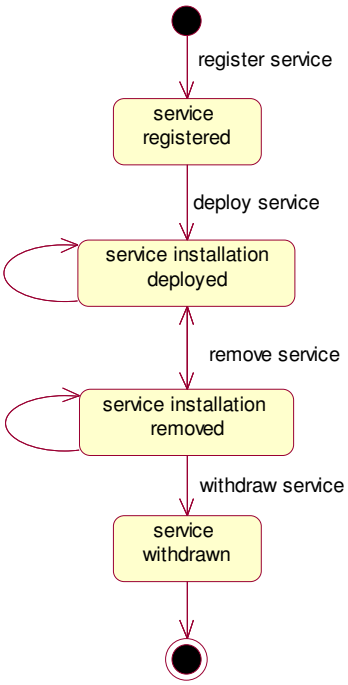


Figure 15. Activity diagram for a service processed by the DUF.

The following subsections explain the use cases identified above.

5.2.3.1 release service

The Service Provider who decides to offer his service in the target system has to release it The service is released by registering its name, its meta-information (a list of required service component descriptors) and a number of the service code modules with DUF.

Releasing service is part of the deployment cycle encompassing all operations that are required to prepare the service to be available for deployment after the development of the service has been complete. This phase include service packaging and announcement as described below:

Packaging

The software components that are to be deployed have to be packaged prior to transfer and delivery to the target nodes. The package must contain the software components in terms of the code and a meta-data description of the system including its requirements and dependencies.

Advertising

This phase caters for the dissemination of appropriate information to interested parties about the characteristics and requirements of the software system to be deployed. This can be achieved either automatically (a part of the deployment architecture discovers a service offered by the Service Provider) or with human interface (Service Provider interacting with the deployment architecture). The end objective is that the Service Deployer be notified of the availability of the service.

5.2.3.1.1 Actors

The following actors participate in this use case:

Service Provider, who wants to release his service and triggers the process.

- Service Deployer, who registers the information about the service released and maintains it for future service deployments.

5.2.3.1.2 Preconditions

1) The service is prepared for packaging.

5.2.3.1.3 Postconditions

1) The service is released, i.e. the service packages are available to the Service Deployer.

5.2.3.1.4 Description

1) Service Provider prepares his service components to release. This includes creating packages for the service components, which include service description and the service code.
2) Service Provider contacts the DUF in the given Target Service and provides him with information about the service to release. DUF records this information . It may involve downloading service packages to the Service Deployer infrastructure or just recording the details needed to download the service packages on demand.

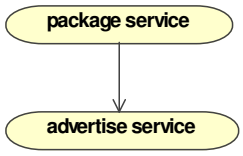


Figure 16. Activity diagram for use case "release service".

5.2.3.1.5 Extensions

2a) The service data is consistent or complete; the service is not properly packaged.

1) Service Deployer breaks processing the release request and signals an error.

5.2.3.2 deploy service

After a service is released in the target system, the Service Deployer may want to deploy his service to a specific target environment, which is most suitable for the given Service User requesting access to the service.

5.2.3.2.1 Actors

Service Provider, who triggers deploying a given service.

Target System, which is a distributed system in which the service is to be deployed.

5.2.3.2.2 Preconditions

1) A given service is released in the target system.

2) The service is deployable in the target system, that is:
    a. there exists at least one target environment within the target system suitable to deploy the given service,
    b. there exists a mapping that assigns a container capable of running a service component for each service component comprising the service.

5.2.3.2.3 Postconditions

1) The service is deployed in a suitable target environment.

5.2.3.2.4 Description

1) Service Provider requests deploying a given service.
2) Service Deployer identifies the target environment by matching the service requirements against the target system capabilities.
3) Service Deployer delivers the required service code to the target environment.
4) Service Deployer installs the service in the target environment.
5) Service Deployer configures the service.
6) Service Deployer activates the service.

5.2.3.2.5 Details

Figure 17 shows an activity diagram of the deploy service use case. The sections below provide more details on the steps of the deployment process.

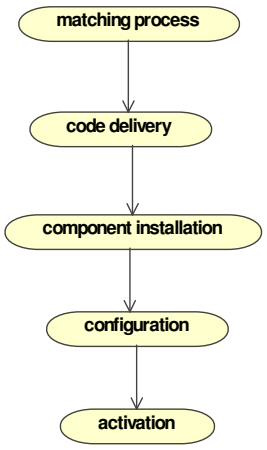


Figure 17. Activity diagram for use case "deploy service".

Matching process

The activity is aimed at determining an optimal placement of the service components onto a set of nodes available for service deployment. The process is based on matching the service requirements expressed in the service descriptor against the capabilities of the available nodes. In case, the service components are not predetermined before this process starts, the matching process also involves determining the components to be deployed for a given service.

The activity diagram giving more insight into a design of the matching process is shown in Figure 18. It shows a variation of a matching process in which the service components are not predetermined before the activity starts. The activity identifies the components required for a service by parsing the information on the service composition from the deployment descriptor. Then, two activities may be performed in parallel: getting information on the available capabilities of the distributed system and parsing the component requirements on the capabilities it needs to be deployed. The capabilities include the underlying technologies (programming environment and a specific hardware platform) as well certain system resources the service needs to fulfill its contract, including computational and communication resources. The information gathered during these two activities provide the input to the actual matching process that determines the assignment of each of the service components to a containers on a node belonging to the distributed system. The activity ends if the found assignment is feasible and the matching process is complete as it may be recursive. Otherwise, another iteration of the matching process occurs.

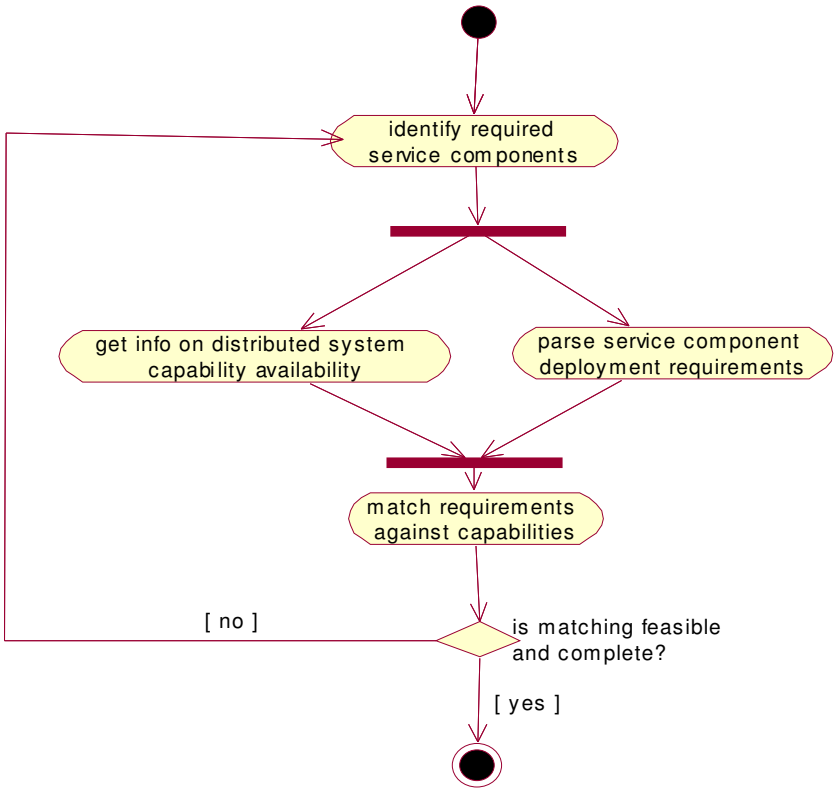


Figure 18. Process of matching service deployment requirements against capabilities of a distributed system.

Delivery

The delivery phase of the component package is done through the available transport channels, including the Internet. This involves considerations on the package size, target node quantity and transport quality. The delivery can follow either a "pull" or a "push" approach and can use wide-used protocols such as http and ftp.

Component Installation

The installation activity covers the actual placement and "tuning" of a software system into a suitable container on a suitable node. Given a released software component, this process combines the metainformation about the service encoded in the service component (e.g. descriptor) with the information at the target node in order to determine how to properly configure the software system. In the case that dependencies are specified, the process becomes more complex since it may involve fetching and deployment of necessary prerequisite components. The installation process may also involve the following activities:

- associating the names to the corresponding run-time code (e.g. upgrade registries, registering to the implementation repository)
- publishing the component references (e.g. registering to a naming service)

Configuration

This activity involves tuning of the components installed in their target environment. The service component may have certain parameters that have to be set with some values to be useable to the user. These values may be either default and can be stored in some meta data describing the initial (pre-set) configuration or may be computed in some specific algorithm using the system information.

Activation

Finally, the service may be activated, that is made available to the user access. This activity usually involves executing some initial code of the component, for instance calling external static functions in a C-based implementation of a component or instantiating a main class in a Java environment.

5.2.3.3 upgrade service

This use case describes service upgrades, which is considered as a special case of deployment in the context of this thesis (cf. section 3.3.3). That is why it is defined as a use case inherited from use case “deploy service”.

5.2.3.3.1 Actors

Service Deployer, who may trigger upgrading the given service.

Target System, which is a distributed system in which the service is to be upgraded.

5.2.3.3.2 Preconditions

1) The given service is deployed in the target system.
2) A new version of the service is released in the target system.
3) The service is designed to be upgradable and the state in which service upgrade is feasible can be reached within a time bound from the moment of *planned upgrade* time.

5.2.3.3.3 Postconditions

1) The service is successfully upgraded.

5.2.3.3.4 Description

1) Service Deployer requests upgrading a service and provides the details of the upgrade process.
2) DUF identifies the location of the upgrade target.
3) DUF delivers the required service code to the target environment.
4) DUF installs the service in the target environment.
5) DUF exchanges the old version of the service components with the new one.
6) DUF uninstalls the old version of the service components, if needed.
7) DUF removes the code of the old version of the service, if needed.

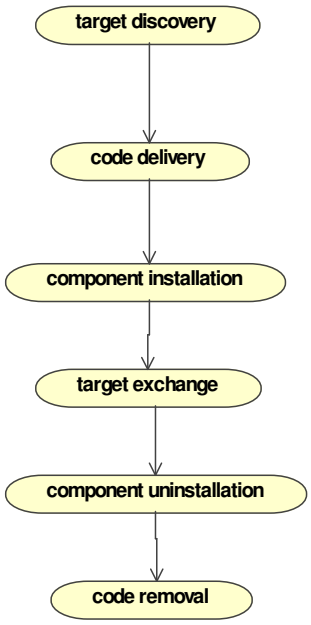


Figure 19. Activity diagram for use case "upgrade service".

5.2.3.3.5 Extensions

2a) The upgrade target cannot be located.

4a) The installation fails.

5a) The exchange of service version fails

5.2.3.4 Details

The process of upgrade is based on a number activities that are described in other use cases. Code delivery, component installation are described in section 5.2.3.2; target discovery, component uninstallation and code removal are described in section 5.2.3.6.

A new activity to this use case is target exchange.

Target exchange

In service upgrade, the old version of the service has to be replaced with a newer one. In a simple upgrade process, there are no requirements on the service availability. For this, the process of exchange may be as simple deactivating all service components to upgrade and

starting a new version of these components. For stateful services, the state can be transferred off-line.

5.2.3.5 dynamically upgrade service

The process of the dynamic upgrade is similar to the process of upgrade. It includes the same activities and could be represented with the same activity diagram as the one depicted in Figure 19.

Target exchange

The main difference is the process of exchanging the target. In case of dynamic upgrades, specialized mechanisms and upgrade algorithms are needed to match additional, mainly nonfunctional requirements on this part of the dynamic upgrade process including R8-R11 specified in section 5.1.4. Compared to the target exchange in the (static) upgrade process, the target exchange is an activity that:

- is performed during and in parallel to the normal operation of the system as an additional activity in the system,
- has to minimize the unavailability period of the upgrade target,
- has to guarantee that the system operates consistently during the exchange, e.g. no messages can be lost, the old references to the target component runtime instances have to stay valid,
- the state has to be transferred on the fly from runtime instances of the old version of component to the newly created instances of the new component version.

Example of mechanisms implementing this activity are described in chapter 7 of the thesis.

5.2.3.6 remove service

The Service Deployer may request to remove a given service installation, that is the deployed service, from its target environment it was deployed in. The DUF identifies all service component deployed for the given service installation and removes each of them from the containers comprising the target environment. The removal process includes the following activities depicted in the activity diagram in Figure 20:

Target discovery

In this activity, the target installations of the given service are discovered in the target system. This consists in finding the hosts and containers belonging to the target environment in which the service has been deployed.

Component uninstallation

This activity is concerned with uninstalling the service components from the containers found in the step above. This includes resetting the container to the state before the service component installation and potentially setting the container resources allocated during the code module installation.

Code removal

This activity results in making the code modules unavailable from the containers of the target environment discovered before.

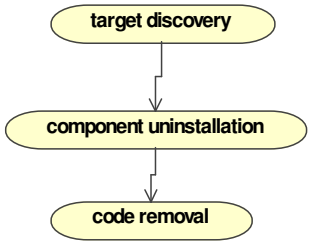


Figure 20. Activity diagram for use case "remove service".

5.2.3.6.1 Actors

Service Provider, who triggers removing the given service.

Target System, which is a distributed system in which the service has been deployed.

5.2.3.6.2 Preconditions

1) The given service is deployed in the target system.
2) The given service installation has not been removed.

5.2.3.6.3 Postconditions

1) The service installation is removed from the given target environment.

5.2.3.7 withdraw service

The Service Provider who has released his service in the target system, may also want to withdraw the service so that is it not available to Service Deployers. The `withdraw service` use case describes the capability of the DUF. To withdraw a service, the Service Provider has to remove both the service metainformation and the service code modules from the DUF.

5.2.3.7.1 Actors

The following actors participate in this use case:

Service Provider, who triggers withdrawing the given service by contacting the Service Deployer.

- Service Deployer, who discards the information about the service to withdraw and so that the service is not available to be deployed in the target system any more.

5.2.3.7.2 Preconditions

1) The service has been released in the target system.

5.2.3.7.3 Postconditions

1) The service is withdrawn from the target system and the service cannot be deployed by the Service Deployer any more.

5.3 Component Model

This section presents a component model that is considered further as the underlying service

model. The basic concepts of the model are defined and their relations explained.

The concepts are classified according to the software component life cycle phase. The implementation phase concepts are presented in section 5.3.1, and the deployment phase concepts in section 5.3.2. The runtime concepts are the contents of section 5.3.3.

5.3.1 Implementation Phase Concepts

The component implementation is specific to the software technology it is developed with. In this component model, the technological specifics are abstracted from and the component is defined as an aggregation of programming language constructs, called *Artifacts*, which are developed during the implementation phase. An sample realization of the artifact in a object-oriented programming language[131], like Java[118] is a class and in a function-oriented imperative language like C is a function. Apart from artifacts, a component implementation may include a number of `ConfigurationData` instances. This data expresses a persistently stored information used at the component's configuration time to initialize the component *runtime image* to reach a pre-configured state. `ConfigurationData` is modeled using the very flexible and dynamical modeling construct of a set of `Properties` objects, whereas each `Properties` object has a name and a value that may have values of `any` type, including other `Properties`. This construct is very expressive and allows for storing quite complex data structures.

The introduced concepts and their interrelations are depicted in a class diagram in Figure 21.

A `ComponentImplementation` is identified by its name. The name is unique among all implementations of the given component. The name may, for instance, include a string codifying the name of the component implementation provider.

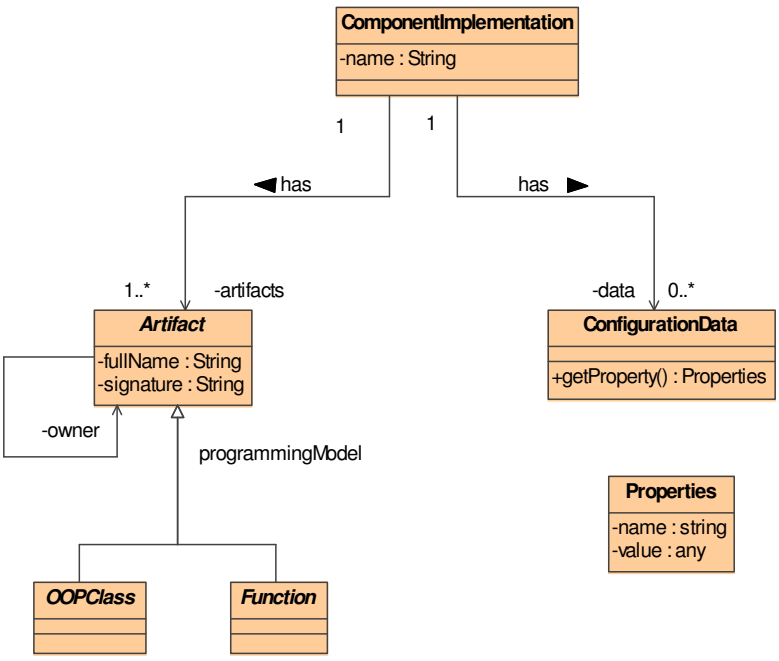


Figure 21. The implementation phase concepts of the component model.

5.3.2 Deployment phase concepts

In this model, service is defined from the deployment perspective. *Service* is a unit of

functionality that a Service Provider wants to offer to the customers by releasing it as described in the use case model in section 5.2.3.1. In terms of deployment, a service consists of a number of interconnected service components The service can be seen as a graph , where the nodes are service components and edges are links describing the communications between connected components.

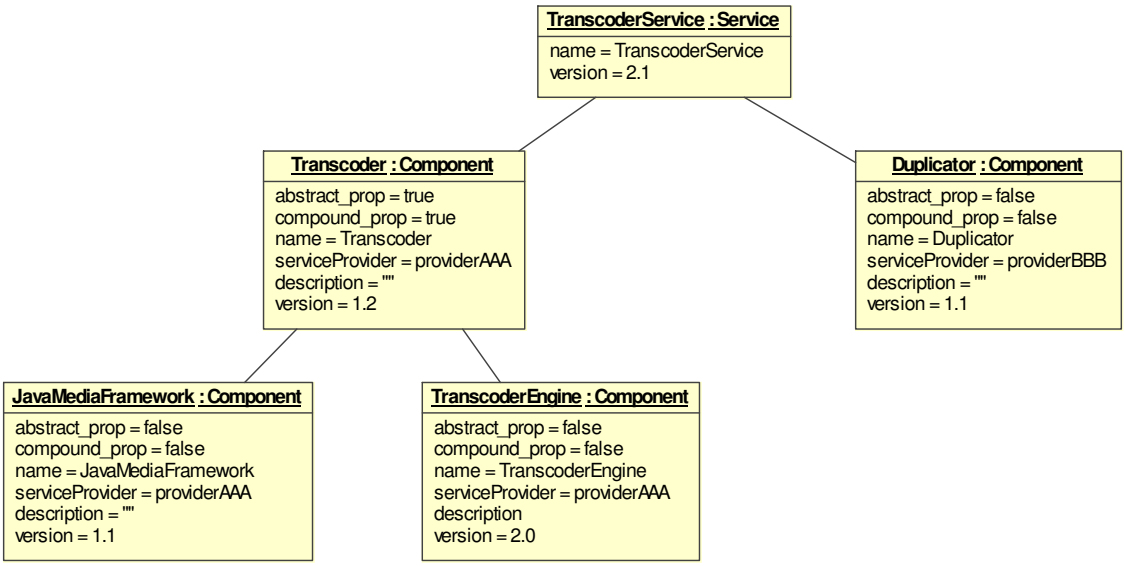


Figure 22. An example component-based service compliant to the component model.

Figure 22 depicts an example service. The service is called `TranscoderService` and consists of two top components: `Transcoder` and `Duplicator`. Whereas the latter is a simple service component, component `Transcoder` contains further subcomponents: `JavaMediaFramework` and `TranscoderEngine`.

One key characteristics of the software component as described in section 2.3, is its composibility. A component may be composed of a number components. In terms of deployment, it means that a component may need to access some other components. This idea is expressed by component dependencies. In the model, class `Component` has access zero of more dependent `Component` objects.

There are three classes of service components that differ in the terms of whether they consist of some service subcomponents and whether they directly refer to a code module.

- *Simple Component* is a service component without any dependencies. It contains just a reference to a code module.
- *Compound Component* is a service component consisting of subcomponents and having a reference to a code module.
- *Abstract Component* is a service component consisting of subcomponents and having no reference to a code module.

The relations between the classes modeling the concepts defined above are depicted in a class diagram in Figure 23.

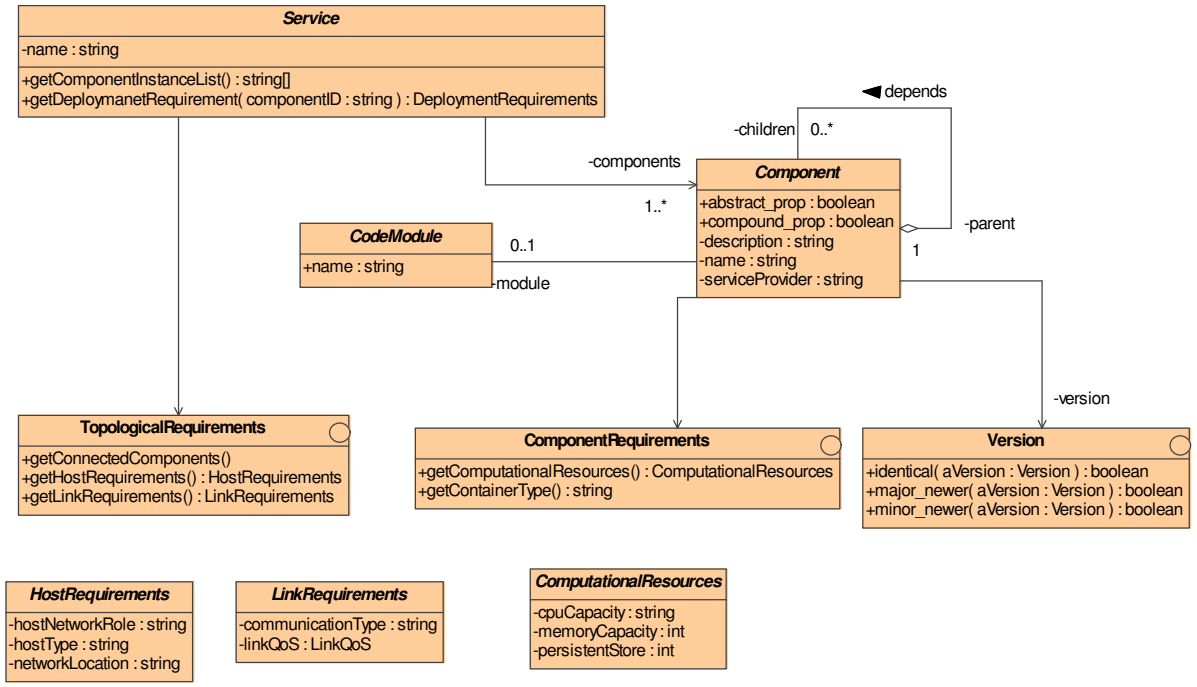


Figure 23. The deployment phase concepts of the component model.

This graph has some additional attributes described as service deployment requirements. There two types of requirements:

- *Topological requirements* represented by class `TopologicalRequirements` in the model. These requirements describe the needed characteristics of the environment where the service will be deployed. The characteristics concern both the hosts and network links between them. Typical host requirements include: the type of host, its role in the network (e.g. whether it is a intermediate egress router or an end node) or network location (e.g. the absolute IP address or relative to hosts where other service components are to be deployed) These requirements are modeled by class `HostRequirements`. Typical link requirements describe the type of communication between two components and the needed quality of the underlying network link (e.g. bandwidth, jitter, round-trip time, etc). The latter are abstracted in the model by class `LinkRequirements`.
- *Component requirements* (represented by class `ComponentRequirements` in the model) determine the needs of the service components with regard to the availability of computational resources in the target environment .Typical computational resources needed are the CPU processing power, the amount of the main memory and persistent store memory.

Class `Service` in the model has a name which determines the service template. Additionally, it provides methods to:

- Retrieve the names of all service components names
- Get the topological requirements of each service component.

Component, or *Service Component* is a basic unit of deployment in the model. It is described by a number of attributes to identify the component and are usually stored in the *service descriptor*. Optionally, the service has a reference to a *code module* which represents a file the component implementation is packaged.

Additionally, the `Component` has a version to distinguish various component releases of a `Component`. Whenever a new component is released, it is assigned a version that is unique. The version is determined by the Service Provider. For the purposes of this work, it is enough to apply a simple versioning scheme. In this scheme, the following rules apply:

Component version V^C is a pair of integer numbers *(m, n):*where *m* is called *major version number* and *n* is called *minor version number*. It will be noted as $V^C(m, n)$.

Version equivalence is defined as follows:

$$VC(m, n) = VC(p, q) \Leftrightarrow m=p \wedge n=q$$

There is an partial order relation defined on the versions as follows:

$$VC(m, n) > VC(p, q) \Leftrightarrow m>p \vee (m=p \wedge n>q)$$

For any two component releases R1 and R2 that occurred at time t_{R1} and t_{R2} it is required:

$$t_{R2} > t_{R1} \Rightarrow VC(m, n) > VC(p, q)$$

Additionally, if any major change occurs in this release then additionally the major version number should be increased so that $m>p$ and *n* reset to 0.

In case of a minor changes, it is required that n>q.

The definition of the major and minor change is related to the evolution model of the target system that specifies the possible changes in evolving components. This topic is not considered in this thesis. Hereafter, it will be assumed that:

a major change is a non-compliant change in the component interface and semantics

a minor change is a compliant change in the code module or in the component dependencies.

Whereas, a compliant change is a change that does not break the component contract that is defined by its interfaces.

To explain these rules using the example service depicted in Figure 22, assume a new version of service `TranscoderService` is released. Provided that a minor change was done in component `Duplicator` by Provider `BBB`, the component has to be released with version 1.2 and the service gets version 2.2. No other components have to change their version, since this modification does not impact them. After some time, `JavaMediaFramework` has undergone some major redesign (including some changes to its exported interfaces) and now Provider `AAA` is released again. Because of these major changes, the component version has to change to 2.0. Moreover, the major version number of all the components and services that depend on this component have to be increased after they are adopted to the changes and released. Thus, new release of component `Transcoder` appears in version 2.0 and service `TranscoderService` in version 3.0.

Service Deployment is a process of mapping the service components onto the target environment. It involves determining the target environment, identifying needed service components, fetching, installing and loading the appropriate code modules into the target environment.

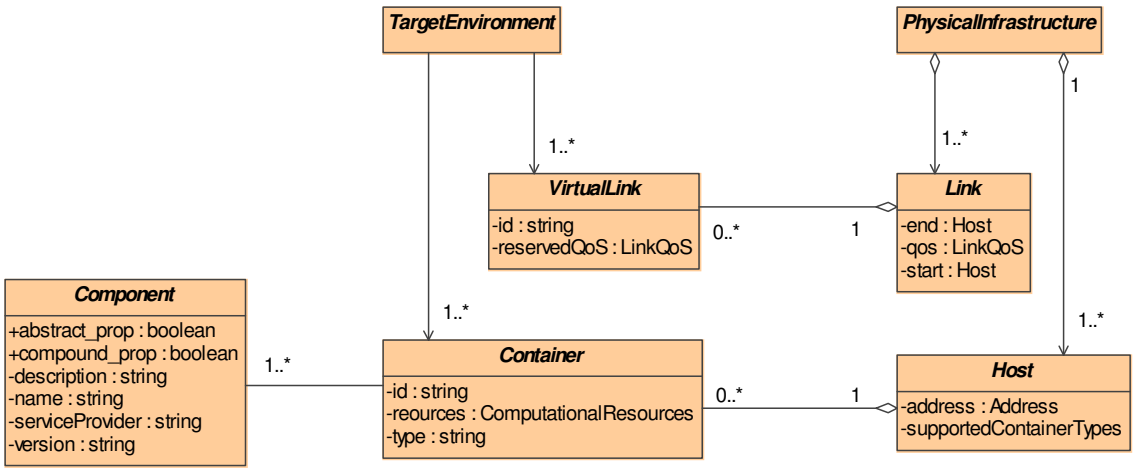


Figure 24. The deployment-phase relations between the component and its target environment.

The physical infrastructure provides the set of computational and communicational resources available to be used during the execution of a service. *Physical Infrastructure* consists of a number of computers, called further *hosts*, and a number of links connecting these hosts. Each link offers some communicational resources, that is connectivity with a given quality of service. Each host is identified with an network address and offers some computational resources.

To deploy a service, its requirements have to be fulfilled by the available computational and communicational resources. These resources are a subset of all resources provided by the Physical Infrastructure. They include a subset of host resources and a subset of link resources. The computational resources are managed using the term of Container.

A Container provides an execution environment with certain amount of computational resources that belong to a given host. The resources include memory, processing power and persistent storage. Each Container has its type that is determined by the execution model. This model is then defined by the development and programming environment, like Java or C++ as well as the basic services it offers.

The communicational resources are managed in terms of virtual links. A virtual link provides connectivity between two hosts with certain quality of service. Each link may have a number of virtual links provided that the sum of resources taken by them is smaller then the total amount of communicational resources of the link.

With regard to service deployment, the deployment process has to match the deployment requirements against the resources provided by the physical infrastructure and chose a subset of the physical infrastructure. This subset is called a Target Environment. In other words, *Target environment* is computed during the process of matching service deployment requirements against the resources available in the physical Infrastructure and will be a runtime environment for the service to deploy. The target environment consists of a number of containers in which the given service component is to be deployed and a number of virtual links that will be used by the service components to interact. It is the matching process (part of the deployment process) that assigns every service component to a container of the Target Environment .

The presented concepts are modeled in the corresponding classes presented in the class diagram in Figure 24.

5.3.3 Runtime phase concepts

Component Runtime Image is a runtime entity that comes into being when a deployed software component is loaded into a *container*. In a given container, only one component runtime image can be loaded at a time. However, a component runtime image may create many instances of its interfaces if needed.

The runtime part of this component model is defined in a very light-weighted aspect-oriented and low-coupled way. A set of interfaces is defined that can be provided by the component instances. This approach enables reuse of other existing component models and facilitates high portability. These interfaces are depicted in Figure 25.

It defines only a few interfaces that a component implementation may provide. Only one of these interfaces is mandatory. It is the `Reflective` interface which defines operations supporting a component with multiple interfaces. It provides operations for retrieving all interfaces provided by a component instance and getting a reference to any of the interfaces provided by the component instance.

Another interface is `InstanceFactory` which supports a universal way of creating instances of components. This interface should be implemented by each the component runtime image.

Yet another interfaces `ActivationListener` defines a way to notify the component runtime image or component instances of the fact being activated or deactivated.

The interface especially relevant to performing dynamic upgrades of software component is called `Upgradable` and is described in detail in section 6.2.6.1.

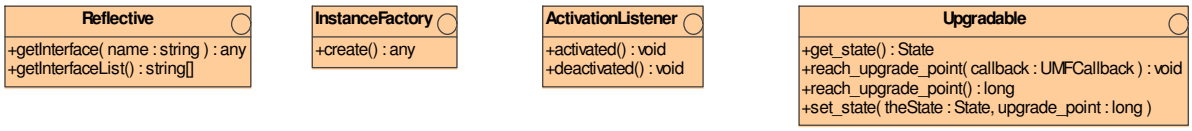


Figure 25. Basic runtime interfaces of component runtime images.

A runtime instance of a container is a instance of an execution environment for components runtime images. It provides component runtime instances with the basic services needed for their execution. In general, these services include access to the computational and communicational resources that are available to the component instances deployed in this container.

5.3.4 Phase Transitions

This section describes the interrelations of concepts defined in different phases of the software life cycle.

A service component has the following life cycle presented in Figure 26:

- First, a service component is *released*. This means that the service component makes the service available to be deployed in the distributed system.
- A released service component may be *deployed* to a target environment in the distributed system. As a result, the service component is installed in a container and is ready to its executable code may run.
- A service component may be *activated* in the container to be able to process the clients requests, if it plays the server role. The activation process may include instantiation of

certain runtime artifacts in case of the object-oriented execution model or invoking some initial functions in case of the function-oriented execution model.

- Once a service component is deployed, it may need to be *upgraded*, that is, a new version of the service component code has to replace the old version. In case, the service component has been activated, the upgrade process is called *dynamic*.
- A service component may be *deactivated* in the container. It makes the runtime artifacts of the service component be: (1) passive in that no external activity will result in execution of the runtime artifacts from the moment of deactivation, and (2) all running activities in the service component are possibly made frozen in a bounded time period after activation is requested.
- A deployed service component, may also be *removed* from the target environment it has been deployed to, if needed.
- Finally, a service component may be *withdrawn* from the distributed system. This makes the component unavailable to any activities in the distributed system.

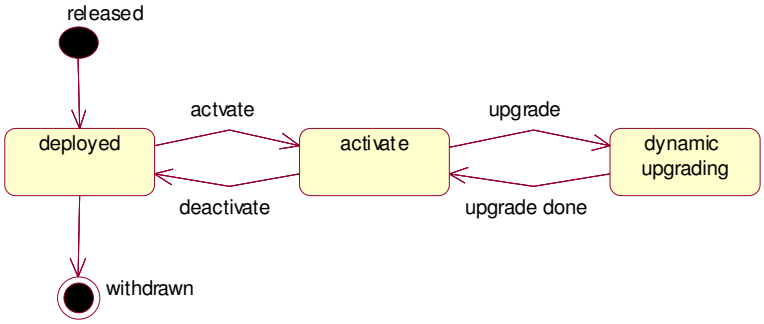


Figure 26. The service component Life-cycle phases.

Code Module is a file with the executable code of the given service component. The contents of the file is implementation technology specific. Given a reference to this file, the container infrastructure knows how to install and load the code module. A code module has the following phases of life cycle (also depicted in Figure 27):

- Initially, it is *released* to the system during the release time of the service.
- It is *fetched* onto a system node from the service repository and may be kept in the local service repository for some time.
- It is *installed* in a container instance, which means it is made available to the container infrastructure to load it if needed.
- It is *loaded* into a container. The module is either dynamically loaded into the running container instance or loaded by a new container instance when it started.
- It is *unloaded* from the container instance. The module may be unloaded and unlinked from the running container instance.
- It may be *uninstalled* from the container if it was previously installed in that container.
- Finally, it may be *removed* from the node.

When a service component is deployed, the code modules associated with it are usually fetched to the appropriate nodes of the target environment, installed and loaded into the target containers. In case, a service component is to be upgraded, the new version of the code

modules have to be fetched, installed and loaded, and afterwards the old version may be unloaded and uninstalled.

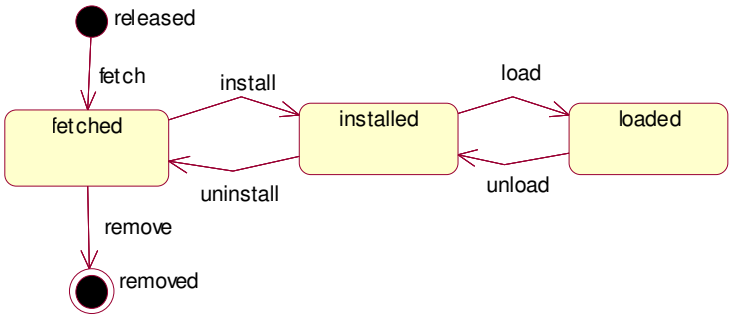


Figure 27. The life cycle phases of the code module.

5.4 Design and Implementation

The presented concepts of the use case and component model presented in the previous sections of this thesis have been realized by the author of this thesis and contributed to the IST FAIN (Future Active IP Networks) project. The functionalities designed in detail and implemented within this project are related to service release and withdrawal, service deployment and removal.

The Active Service Provisioning system is a subsystem of the FAIN system architecture[27] aimed at defining a generic architecture for IP-based active networks. It supports deployment of distributed heterogeneous services onto a network of active nodes, which are computing nodes which can be located as intermediate nodes in the network infrastructure, like traditional network routers and switches in the communication network, and are capable of running programs processing data traffic flowing through them. (see [125] for an introduction to active networks). Nevertheless, the approach is not limited to active networks and it be can applied to distributed systems including a typical enterprise distributed system Further details of the system design and implementation can be found in [112],[58],[23] and [22].

5.4.1 Approach Summary

The Active Service Provisioning System provides an example implementation of the fundamental capabilities for service deployment. These include releasing a service in a system, deploying it with a given set of deployment requirements as well as removing such a deployed service and withdrawing a service from a system.

The approach of the FAIN ASP system is summarized be the following list of features:

- *Two-layered architecture:* The rationale for choosing this architecture was a separation of concerns in the service deployment problem space. Whereas the network-level ASP deals with network issues including identifying nodes of the target environment for a given service with regard to the topological service requirements and network Quality of Service requirements, the node-level ASP is concerned with node specific requirements, including technology and other service dependencies.
- *Heterogeneous active service support:* The ASP enables deployment of active services

independently of the implementation technology they are based on. As long as a service is structured in terms of components and described using universal service descriptors defined in XML, it can be deployed using the ASP in the same way.

- *Multi-EE node support:* The ASP allows for deployment of active services on top of multi execution environment nodes. A service may consist of components to be deployed in different execution environment on a node. The decision as to which EE should be chosen depends on the execution capability and the availability of the suitable component implementation.
- *Deployment support for service components in different planes:* ASP is designed for deploying service code independent of the purpose. In the same way, it is possible to deploy components to run in management, control and data plane.
- *Hybrid two-phase process for the selection of a target environment:* The selection of active nodes suitable for a deployment of active services is designed as a hybrid of a centralized pre-section of the candidate nodes to be used for service deployment and a decentralized checking that the actual node capabilities on the candidate nodes suffice the service needs.
- *Universal service meta-information description:* The service descriptors are expressed in XML, a commonly-used SGML-based language standardized by W3C. By applying this language, the descriptors are easy to write for the service providers, easy to process by the programs (e.g. to generate it automatically by developing a service or to parse it) and, last but not least, as easy to extend. The common availability of the parsers also makes the software processing XML-based service descriptors easy to port to other platforms.
- *Binding of service components*: The FAIN ASP also supports binding service components forming a service. A service descriptor enables describing the way the components should be connected with each other and the node level ASP can interpret this information and perform the necessary actions.

Additionally, the service model supported by the ASP system is based on the one presented in section 5.3. The one presented in this thesis has been extended by adding the implementation phase and some runtime phase concepts to the model. The definition of some of the concepts in our model have been elaborated from the ASP Component Model and extended by the necessary definitions relevant to dynamic upgrades.

5.4.2 Realization overview

The design of the ASP system follows a two-level type architecture, where the network and the node level can be distinguished. This architecture is depicted in Figure 31.

On the network level the ASP consists of the Network Manager working as the central access point of the ASP to other Non-ASP sub-systems. The Network Service Registry and the Network Service Repository are dedicated to service information storage and delivery:

On the element level the ASP consists of the Node Manager, which is the central access point for the ASP on the element level. The Node Manager on the candidate nodes selected for deployment of service components is contacted by the network manager . In addition the Code Manager, the Service Creation Engine, the Local Service Registry, the Local Service Repository and the Reconfiguration Manager make up the node level ASP:

The following section describes the main entities of the Active Service Provisioning system.

Service Registry is used to manage the meta information about the services in a form of service descriptors. Service descriptors XML documents containing the information on the service attributes, like the service name, version, the provider, as well as service requirements in terms of container (execution environment) and network capabilities, and service dependencies. The descriptors are stored in the Service Repository, when a service component is released in the network. Network Manager and the Service Creation Engine may contact the Service Registry to fetch service descriptors. Finally, service descriptors are deleted from the Service Registry, if a service is withdrawn from the network.

- *Service Repository* is a data base for code modules. A code module is stored in the Service Repository, when a service descriptor referencing the particular code module is released in the network. The Code Manager, which is part of the node level ASP system and is described below, may fetch code modules from the Service Repository. A code module is deleted, if a service descriptor referencing the particular code module is withdrawn. As is the case for the Service Registry, several Service Repositories may coexist in a big network.

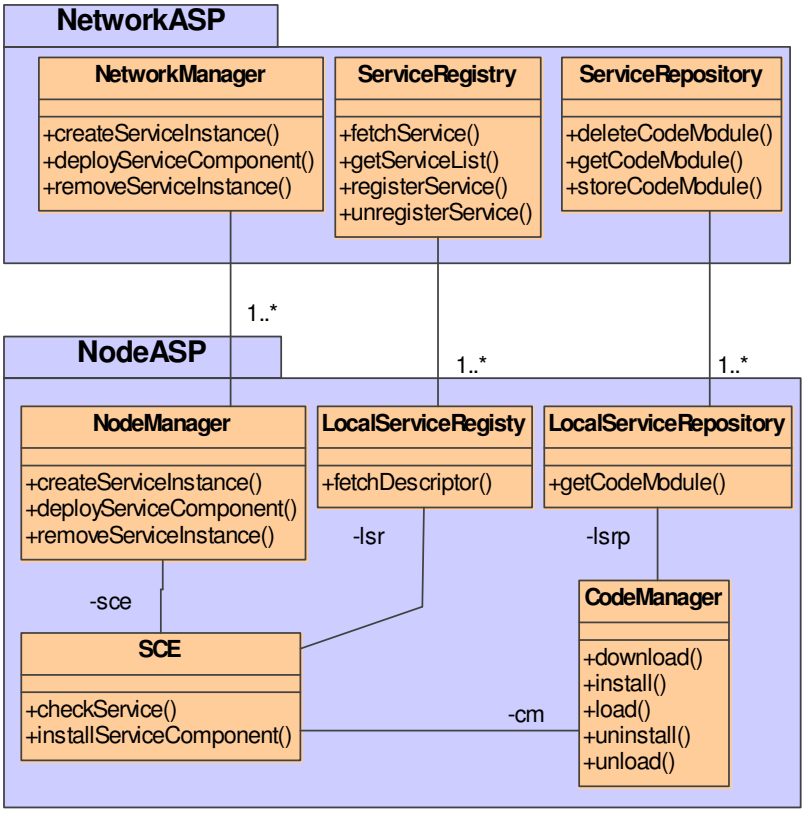


Figure 28. Design classes of the ASP system.

- *Network Manager* serves as an access component to the ASP system. In order to initiate the deployment of an particular service, a Service Provider contacts the Network Manager and requests a service to be deployed as specified by the service descriptor. It is also responsible for processing the network level service deployment requirements and matching them against the actual network capabilities. Deployment requirements include the service topology, which is determined by a number of service components to be located on nodes with given characteristics. The node characteristics include their node

roles in the network, like egress, ingress, end node, position absolute in terms of the network address or relative to nodes requested by other service components. Other deployment requirements are handled at the node level. This component has access to the information on how to access component installations and runtime component instances.

- *Node Manager* is the peer component to the network manager on the node level. The network manager communicates with the node ASP manager in order to request the deployment, upgrading and removal of service components.
- *Service Creation Engine* is a component that resolves the dependencies of a service. It does so by parsing the service metainformation contained in the service deployment descriptor. The dependency resolution is a recursive process which involves resolving dependencies of service components that the given service depends on. The dependency resolution process is not relevant to this thesis and it is not described here. An example of such a resolution and a detailed design of this component can be found in [23]
- *Code Manager* is a component which maintains the information about the code modules *installed* on the node. This component is contacted after the Service Creation Engine has resolved the dependencies of the service component requested to be deployed. The service component information comprises:
  - Service Component dependencies:
    - Resolved inter-component dependencies, in the form of a list of all the service components that the given service component depends on.
    - Environment dependencies, i.e. the dependencies on the execution environment that the code module associated to a service component is supposed to run in.
  - Service Component local installations:
    - Expiration date,
    - VE identifier and EE identifier, where the code modules are installed.

  The Code Manager holds the information about the installed service components in a data structure forming a directed acyclic graph (DAG). The nodes in the data structure represent the service components installed onto the node, whereas the edges represent the dependencies between these components. The data structure used to keep this information is depicted in Figure 29. The DAG has two levels: the first level consists of nodes representing service components that were requested to install by the SCE. Nodes on the second level represent the service components that the service components from the first level directly or indirectly depend on.

  The information maintained by the Code Manager is upgraded by:

  - SCE in case it requests fetching and installing a service component,
  - SCE in case it requests upgrading a deployed service component,
  - Code Manager itself whenever a service component expires and needs to be uninstalled from a given target environment.

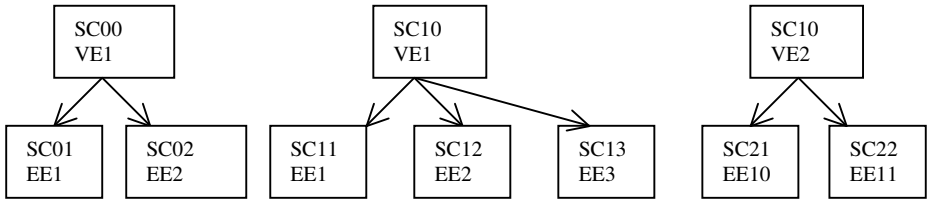


Figure 29. Data structure with the code module information in the Code Manager.

5.4.2.1 Releasing a service

The realization of use case *release service* described in section 5.2.3.1 is straightforward. The needed interactions are depicted in a collaboration diagram in Figure 30. **Table 3** explains the interactions depicted.

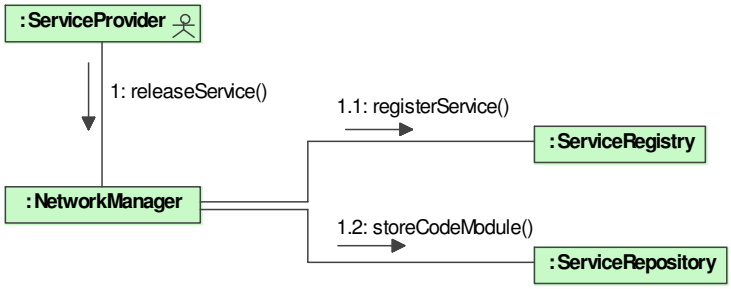


Figure 30. Collaboration Diagram: Releasing a service

Seq. No.	Interaction description
1.	Service Provider requests releasing his service. He accesses the network manager and provides the necessary information, including the service name, service meta-description as well as the references to the service's code modules.
1.1	Network Manager contacts then the service registry where it uploads the service descriptor(s).
1.2	It also uploads the code modules of the service onto the Service Repository.

Table 3. Interactions during releasing a service.

5.4.2.2 Deploying a service

Deployment of component-based services is a key functionality of the Active Service Provisioning system. In section 5.2.3.2, the use case description presents the main activities of the deployment process. These activities are now presented by adding the details of the ASP system. Figure 31 depicts a collaboration diagram with all the interactions within the ASP system and **Table 4** describes the details of these interactions.

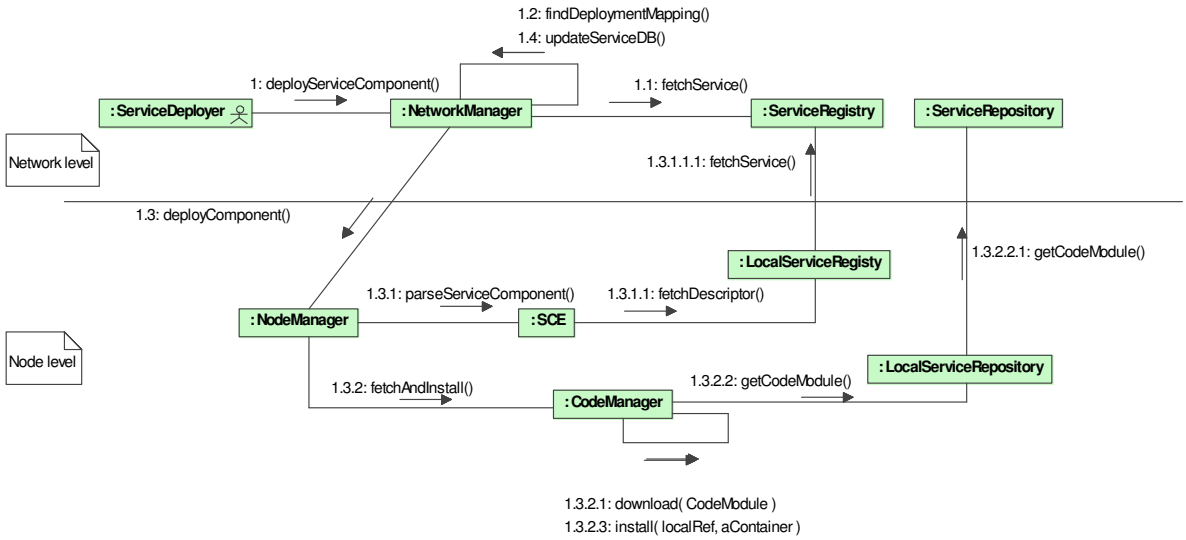


Figure 31. Collaboration Diagram: Deploying a service .

Seq. No.	Interaction description
1.	Service Deployer initiates the deployment procedure by requesting the network manager to download the service code and install it accordingly.
1.1	Network Manager fetches the network level service descriptor from the Service Registry to identify the service's deployment requirements with regard to the network topology.
1.2	It matches these requirements against the actual capabilities of the underlying network and finds out a number of candidate target environment, each of which consists a number of nodes. Each of these environments suits the topological requirements of the service. As a result of this operation, a number of mappings between each service component of the service and a candidate node is defined. It is however not sure whether the candidate nodes provide adequate computational resources to the service components.
1.3	To determine one target environment, Network Manager tries out each of these mapping one by one. Each sends the deployment request to each of the active nodes of the target environment.
1.3.1	Node manager supports the network level ASP with a selection of nodes of the target environment by checking whether the service component proposed in the mapping is deployable on the give node. In this simple scenario, it just passes the deployment request with its description to the Service Creation Engine, or SCE for short The latter resolves recursive dependencies of the service to deploy. It does so by parsing and processing further the service's deployment descriptor. When resolving dependencies, SCE has to decide which service component to chose from the set fulfilling the service component requirements. It does this by matching the service components requirements and the local capabilities of the node, like available container (execution environment) types or other resources that may be allocated by the Service Provider. The result of the dependency resolution process, is the so-called service tree. It is a data structure containing the

	information on the code modules needed to be fetched and installed in the suitable execution environments.
1.3.1.1	SCE fetches the node-level metainformation about the service to deploy from the Local Service Registry
1.3.1.1.1	If the Local Service Registry does not contain the needed information, it contacts the Service Registry and downloads the needed deployment descriptors
1.3.2	After the dependencies of the service components are resolved, Node Manager may fetch all missing code components onto the node. For this, it requests the Code Manager to coordinate further steps of the service deployment process and passes the service tree to it.
1.3.2.1	Upon this request, the Code Manager checks with its internal data base whether each of the service components in the service tree is available on the node, i.e. its code module has not been downloaded and installed, and its code modules are stored in the local service repository. If the code module is in the local cache of the Local Service Repository, a local reference to the code module is returned.
1.3.2.1.1	Otherwise, the Local Service Repository contacts the Service Repository and downloads the file.
1.3.3	Now the actual installation process is started. Node Manager requests the Code Manager to install the downloaded code module. Optionally, the component installation for *activatable* components may be then instantiated.
1.4	Network Manager updates the internal data base where the information about component installation are kept.

Table 4. Interactions during deploying a service.

5.4.2.3 Upgrading a service

Upgrading of a service starts with searching the installation of the service component in the target system. This information is kept by the Network Manager in its internal data base. Then the Network Manager request the upgrade in each of identified container, in which the components of the given service are deployed. More details are given in **Table 5** describing the interactions in the ASP system depicted in a collaboration diagram in Figure 32.

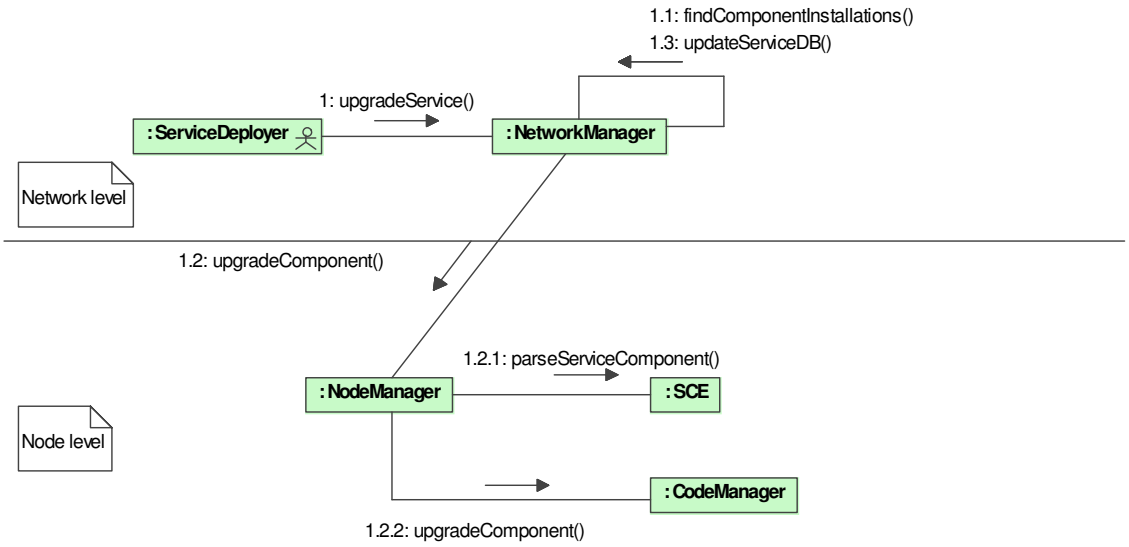


Figure 32. Collaboration Diagram: Upgrading a service

Seq. No.	Interaction description
1.	Service Deployer requests upgrading a deployed service. For that, he contacts the Network Manager.
1.1	Network Manager finds out where the given service type is deployed by searching its internal data base where component installations are stored whenever a service is deployed.
1.2	Network Manager iterates through all nodes where the service components are deployed and requests upgrading these components one by one.
1.2.1.	Node Manager identifies all the components that belong to the service by contacting the SCE that parses the service component dependencies. The dependencies may be already stored by the SCE so there is no need to parse the service descriptor.
1.2.2	Node Manager requests CodeManager upgrading each component that has to be upgraded.
1.3	Network Manager updates the internal data base where the information about service component installation are kept.

Table 5. Interactions during upgrading a service.

5.4.2.4 Removing a service

Removing of a service starts with searching the installation of the service components in the target system. This information is kept by the Network Manager in its internal data base. Then the Network Manager request the component removal in each of containers, in which the components of the given service are deployed.

The design of use case *remove service* described in section 5.2.3.6 is depicted in Figure 34. **Table 6** below explains the interactions depicted.

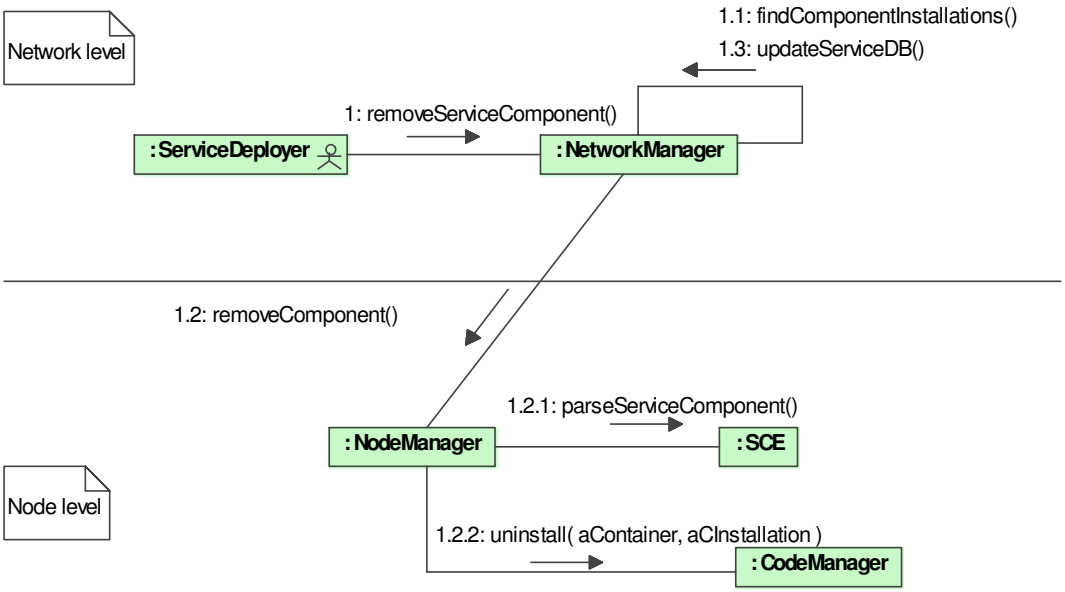


Figure 33. Collaboration Diagram: Removing a service

Seq. No.	Interaction description
1.	Service Deployer requests removing a service deployed before. For that, he contacts the Network Manager.
1.1	Network Manager finds out where the given service type is deployed by searching its internal data base where component installations are stored whenever a service is deployed.
1.2	It can iterate through the nodes where the service components have been deployed and request removing the components that belong to the given service.
1.2.1	Node manager coordinates the removal process on its node. It iterates all the components to remove and requests all dependent components from the SCE.
1.2.2.	Code Manager removes then all dependent components unless they are not used by component installations.
1.4	Network Manager updates the internal data base where the information about component are kept.

Table 6. Interactions during removing a service.

5.4.2.5 Withdrawing a service

The design of use case *withdraw service* described in section 5.2.3.7 is depicted in Figure 34. **Table 7** explains the interactions depicted.

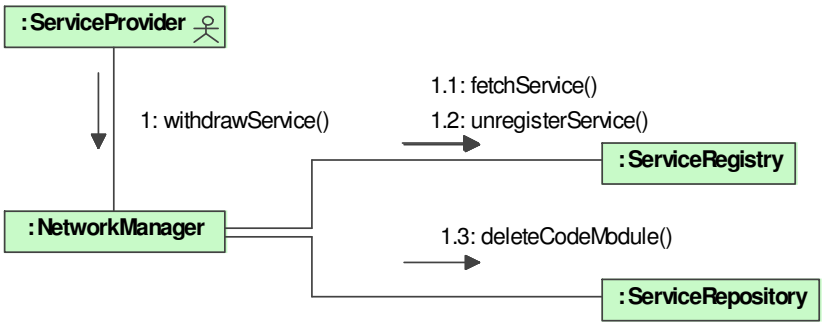


Figure 34. Collaboration Diagram: Withdrawing a service

Seq. No.	Interaction description
1.	Service Provider requests withdrawing his service. The service is referred by its name.
1.1	Network Manager access the service registry to fetch the meta-information on the given service. It parses the information to identify the code modules related to the service.
1.2	Network Manager removes the metainformation related to the service.
1.3	It also removes the identified code modules from the Service Repository.

Table 7. Interactions during withdrawing a service.

5.4.2.6 Service Description

Service composition and their deployment requirements are expressed in terms of service descriptors.

The process of deployment is split into two phases for the sake of separation of concerns, the service description has been split into two types of service descriptors. Network and node level descriptors. Whereas the network-level descriptor describes network issues including identifying nodes of the target environment for a given service with regard to the topological service requirements and network Quality of Service requirements, the node-level ASP is concerned with node specific requirements, including underlying technology and dependencies of other service components.

Network–level service descriptor. It provides the generic descriptive information about the service, lists the top components of the services and describes the topological requirements on the target environment. The network-level service descriptors are processed by the network ASP subsystem. A network-service descriptor for the example service `TranscoderService` depicted in Figure 22 is presented in Listing 1.

Listing 1 TranscoderService.xml:

```
    <?xml version="1.0" encoding="UTF-8"?>
    <NETWORK_SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="FAIN_NETWORK_LEVEL_DESCRIPTOR.xsd"
xsi:type="NETWORK_SERVICE">
        <DESCRIPTION>
            <SERVICE_NAME>TranscoderService</SERVICE_NAME>
            <SERVICE_ID>extended_transcoder_pure_java</SERVICE_ID>
            <PROVIDER>AAA</PROVIDER>
```

```
            <VERSION>2.1</VERSION>
            <SIGNATURE>0x1234ab006f8b11fa8c</SIGNATURE>
            <CLASS>economy</CLASS>
            <LICENSE>0.2</LICENSE>
        </DESCRIPTION>
        <SERVICE_COMPONENT>
            <NAME>Duplicator</NAME>
            <INSTANCE_NAME>d1</INSTANCE_NAME>
            <LOCATION>
                <RELATIVE>
                   <NODE_ROLE>ingress
                   </NODE_ROLE>
                </RELATIVE>
            </LOCATION>
        </SERVICE_COMPONENT>
        <SERVICE_COMPONENT>
            <NAME>TXengine</NAME>
            <INSTANCE_NAME>tx1</INSTANCE_NAME>
            <LOCATION>
                <RELATIVE>
                   <NODE_ROLE>egress
                   </NODE_ROLE>
                </RELATIVE>
            </LOCATION>
        </SERVICE_COMPONENT>
</NETWORK_SERVICE>
```

Node-level service descriptor. It describes a service component and its requirements. Like the network-level descriptor, it also provides generic descriptive information about the component including the component name, provider and its release version. The second part describes the configuration data needed to configure and activate the component. The data is as properties in a similar way as defined in the component model.

The last part of the service descriptor is dependent on the class of service component described. For a simple implementation component. this part contains a reference to a code module and identifies the target environment where the code module is to be installed. It also contains EE-specific information, which is used to perform EE-specific part of deployment process. It may also specify the computational resource needed by the component.

The service descriptor of an abstract component holds information about required sub-components and how they are to be bound to each other in order to perform the expected functionality. Finally, a compound implementation is a mixture of the two classes above, and hence contains both sets of information. Listing 2 presents the node-level descriptor for the example component `TranscoderEngine`.

Listing 2 TranscoderEngine.xml:

```
   <?xml version="1.0" encoding="UTF-8"?>
   <SERVICE xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="FAIN_NODE_DESCRIPTOR.xsd" xsi:type="COMPONENT">
      <DESCRIPTION>
            <SERVICE_NAME>TranscoderEngine</SERVICE_NAME>
            <SERVICE_ID/>
            <PROVIDER>AAA</PROVIDER>
            <VERSION>1.1</VERSION>
      </DESCRIPTION>
      <PROPERTIES>
            <PROPERTY>
                  <KEY>mainClassName</KEY>

      <VALUE>org.ist_fain.services.transcoder1.TranscoderManager</VALUE>
            </PROPERTY>
            <PROPERTY>
                  <KEY>mainCodePath</KEY>
                  <VALUE>/usr/local/jmf-2.1.1/lib/jmf.jar:/usr/local/jmf-
2.1.1/lib/sound.jar:/usr/local/jmf-2.1.1/lib:code/demux.jar</VALUE>
            </PROPERTY>
            <PROPERTY>
                  <KEY>AdmissionTimeOut</KEY>
```

```
                <VALUE>30000</VALUE>
            </PROPERTY>
        </PROPERTIES>
        <ENVIRONMENT>
            <EE_NAME>JVM</EE_NAME>
            <EE_VERSION>1.3.1</EE_VERSION>
        </ENVIRONMENT>
        <CODE xsi:type="CODE_LOCATION">
            <CODEBASE>jvm.TranscoderService.AAA.TranscoderEngine.jar</CODEBASE>
        </CODE>
</SERVICE>
```

The service descriptor is specified as XML schema. This technology has been chosen because of the platform-independent character of the language XML and its simplicity to specify, validate and process in automatic way because of huge availability of XML development support tools, including editors and parsers.

5.4.3 Discussion

The Active Service Provisioning system provides the advanced deployment support but only rudimentary support for upgrading services for distributed services. Namely, it provides the support for service release and withdrawal as well as service deployment and its removal. The most recent design of the ASP system [23] also provides capabilities for service reconfiguration in terms of adding or removing components to an existing running service as well as modification of connections between components at runtime. This reconfiguration can be initiated directly by the Service Deployer or by the service itself (the so called auto reconfiguration).

However, this system does not support all the features needed to upgrade services in a fully dynamic way. The following features still have to be provided:

- *Upgrade Process Management.* The ASP system follows the client-server paradigm, in which the ASP provides the deployment-related functionalities when requested. It itself is however not proactive and needs an external trigger to work, either from the Service Provider (Service Deployer and Provider are not distinguished in the FAIN business model) directly requesting a deployment-related action from the Policy-Based Management System that acts on behave of the Service Provider. Even though the Management System framework is very generic and extensible, no specialized policies and their enforcement modules are defined to cope with dynamic upgrades. Furthermore, defining such policies and implementing their logic in form of Policy Decision Points and Policy Enforcement Points is not trivial and their deployment requires running a heavy-weighted policy framework.
- *Specialized Upgrade Mechanisms.* The ASP does not provide all the mechanisms needed to perform dynamic upgrades. In particular, upgrade algorithms have to be defined for different types of upgrade targets.
- *Non-functional aspects of dynamic upgrades.* The ASP does not address the non-functional requirements R8 -R11 defined in section 5.1.4. In particular, the following aspects are not covered:
    - *Real time aspects.* The mechanisms should be concerned with the real-time aspects The target system availability should not be reduced
    - *Dependability aspects.* Upgrading of a system should not make it more susceptible to outrages.
    - *Upgrade Transparency.* No mechanisms are defined to make a dynamic upgrade transparent to the rest of the system.

Consequently, in the next chapters of this thesis, it is assumed that the middleware platform provides the functionalities as realized by the Active Service Provisioning system. In particular, it is assumed that the Infrastructure enables functionalities as defined in the use case model described in section 5.2 and that the services to be upgraded are compliant to the component model described in section 5.3:

In the subsequent chapters of this thesis, the features identified above needed to fully support dynamic upgrades of distributed component-oriented services will be investigated. The following features characteristics summarizes the approach presented:

- *A light-weighted framework for dynamic upgrade management.* This framework is presented in chapter 6.
- *Specialized Upgrade mechanisms.* The upgrade algorithms are presented in chapter 7.

5.5 Summary

In this chapter, a model for deployment and upgrade has been presented. The model is defined using the UML use cases. The model identifies the main actors involved in the deployment process, and the upgrade process in particular, and describes the key capabilities that the support system has to provide. The system upgrade is considered as a special case of software deployment, in which a service in redeployed so that the same service that has been previously deployed is replaced. A dynamic upgrade is then a special case of service upgrade, in which additional constraints have to be taken into account during the process.

The chapter also introduces a component model defining basic concepts describing software in subsequent phases of its life cycle including: implementation, deployment and runtime. The focus of the component model is set on the deployment phase.

Finally, one realization of the introduced concepts in the presented model is briefly described. This realization is based on a prototypical implementation of the FAIN Active Service Provisioning system that the author of the thesis has contributed to. The realization description concludes with a summary of the capabilities missing in the ASP system to fully support dynamic upgrades. These capabilities include a specialized upgrade management and specialized dynamic upgrade mechanisms. These mechanisms are the main topic of the next chapters of this thesis.

6 Managing Dynamic Upgrades

The management of highly available distributed systems involves activities to adapt and customize the system to the current requirements of its users. Dynamic upgrades allow to introduce the necessary changes to a running system without compromising its availability. As the process of upgrading may itself be a complex task that has to be done multiple times throughout the system's life span, an automatic support for its management could facilitate it and make it less error-prone. A support for well-balanced design of a management support for dynamic upgrades in a distributed environment is one of the research challenges for the designers of dynamically upgradable systems as mentioned in section 3.4. In section 5.1.5 a set of requirements on a dynamic upgrade support system were formulated.

In this chapter, the domain of dynamic upgrade management is presented in section 6.1. In section 6.2, a design model for the Dynamic Upgrade Management Framework is described. It describes the main subsystems and their design classes and documents the key design decisions. The presentation of the solution design uses the UML notation [96][81] and the design process is based on the Unified Process. The implementation related issues are described in section 6.3. Finally, section 6.4 summarizes the whole chapter.

6.1 Domain Model

The Dynamic Upgrade Management Facility represents the entity, external to the system to be upgraded, that is responsible for managing the upgrade process. Its task is to:

- Manage the service code provisioning and in particular deployment.
- Initiate the upgrade process if the upgrade is to be triggered externally.
- Coordinate the upgrade process which includes:
    - enforcing the upgrade management policies,
    - selecting the upgrade targets in case of multiple upgrade needed,
    - recovering the system from upgrade failures.

While designing our solution the following assumptions were considered.

- The application target of the Deployment and Upgrade Facility are large distributed systems in which management decisions would be too difficult and their enforcement rather intricate for a human administrator to make them. In particular, the complexity of the distributed software, like the dependencies between components, the distribution of the software components deployed in the system is fairly high.
- The system management and supervision may be done in a distributed way, possibly by multiple system administrators in parallel. There is a need to coordinate their actions and resolve potential conflicts in the management decisions.
- The management decisions and the mechanisms used to enforce these decisions can easily be separated from each other. The mechanism to perform upgrades can be located in the middleware platform whereas the policies may be defined based on the expertise of system administrators and interpreted by the management framework.

Thus, an automatic or semi-automatic support for management of dynamic upgrades is beneficial. Furthermore, a policy-based approach is suitable in this context.

6.1.1 Overview

This overall architecture of the DUF is depicted in Figure 35. It consists of the following

logical subsystems:

- *DU Management Framework* which is an extendable object-oriented policy-based framework providing support for upgrade management and coordination in a distributed system. The framework defines basic support for policy-based way of managing maintenance events in the system and dynamic upgrades in particular as well as a number of needed classes for controlling upgrade processes in a unified way. The concrete upgrade management policies and upgrade algorithms are defined in the following subsystems :
  - *Upgrade Management Policies* contains a set of predefined policies defining typical management schemes for dynamic upgrades.
  - *Upgrade Algorithms* contains a collection of different variants of upgrade algorithms. New algorithm implementations may be added to this collection.
- *DUF Infrastructure* is a part of the solution that is located in the middleware platform. This infrastructure is needed by the Management Framework and Upgrade Algorithms to control the communication in the system parts being upgraded as well as certain service life-cycle management, including component implementation loading, or activation.
- *Upgrade Target* is a part of the distributed *Target System* which is to be upgraded within an upgrade procedure. An upgrade target may consist of a component, which is the minimal unit of upgrade, or a set of components distributed in the system.

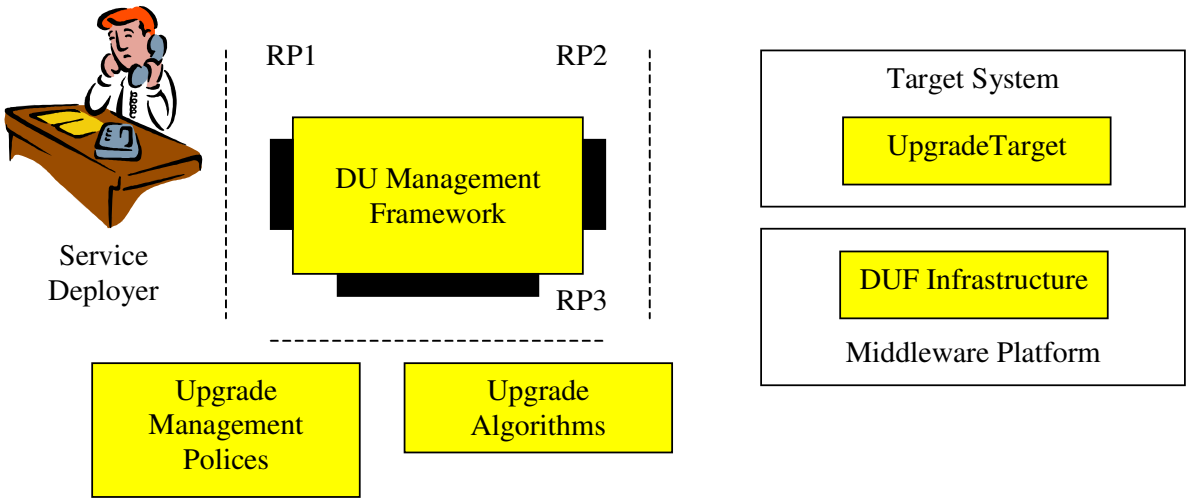


Figure 35. The architecture of the DUF and its distributed system with highlighted components related to the upgrade.

Three reference points, RP for short, are defined to allow for communication between the framework and other components of the system. The reference points are defined by a number of public interfaces of the logical components between which the reference point is defined:

- *RP1* which is offered to the Service Provider and allows for configuring the DU Management Framework. The Service Deployer may set up the framework to manage and coordinate upgrades in the system using the existing upgrade management policies and upgrade mechanisms.
- *RP2* is an interface defined for communication between the DU Management Framework and the Target System. It is used for coordinating and controlling a part of

the middleware platform relevant to the upgrades, called DUF Infrastructure. The interface is also use for retrieving information about the system for evaluating the upgrade management policies.

- *RP3* is an interface which enables extending the DU Management Framework by plugging in new management policies or upgrade mechanisms. In order for the policies and upgrade mechanisms to be plugged in, they have to implement certain interfaces and make use of the interfaces provided by the framework as defined in *RP3*.

6.1.2 Conceptual Classes

Figure 36 depicts more details of the DUF architecture. The architecture comprises the following conceptual classes:

- `UMFServer` is the entry point class of the DUMF. Its role is to accept upgrade requests from the service deployer, check their correctness and transform them into upgrade management policies.
- `UpgradeInitiator` is a class in the DUMF responsible for triggering upgrade processes. It does so by either regular evaluation of the triggering conditions of a set of upgrade initialization policies which are passive or by letting active policies to initiate corresponding upgrade processes by themselves.

`SystemMonitor` is a subsystem that monitors the state of the distributed system and reports it to the upgrade initiator. This information is needed to check whether the triggering conditions of the policies are fulfilled and the corresponding policy action can be started. The functionality of this class is directly related to the upgrade support and is rather of general character. As such, it is skipped in further design of the system elaborated in this thesis.

- `UpgradeProcess` is an instance of a deployment and management process that deals with exchanging programming artifacts comprising a target system. An upgrade process is determined by a concrete *Upgrade Target* and an *Upgrade Algorithm*.
- `UpgradeAlgorithm` is an algorithm that defines the steps to follow during a system upgrade process. Upgrade algorithms belong to upgrade mechanisms and are controlled by the DU Management Framework described above. There are a number of algorithms possible and each of them may be applied only in a specific context. An upgrade algorithm depends on the upgrade target and the parameters of the upgrade process, such as allowed dynamics of change. A number of algorithms are predefined in this thesis and new ones can be smoothly added to the framework.
- `UpgradeTarget` represents a set of component runtime instances that are to be upgraded in the given upgrade process. An upgrade target is one of the attributes determining an `UpgradeProcess`.

`ComponentRuntimeImageInstance` represents a single runtime instance of a component. Each instance is part of a runtime image of the distributed system; it runs in a container which is deployed in a host that belongs to the target system. Each runtime instance implements one `ComponentImplementation`.

`ComponentImplementation` abstracts an implementation of a component. A component implementation is determined by its code which contains a number of software artifacts specific to the programming language of the implementation.

`UpgradePoint` abstracts an upgrade point, which is defined as a point in the component

implementation code, which when reached enables to freeze the execution of the component runtime instance. Upgrades points are candidates when an upgrade process on runtime instances of this component implementation can start.

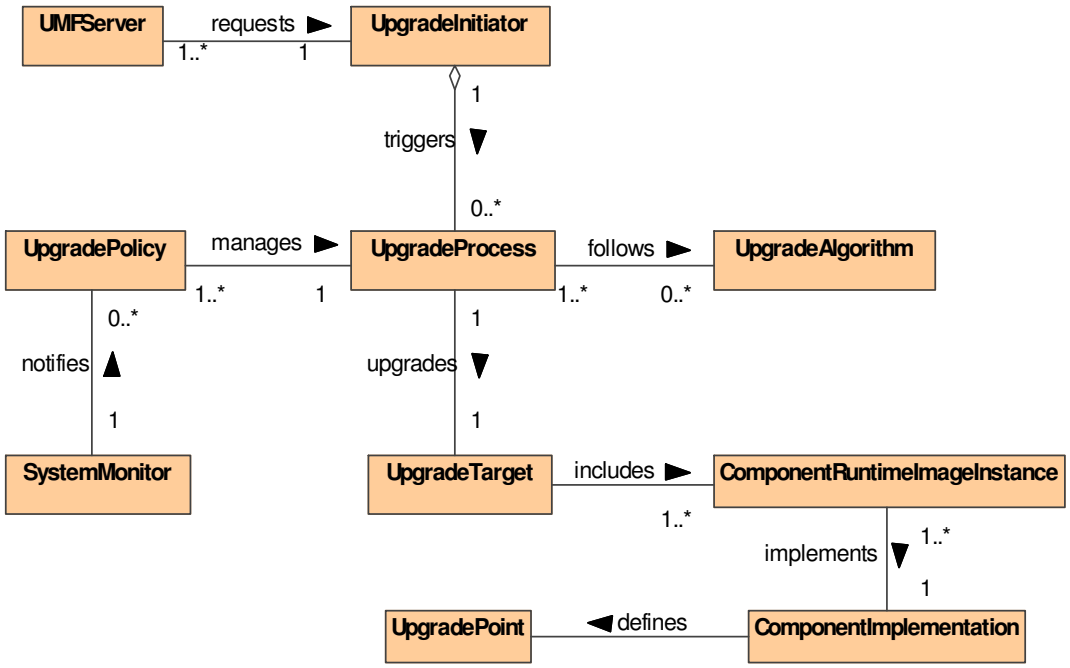


Figure 36. The conceptual classes of the DUF system.

6.1.3 Upgrade Management Dimensions

The process upgrade is a complex task and the way it is done may vary in many aspects. The objective of the upgrade management is to provide a means to control how and when the upgrade is performed within the permissible limits. The following dimensions of the upgrade process are distinguished:

- Initiation Source, meaning who or what initiates the upgrade process.
- Time, meaning when the upgrade process is initiated.
- Range, meaning a part of the system that the upgrade may influence.
- Atomicity, meaning the atomicity of the upgrade process.

Below these dimensions are described in more detail.

6.1.3.1 Initiation Source

This dimension concerns the way the upgrade process is triggered. Two main initiation sources of the upgrade may be distinguished:

- *Externally.* In this case, an entity external to the system to upgrade is responsible to start the upgrade process. The initiation may be started as a consequence of deployment of new software to the system or as a reconfiguration request of the system manager.
- *Internally.* The system itself checks the condition triggering the upgrade and starts the upgrade process. The upgrade is then a consequence of a state change of the system.

6.1.3.2 Time

Section 3.4.3.3.3 introduces the problem of finding an appropriate time when the upgrade can be carried out. On the other hand, the system handling upgrades may propose starting an upgrade process for some management related reasons at a given point in time. Because of the potential mismatch between a necessity and a feasibility to perform an upgrade in the target system, this thesis differentiate the *planned upgrade time* and the *real upgrade time,* when the upgrade process actually is started . Even though, the *planned upgrade time* may be equivalent to the real upgrade time, in general, the *real upgrade time* follows the planned upgrade time. This delay is related with the need to prepare the upgrade target for an upgrade.

There are some a number of possible predicates that may describe when to initiate the process of upgrading a system on the fly. With regard to the upgrade time of a component, an upgrade process may commence for instance:

- *On new code availability.* Whenever a new code implementing the object to be upgraded is available to the upgrade system, the upgrade may commence.
- *On time trap:* The upgrade may be scheduled at a given time point.
- *On certain system state*. The system state may be related to:
    - *The system load*. An upgrade may recommended only when the upgrade target is not actively processing any requests or is loaded to a certain extend.
    - *Malfunctioning detection.* The upgrade be triggered by a monitoring subsystem that discovers malfunctioning of the given software component expressed, for instance, by frequent crashes of the component.

6.1.3.3 Range

This dimension describes the part of the system that the upgrade process is to be applied. This unit of upgrade range is called an *upgrade zone*. In the distributed component systems, the upgrade zone can be:

- container,
- node, or
- domain consisting of a set of nodes.

Upgrade target multiplicity is another parameter related to the upgrade range. As the upgrade concerns the change of the component type, which may be instantiated in a running system, the runtime instances impacted by this change have to be described. The runtime entities may be:

- a certain instance of a component type,
- all instances in the upgrade zone (e.g. all the instances running on a specific node)

6.1.3.4 Atomicity

Another issue of upgrade management concerns the way the component is to be upgraded. The upgrade may be seen as:

- an atomic process, in which the component is upgraded as if it was performed in an instance, i.e. other components use either the old version of the component or the new one at a time. This strategy may be used when two versions of a component running in parallel and processing requests are not wished in the system.
- not atomic process, in which some components use still the old version while some

others use the new version of the component. This strategy may be used when invocations started before the real upgrade time are to be completed using the old version of the component, whereas the requests that came after the new version of the component is up and running, may be processed by the new component [87].

6.2 DUF Design Model

This section presents the details of the design model for the Deployment and Upgrade Facility. The model is composed of design classes that are contained in a number of packages. Section 6.2.1 gives an overview of the packages and the subsequent sections give more details of the classes, their relationships and interactions.

6.2.1 Package Overview

The DUF facility is comprised in the following packages:

- `DUF.ComponentModel` that contains rudimentary definitions of the component model. These definitions underlie the definitions in package `TargetSystem` and `Infrastructure`.
- `DUF.DUMF` is the core package of the DUMF framework. It contains definitions related to managing upgrade processes in the system as well as external interfaces to other subsystems including the Service Deployer and the Upgrade Algorithms.
- `DUF.DUMF.Policies` defines the modeling artifacts related to the policies for managing the upgrade processes.
- `DUF.Infrastructure` defines interfaces abstracting the functionality of the middleware platform needed to control and coordinate upgrades in the system.
- `DUF.TargetSystem` defines interfaces for communication with the target components that should be upgraded in the system.
- `DUF.UpgradeAlgorithms` includes definitions related to upgrade algorithms. The concrete algorithm implementations may be inserted in this package or in specialized packages.

The package structure is shown in Figure 37. The diagram shows the packages listed above, their main interfaces and abstract classes explained later as well as package interdependencies. `ComponentModel` provides basic meta-definitions that are used in all other packages and in particular in the `TargetSystem` that contains definitions related to upgradability in the target software systems. As `DUMF` provides basic interfaces and mechanisms supporting upgrade management, these definitions are used in package `UpgradeAlgorithms` comprising the implementations of upgrade algorithms. Both packages `UpgradeAlgorithms` and `DUMF` depend on package Infrastructure with definitions for underlying mechanisms offered by the middleware platform. The model elements related to policy management are encapsulated in a package `DUMF.Policies` internal to `DUMF`. The policy-based management is thus not visible outside `DUMF`. This allows for changing the paradigm to handling upgrade management in future design and prototypes.

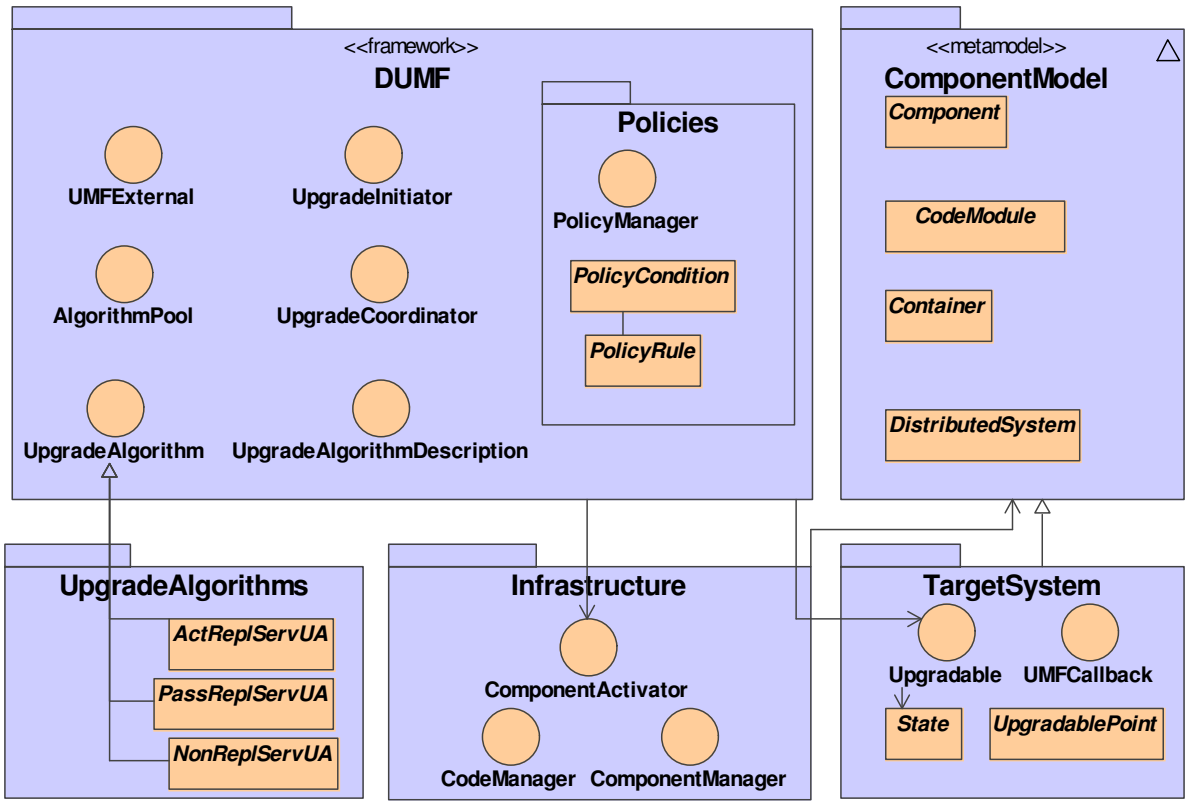


Figure 37. The DUF package structure and package interdependencies.

6.2.2 ComponentModel package

This package comprises basic notions of the underlying component model, including component, container or code module. These definitions are described in section 5.3.

6.2.3 DUMF package

This package comprises model elements that are related with Dynamic Upgrade Management. The classes in the package allow for requesting upgrades in the system, managing a number of upgrades processes independently requested and extending the framework by both new upgrade management policies and upgrade algorithms.

6.2.3.1 Public Definitions

The following interfaces are defined in package `DUF.DUMF`:

- `DUFExternal` is an interface that is the DUF access interface for the external Service Deployer. It allows the Service Deployer to configure the DUF so that it can automatically perform the upgrade requested. The DUF configuration is defined as a sequence of configuration policies. This interface defines operations to:
  - `request_upgrade()` for requesting the DUMF facility to take control over a given upgrade process according to the given parameters.
  - `cancel_upgrade()` for cancelling on demand an upgrade process that has already started or has been scheduled to start in as previously requested by the Service Deployer.

- `UpgradeCoordinator` is an interface that defines operations related to coordinating an upgrade process. In particular, it provides the following operations:
  - `initiate_upgrade()` supports triggering an upgrade process. It is usually used by the `UpgradeInitiatorImpl` when the latter positively evaluates the triggering conditions of the associated upgrade initiation policy.
  - `register_upgrade_algorithm()` allows for associating an `UpgradeAlgorithm` with the `UpgradeCoordinator`.
- `UpgradeAlgorithm` is an interface which has to be implemented by every upgrade algorithm. The operations offered by this interface allow for:
  - performing the subsequent steps of the algorithm whereas an algorithm is a N-phase transaction, where N>=1. Each phase of the algorithm is atomic from the perspective of the whole upgrade algorithm. It starts when executing operation `prepare()` and its effects can be made permanent by calling `commit()or` undone if needed by calling `rollback()`. The state diagram of an upgrade algorithm is shown in Figure 38.
  - algorithm configuring by providing parameters customizing the algorithm to the needs of the upgrade process (operation `configure()`).
  - retrieving some general characteristics of a concrete upgrade algorithm supporting this interface, including the number of phases (operation `phases()`).

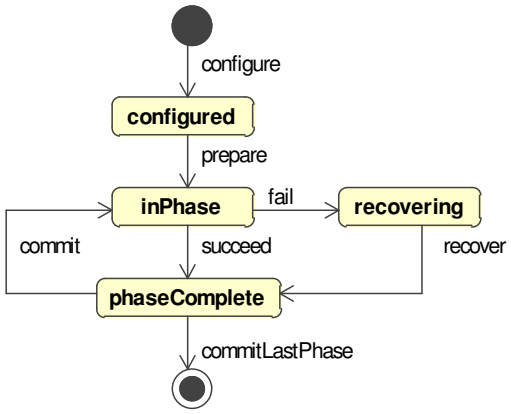


Figure 38. The state diagram for an upgrade algorithm.

- `UpgradeAlgorithmDescription` represents a meta description of an algorithm. The description contains information on the upgrade target type (operation `getAlgorithmName()`) that the algorithm is applicable and some attributes allowing to describe the properties of the algorithm, including (operation `getUpgradeTargetType()`).
- `AlgorithmPool` is an interface that abstracts a searchable container of upgrade algorithms. Upgrade algorithms may be registered and unregistered at runtime. The algorithms have different characteristics and applicability and there is a need to choose one that is most suitable for the given `UpgradeProcess`. The algorithms are searched after some criteria that are matched against the algorithm description.

6.2.3.2 Main Classes

The main model elements of the DUMF package are depicted in Figure 39. It also depicts classes and interfaces from other packages (Policies and ComponentModel) that are used by the DUMF model elements. The classes are filled with white color.

Many classes of the DUMF package implement the public interfaces defined above. The example of these classes include: `UpgradeProcess`, `UpgradeIntitatorImpl`, `AlgorithmUpgradePoolSimpleImpl` that implement the corresponding interfaces: `UpgradeCoordinator`, `UpgradeIntitator` and `AlgorithmUpgradePool`. The classes add some protected and private methods and attributes which define the logic of the operations contained in the interfaces. The following paragraph give more details on some of the classes defined.

- `AlgorithmPoolSimpleImpl` implements a searchable container keeping upgrade algorithm types. The container actually keeps a list of object instances of `UpgradeAlgorithmDescriptionImpl`. The list has typically a few element objects. The searching is based on simple iterating through the list and comparing the algorithm characteristics stored in the algorithm attributes against the constraints in the lookup criteria. The list is open-ended.
- `UpgradeAlgorithmDescriptionImpl` implements interface `Upgrade-AlgorithmDescription`. It is designed as an abstract factory[28] and the method `create()` has to be overwritten by a factory class of a concrete upgrade algorithm.
- `UpgradeProcess` is a class representing a concrete upgrade process determined by the upgrade target. Upgrade process are triggered by the `UpgradeInitiator` and coordinate the activities related to the upgrade by following the `UpgradeAlgorithm` associated with the process. The class implements thus interface `UpgradeCoordintator`. An `UpgradeProcess` follows exactly one `UpgradeAlgorithm`. It is designed to start the consequent phases of the associated upgrade algorithm and handle the phase failures reported by the algorithm by repeating the failure. A more complex coordination schemes can be added to the framework by inheriting this class.
- `UpgradeAlgorithmDescriptionImpl` is an abstract class that implements interface `UpgradeAlgorithmDescription`. It is designed as an abstract factory[28] and method `create()` has to be overwritten by a factory class of a concrete upgrade algorithm.
- `UMFServer` is a class that provides access logic to the DUMF framework for the Service Deployer. It allows the latter to request the facility to perform upgrades of given parts of the target system, the `UpgradeTarget` on the given conditions. Whenever the server deployer has requested an upgrade, the facility takes responsibility of detecting whether and when the given condition occurs and triggering the corresponding actions leading to the upgrade. The class is responsible for accepting the service deployer's request, checking its validity and translating the request into a form for easy further processing with the DUF facility.

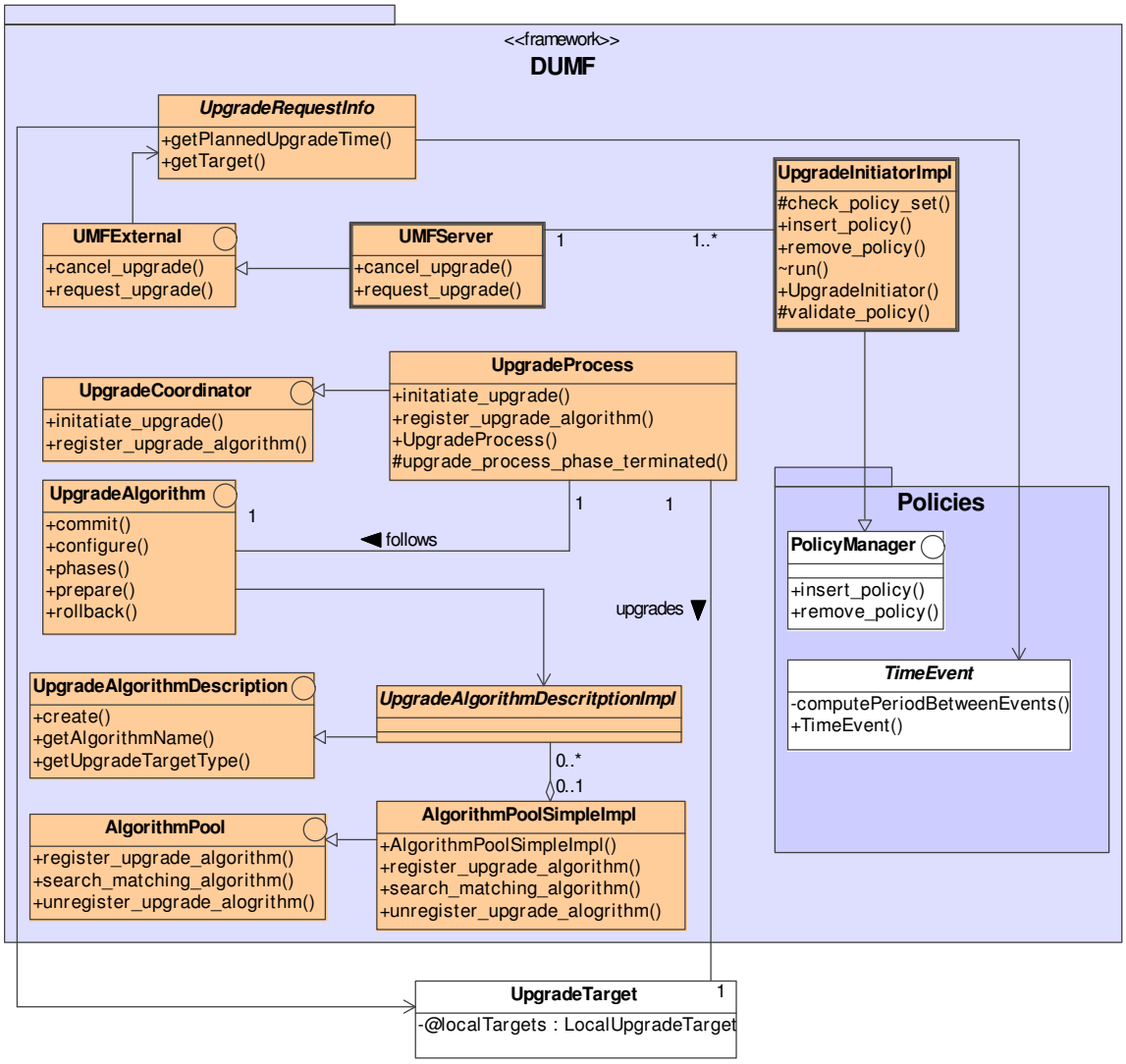


Figure 39. Class diagram for the `DUMF` package.

- `UpgradeInitiatorImpl` implements the process of continuously evaluating the upgrade initiation policies and determining the start time of the upgrade processes managed by the DUF facility. The activity diagram for the `UpgradeInititor` object is depicted in Figure 40.
- `UpgradeRequestInfo` is a simple data set class. It represents an upgrade request that is ordered by the Service Deployer. The request is created and filled in with the upgrade parameters including the upgrade target and the event that should trigger the upgraded, by the latter and sent to the DUMF framework for execution.

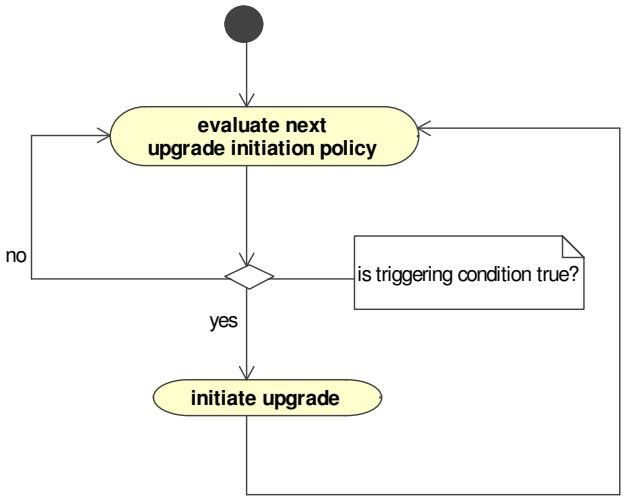


Figure 40. An activity diagram for the `UpgradeInititorImpl`.

6.2.3.3 Interactions

This section describes the interaction between the key components of the DUMF framework.

6.2.3.3.1 Handling an upgrade request

When a Service Deployer wants to request an upgrade in the target system, he has to contact the DUF facility. It is realized by accessing object `UMFServer` implementing interface `DUFExternal` as shown in Figure 41. `ServiceDeployer` creates and sends an upgrade request describing the details of the upgrade (steps 1-3). Such an upgrade request includes information on the upgrade target and the planned upgrade time. The planned upgrade time may depend on the system state. Internally, such a request is translated into an upgrade initialization policy validated (step 5) and stored with the `UpgradeInitiator`. The latter object is responsible for evaluating all upgrade initiation policies registered (step 8) and triggering upgrade processes when the corresponding triggering condition is fulfilled. In this case, object `UpgradeCoordinator` is created and configured according to the specifics of the initial upgrade request so that a corresponding upgrade process can start (step 9-10).

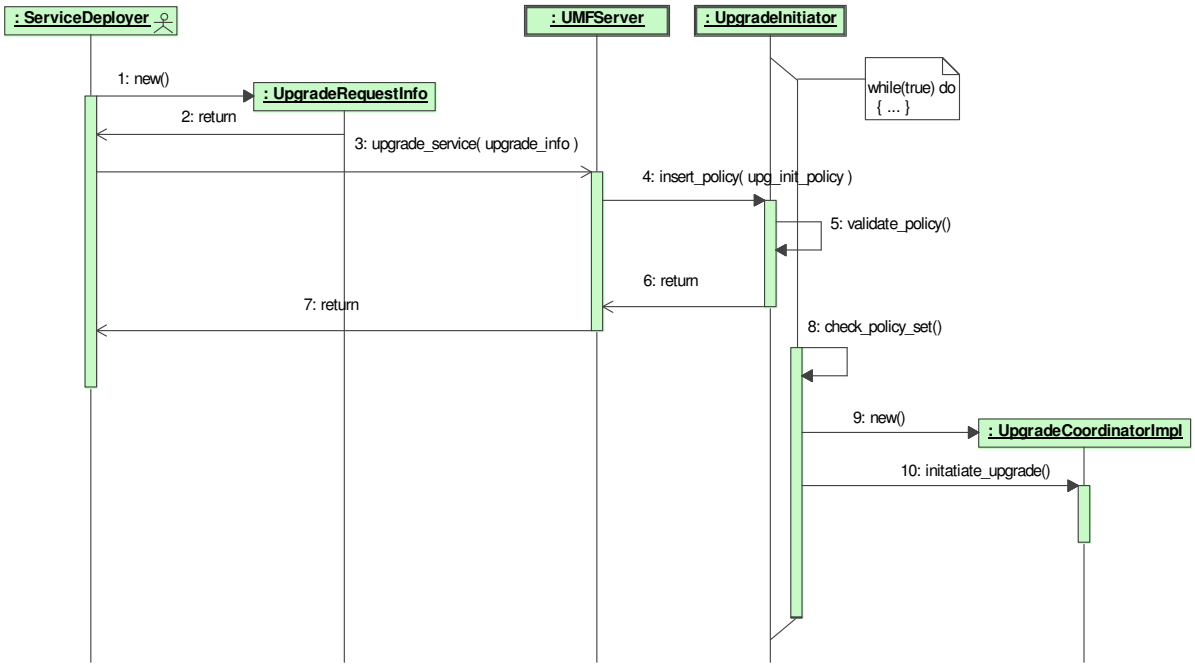


Figure 41. Handling an upgrade request(sequence diagram).

6.2.4 UpgradeAlgorithms package

This package contains definitions related to upgrade algorithm. It provides some abstract classes and interfaces that has to be implemented by concrete upgrade algorithm classes. The classes defined currently in this package define some simple classes that help with the upgrade algorithm classification. The concrete implementations of the upgrade algorithm may also have some implementation parts that is platform specific. This package is thought for the platform independent parts of the algorithm implementation. The other part of the algorithm is placed in the platform-related packages. These packages are named: `DUF.UpgradeAlgorithms.Jgroup` for the Jgroup/ARM[63] middleware platform or `DUF.UpgradeAlgorithms.CORBA` for the CORBA[73] middleware platform.

6.2.4.1 Public Definitions

The following abstract classes are defined in package `DUF.UpgradeAlgorithms` :

- `ActReplServUA` defines an abstract class for upgrade algorithms suitable for upgrading actively replicated servers. Concrete implementations, like the one presented in section 7.2.4, of such algorithms should be derived from this class according to the Strategy[28] design pattern.
- `PassReplServUA` defines an abstract class for upgrade algorithms suitable for upgrading passively replicated servers. Concrete implementations of such algorithms should be derived from this class according to the Strategy[28] design pattern.
- `NonReplUA` defines an abstract class for upgrade algorithms suitable for upgrading non replicated servers. Concrete implementations of such algorithms should be derived from this class according to the Strategy[28] design pattern.

6.2.4.2 Main Classes

The following classes are defined in package `DUF.UpgradeAlgorithms`:

- `NRSUA` defines a concrete class implementing a upgrade algorithms suitable for upgrading non-replicated component instances. The details of the algorithm design and implementation is described in section 7.1.
- `ARSUA` defines a concrete class implementing a upgrade algorithms suitable for upgrading actively replicated servers in Jgroup. The details of the algorithm design and implementation is described in section 7.2.
- `PRSUA` defines a concrete class implementing a upgrade algorithms suitable for upgrading passively replicated servers in Jgroup. The details of the algorithm design and implementation is described in section 7.3.

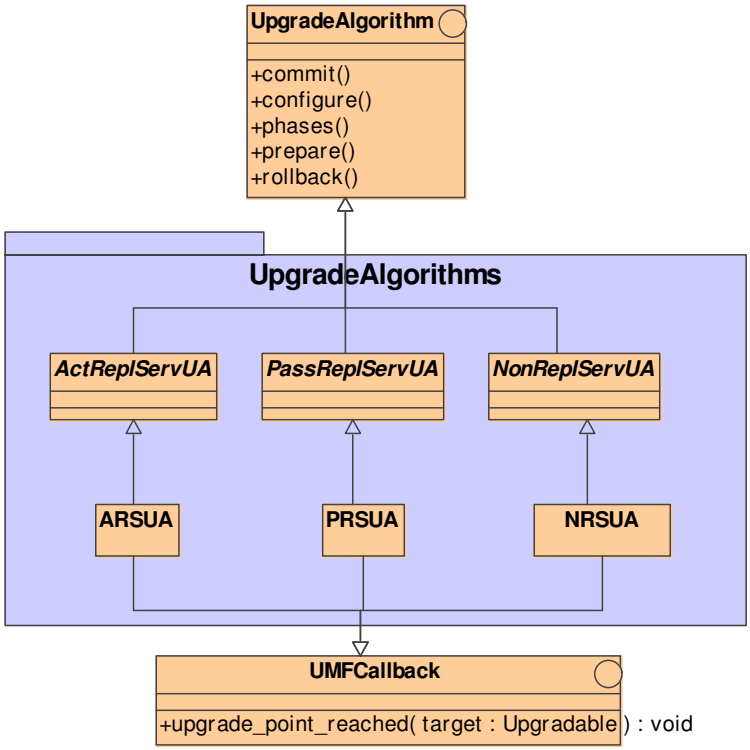


Figure 42. Class diagram for `UpgradeAlgorithms` *package.*

6.2.4.3 Interactions

The package contains definitions of the abstract classes and interfaces that are then extended by concrete implementation of the upgrade algorithms. The upgrade algorithms are usually independent from other algorithms and therefore there is no inter-algorithm interactions. The details of the class interactions specific to an upgrade algorithm are described in sections on the implementation of this algorithm.

6.2.5 Infrastructure package

This package describes classes and interfaces to be implemented by a traditional component-based middleware platform that the DUF needs.

6.2.5.1 Public Definitions

The following interfaces are defined in package `DUF.DUFInfrastructure`:

- `CodeManager` provides methods for managing the code modules. It allows for:
  - Downloading the code modules from the service code repository.
  - Installing the downloaded code into a given container (execution environment) as well as uninstall it from the container.
  - Loading and unloading the installed code into a given instance of a container.
- `ComponentActivator` defines methods for managing life cycle of the components. It offers the following functions:
  - Component activation, which makes the component runtime image active in that its operations can be accessed and instances of the component runtime image can be created in the container.
  - Component deactivation, which makes the component runtime image inactive in that no operations on it and its instances in the container, including instance creation, can be performed.
  - Blocking the invocation dispatching, which makes the container stop dispatching incoming requests and messages to a given component runtime instance. This method is used for making the component instance quiescent to perform the upgrade.
  - Continuing the invocation dispatching, which makes the container (re)start dispatching requests to a given component runtime instance
  - Creation of a component interface reference that can be used for accessing the component from remote software.
  - Assigning a reference to a component runtime instance.
- `Discovery` describes operations for discovering component runtime instances running in the system as well for advertising components so that they can be accessed by other entities in the system.

6.2.5.2 Main Definitions

The following classes are defined in this package:

`DiscoveryPattern` is a class that includes some selected information on the component runtime instance. It is used to advertise the component runtime instance in the system and for its later discovery, where the `DiscoveryCriteria` are matched against.

- `DiscoveryCriteria` is a class that abstracts criteria defining a set of component runtime. It offers a method that evaluates a given piece of information on a component runtime instance.

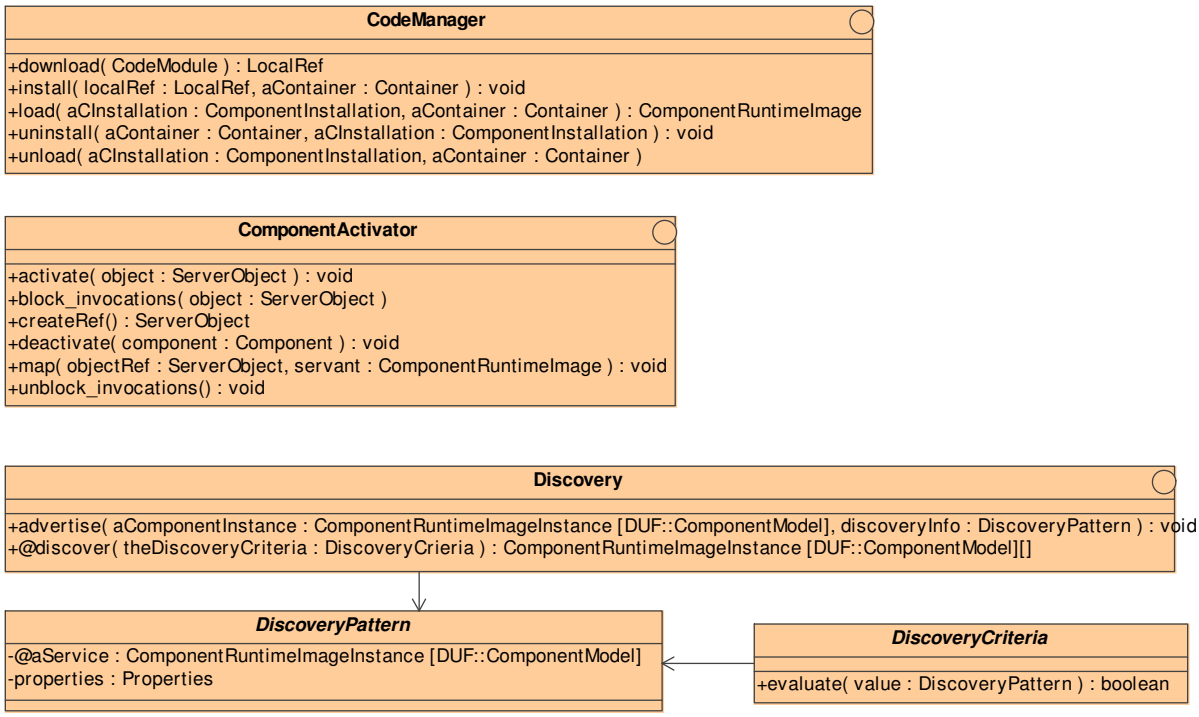


Figure 43. Main definitions of package `DUF.DUFInfrastructure`

6.2.5.3 Interactions

The class interactions within this package are not described here. The interfaces and abstract classes describe typical functionalities of traditional middleware or operating system and thus should be implemented by the underlying middleware platform.

6.2.6 TargetSystem package

This package defines interfaces and classes for communicating with upgradable components. Whereas some interfaces have to be implemented by the upgradable components (`Upgradable`), other are used for the upgradable components when accessing the DUF. The implementation of the interfaces provided by the upgradable components has to be currently provided by the component developer. In future, some automation may be supported by the DUF.

6.2.6.1 Public Interfaces

The following interfaces are defined in package `DUF.TargetSystem`:

- `UpgradeTarget` describes the target of an upgrade. It is a list of tuples (host, container, a list of component instances), where the list contains component instances running in a give container on a host that shall be upgraded.
- `Upgradable` describes the key interface that an upgradable has to provide. This interface defines methods used for making a component quiescent and then transfering the state of the component runtime image instance.
- `UMFCallback` describes an interface to be implemented by the upgrad algorithm. It defines methods for being notified of reaching an upgrade state by the target component.

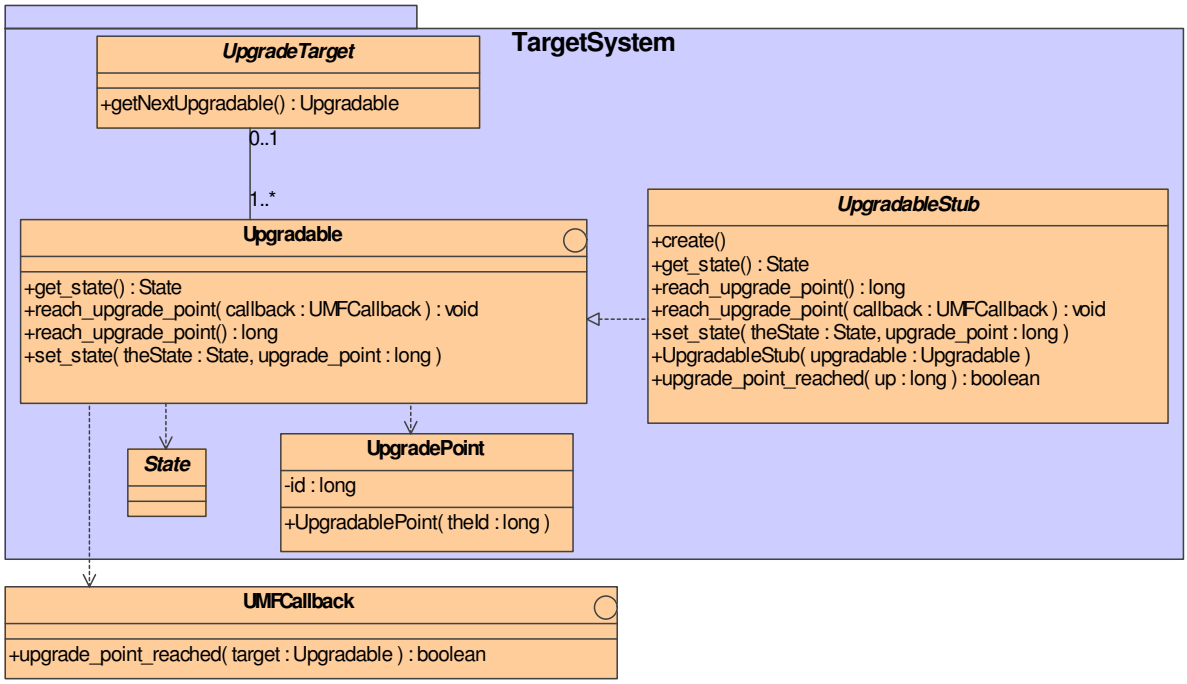


Figure 44. Main classes and interfaces in package DUF.`TargetSystem`.

6.2.6.2 Main Classes

Figure 44 shows the main classes of package `DUF.TargetSystem`. Their details are described below:

- `State` of a component represents the history of component computations. In practice, it is usually expressed by a set of internal variables that describe the results of the computations and by the system variables that describe that execution state. Most of the variables describe or point to the objects co-located with the component. Some other internal variables can be pointers referring to remote components. Managing a transfer of this part of the state is considered a special case of state transfer.
- `UpgradePoint` describes the point in the execution of a component instance in which it is possible to perform a replacement. The points can be determined manually or in a semi-automated way. Potential candidates for upgrade points:
  - Places where threads block (see [37]). Requirements discussed in [37] have to be fulfilled; this approach depends on the language and the OS calls threading characteristics.
  - Places where the control enters/exists programming entities, like procedures, object/classes. Approach depends on the programming language structuring concepts.
- `UpgradableStub` describes a default implementation of the `Upgradable` interface for one-threaded component implementations. The implementation of method `reach_upgrade_point()` saves the request and the callback reference. As soon as it is notified by the component instance of reaching an upgrade point or only a given one by calling method `upgrade_point_notify()`, it forwards this information to the callback. The component instance is blocked in at this upgrade point waiting for the return of the `upgrade_point_notify()` method. See

section 6.2.6.3 for the sequence diagram visualizing these interactions.

6.2.6.3 Interactions

The algorithm allowing for reaching an upgrade point is component specific. In general, it may be a non-instantaneous process that needs completing a certain amount of computations performed by the component implementation. For certain class of component implementations, the framework defines a default mechanism that facilitates decoupling the point in time when a request to reach an upgrade point is issued from the time in point an upgrade point is actually reached. The component implementations have to fulfill the following constraints:

The component implementation is one-threaded.

- The implementation notifies the framework, that is its associated `UpgradableStub`, of reaching subsequent upgrade points as they occur during the component execution.

The time gap between two subsequent upgrade points is bound.

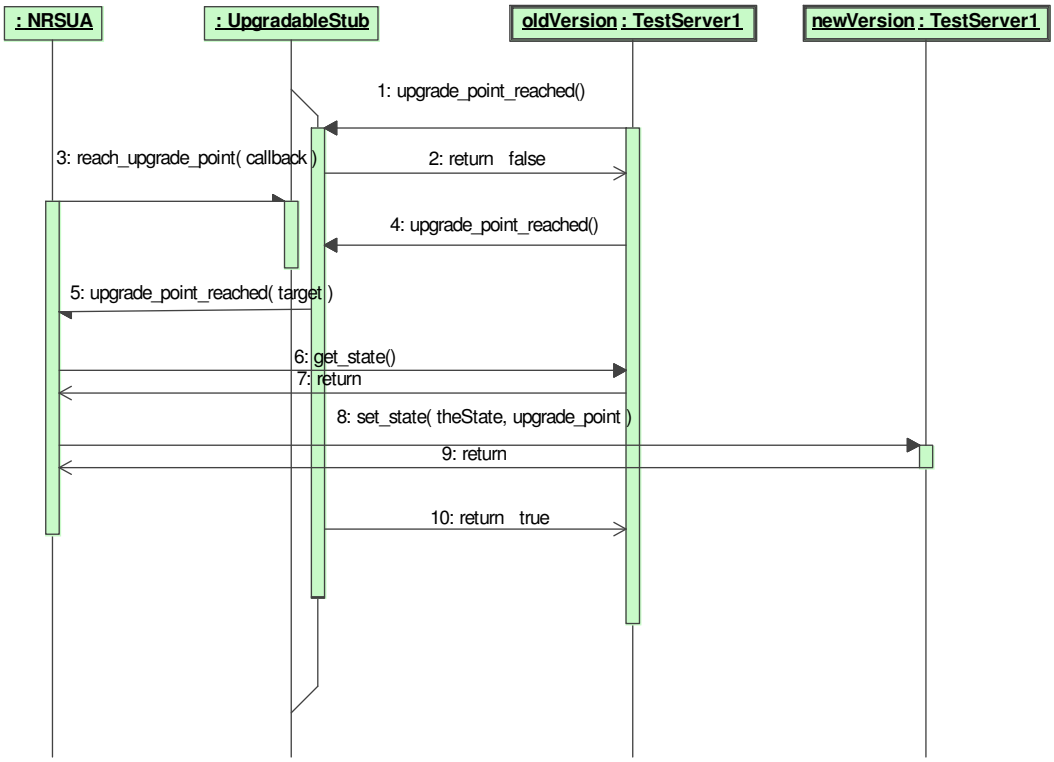


Figure 45. Reaching an upgrade point and a state transfer.

Figure 45 depicts example interactions occurring during the process of reaching an upgrade point as supported by the `UpgradeStub`. The interactions involve the DUF facility represented here by the NRSUA upgrade algorithm, the `UpgradeStub` and two versions of an upgradable component occur in the sequence presented in **Table 8**:

Seq. No.	Interaction description
1.	The old instance of the upgradable component notifies its upgradable stub of reaching an upgrade points whenever it reaches one.

2.	After checking that there are no pending upgrade requests, the `UpgradableStub` returns immediately control to the old instance.
3.	The upgrade algorithm decides the upgrade the component instance and issues an asynchronous request to the component, that is, its `UpgradableStub`. The callback reference is passed at this call.
4.	Whenever a next upgrade point is reached by the old instance of the component, it the `UpgradeStub` is notified calling `upgrade_point_reached()`.
5.	This time the `UpgradeStub` does not return control back to the old instance and instead forwards the notification of reaching an upgrade point to the upgrade algorithm.
6.	Now, the actual state transfer may start. The upgrade algorithm gets the state of the old component instance, which execution is now blocked waiting for the return from invocation 4.
7.	The component state is returned to the algorithm.
8.	The algorithm may transfer this state to the new instance of the component that has been already instantiated. The upgrade point is also passed to bring the component instance to an execution state equivalent to that one returned by the old component instance in interaction 4.
9.	After the new component is set with the state transferred, the state transfer successfully terminates
10.	The execution control is returned the old component instance. The old component is notified of a state transfer and may be deactivated.

Table 8. Reaching an upgrade point with support of `UpgradableStub`.

6.2.7 Policies package

The complexity and variety of parameters determining the way that the process can be performed needs to be expressed in a simple an consistent way. Policy-oriented management provides a good approach to express the multifold aspects of management process, i.e. the policies.

6.2.7.1 Policy Information Model

Upgrade management policies are used to manage the initiation of the upgrade process. They allow to specify or declare a condition under which certain management action related to dynamic upgrade should be carried out.

The policy model used in our solution is based on some of the concepts introduced in the Policy Core Information Model defined by the IETF community[64]. The model is based on the declarative approach in that it does not define either the algorithm to produce a result using the attributes or an explicit sequence of steps to produce a result.

Figure 46 depicts a class diagram with the core classes of the policy information model. The classes represent a simplified policy information model that is suitable for managing dynamic upgrade processes in a distributed system.

A `PolicyRule` is defined by a number of policy conditions, each represented by class `PolicyCondition` and a number of policy action, represented by class `PolicyAction`.

The basic idea of a policy is that a policy rule checks its policy conditions and if any one or all of them, depending whether it is a ORed or ANDed policy rule, is fulfilled, the policy action is triggered by calling its `start` operation.

Policy condition evaluation can be:

- *passive* if it is the external entity to the policy condition that regularly calls `fulfills` operation on the policy condition. This type of policy conditions is abstracted with abstract class `PassivePolicyCondition`.
- *active* if it is the policy condition object itself that notifies the policy manager of the condition being fulfilled. This type of policy conditions is abstracted with abstract class `ActivePolicyCondition`.

Policy rules can be put together in a container `PolicyGroup`. Policies can be added, removed or retrieved from a `PolicyGroup`. Policy groups are used to manage sets of related policy rules.

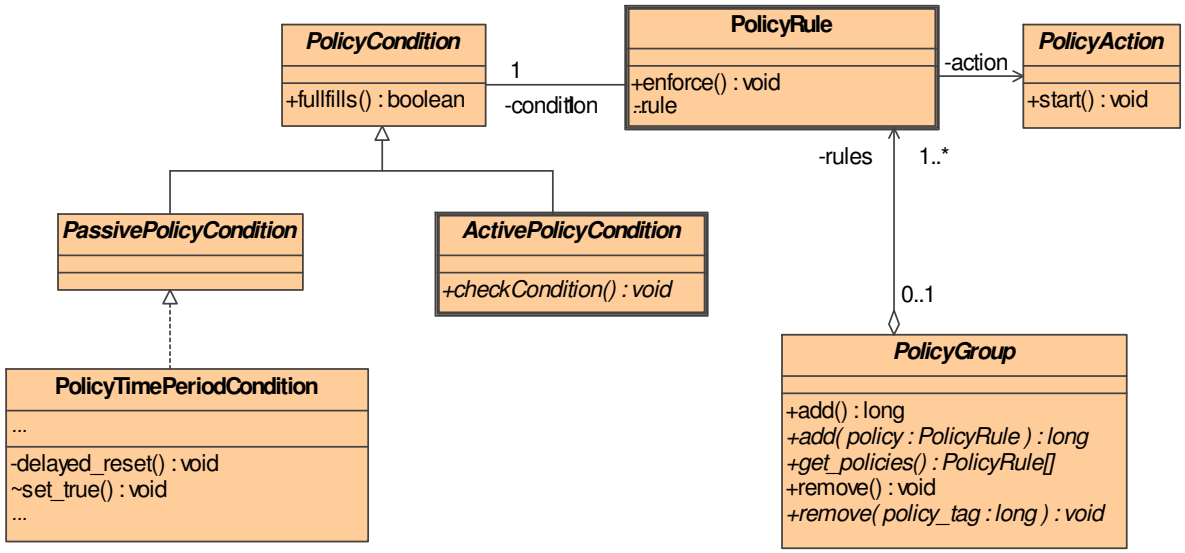


Figure 46. The upgrade management policy model.

6.2.7.2 Policy Types

In the DUMF framework, a few upgrade management policy types are pre-defined. These policies include:

Upgrade Initiation Policy, which defines when an upgrade process should be started. The policy allows defining a set of calendar events, potentially reoccurring, that determine the start of an upgrade process.

Upgrade Completion Policy, which defines when an upgrade process should terminate. If an upgrade is still being carried out, the upgrade process is stopped and cancelled.

- Some other types of policies may include:

Upgrade Failure Policy, which defines how to proceed with the upgrade process that has failed for some reason that cannot be coped with by the upgrade algorithm.

Upgrade Process related Policies. parameter of the upgrade process (e.g. the minimum replication level).

Upgrade Validation, which enables managing the upgrade validation process. Policies of this type define under what condition and how to validate an upgrade process that has successfully terminated.

6.2.7.3 Policy Condition Determinants

The Policy Condition can be determined by evaluation logical expressions applied to the following factors:

- *State of the Target System*. The system state includes the current or predicted availability of system's computational and communicational resources. Typical *examples of such state constraints are system load threshold, request queue length* threshold in a server to be upgraded or simply an calendar event. The constraints may be also defined on the system state history.
- *External factors.* These factors do not depend on the state of the target system and can be evaluated only using some input from external entities. Typical examples of such factors include code availability of software deployed in the target system or an immediate upgrade request issued explicitly by the service deployer.

6.2.7.4 Policy Specification

Various specification methods and formats exist in the literature. In our approach, the following were considered:

- *Type-based Declaration.* It allows expressing the type of the policy and some parameter values determining a specific policy instance of that type. The approach requires that policy types and their parameters be defined. Additionally, some generic code for each policy type is needed which defines the semantics of the policy type and is used to enforce policy instances of the type. XML is a commonly used markup language for policy declaration due to its easy parsing and programming platform independence.
- *Constraint-based format*. The policy rules are specified in a constraint language as a logical expression. If a rule condition is evaluated positively the corresponding policy action is triggered. The actions are specified usually only as names that have to be mapped to program invocation.
- *Manual policy coding in a programming language.* Both the policy condition and the policy action are coded directly in a programming language. The policies are defined according to a implementation mapping of a policy information model are integrated into the policy-based management framework.

In the prototypical implementation of the DUMF framework, the latter policy specification approach was chosen and both policy conditions and actions are coded in the Java programming language. Upgrade management policies implement interfaces and extend some predefined classes that are a result of a Java mapping of the Policy Information Model presented in section 6.2.7.1. The decision was a trade-off between the effort to implement a fully-fledged policy specification mechanism and the time allocated for this task. However, the approach can be easily extended either to the constraint-based or type-based format, if needed.

6.2.7.5 Example upgrade initiation policy

This example shows a concrete example of an upgrade initiation policy. It is assumed that the

Service Deployer wants to perform to check the availability of new version of the component X code on Fridays at 21 hours every week and perform an upgrade of all instances of the component provided that the average system load does not exceed the threshold of 50%.

The example policy can be expressed either using a single policy condition or defining a ANDed policy rule with the following subconditions. Below, the example policy is expressed as a single condition and presented in a simplified Java-like notation[4]:

Listing 3 An example upgrade initiation policy.

```
class ExamplePolicyCondition implements PolicyCondition{

   ExamplePolicyCondition() { … };

   Boolean fullfils() {
     return (today.dayOfWeek() == Calendar.FRIDAY) &&
     (today.time().IsAround(Time(21,00))) && (CodeRepository.hasComponent(X)) &&
     (SystemMonitor.averageSystemLoad() < 0.50);
   }
}
```

6.2.7.6 Policy Management

Policies can themselves be seen as services as they need to be deployed and managed. The policy information model describes their properties. Each policy is identified by its unique name. Additionally, policies are categorized by their type.

Policies are maintained in the DUMF framework in the Policy Registry in a similar way the FAIN ASP maintains active services. Whenever a policy is to be used by the DUMF framework, it needs to be registered with the Policy Repository. DUMF defines a number of predefined policies and they are registered with the Policy Repository at the DUMF bootstrap.

Except for the predefined policies, the DUMF framework enables to install new policies. New policies also need to be registered with the Policy Repository in order to make them visible to the DUMF framework

6.2.7.6.1 Registering a new policy

Whenever a new policy is introduced into the DUMF framework, it has to be registered with the PolicyRepository. In case of the upgrade initiation policy, such a policy is registered using `UpgradeInitiator` interface presented in section 6.2.3.

6.2.7.6.2 Enforcing a policy

As soon as the upgrade initiation policy is registered with the `PolicyRepository`, the latter is responsible for evaluating the triggering condition of the policy. Depending on whether the policy is active or not, it may evaluate the value of the condition expression in regular time gaps, in case of passive policies, or let the policy check the triggering condition by itself in case of active policies. On positive evaluating the policy condition, the upgrade initiation policy is enforced and a corresponding upgrade process started as a result of the policy action being triggered. This situation is described in steps 8-10 in Figure 41.

[4] Note that this notation is only close to implementation of the example policy condition. The framework provides a lot of facility classes that would make the condition coding easier through reuse of paramterized classes.

6.2.7.6.3 Removing a policy

There are two possible ways of removing a policy from the `PolicyRepository`:

- *Removal by Policy Expiry*. One of the policy attributes is its Time To Live. This parameter describes the time validity of the policy. Each evaluation of a policy triggering condition is preceded with a policy time validity check. If the policy is not valid any more, it is removed from the Policy Repository. Otherwise, the evaluation of the policy triggered condition follows.
- *Explicit Removal.* A policy also may be requested to be removed from the Policy Repository. This is done by invoking `remove_init_policy()` on `UpgradeInitiator` interface. In case of a policy that has been enforced, the policy action is stopped and recovered if needed.

6.3 Implementation

This section gives some implementation details. The methodology as well as the selected implementation technology and the developing tools are presented in section 6.3.1. The issues related to coding are the topic of section 6.3.2.The deployment of the DUF component is described n section 6.3.3.

6.3.1 Methodology and Tools

The development process of the Deployment and Upgrade Facility is based on the Unified Process[52] as mentioned in the previous sections of the thesis. Thus, the platform-independent part of the Deployment and Upgrade Facility has been prototyped in parallel to developing the design model in an iterative process. After the design classes are initially conceived , they are used to define the corresponding implementation classes.

In our approach, the Unified Process was supported by the capabilities of the Magic Draw[70] modeling tool. The tool allows building UML models that are used to generate code skeletons in a number of programming classes. We used the Java generator to produce Java implementation classes.

These classes have been then extended by the functionalities described in the model by manual coding. Eclipse[14], an open source Integrated Development Environment, has been used to write and later debug the code. This tool is integrated with Magic Draw so that changes in the class definitions have been reflected back in the design model. In this way, it is possible to update the design model with the changes done in the implementation code and to close the feedback loop as prescribed in the Unified Process.

For the code testing, Junit[47], a testing framework has been used in our developments. The usage of this framework supports writing unit tests of the code both that is to developed and that has been developed. The tests have to be defined manually to check the basic semantics of the classes and subsystems as described in the design model.

6.3.2 Code structure

The DUF framework has been prototypically developed in Java. For the sake of better portability as requested in requirement R5 (see section 5.1.2), the code is divided into two parts:

- *Platform-independent part.* This pat of the DUF code does not depend on any specific middleware platform, in terms of using any interfaces and classes provided by

a concrete middleware platform. Thanks to no dependencies of any platform specifics, it can be ported easily.

- *Platform-specific part.* This part of the DUF code is written for a give middleware platform. It uses the platform specifics and is not easily portable to another middleware platform.

The platform independent code is structured in the Java packages corresponding to the UML packages in the design model presented in section 6.2.1. These packages include: `DUF.ComponentModel`, `DUF.DUMF`, `DUF.DUMF.Policies`, `DUF.Infrastructure`, `DUF.TargetSystem` and `DUF.UpgradeAlgorithms`.

The implementation classes specific to middleware platforms are defined in separate packages. Currently, the prototype includes implementation classes for the Jgroup platform. The functionality related to this platform is collected in package `DUF.Infrastructure.jgroup` and `DUF.TargetSystem.jgroup`.

6.3.3 Component deployment

The design model presented in section assumes that some functionalities have to be located on each system host where upgradable components run and some functionalities may be deployed only node. The functionalities of the first category include packages `DUF.Infrastructure` and `DUF.TargetSystem`. Package `DUF.DUMF` belongs to the latter category. Package `DUF.Algorithms` and the derived ones may be of either category depending whether they are implemented in a centralized or decentralized manner and whether they use remote communication for accessing the infrastructure capabilities or not.

As a result, two types nodes have been identified for the deployment purposes:

- *Management node*, on which components are deployed, which are singletons in the system.
- *System node*, which abstracts a regular node used for deploying components of the target system.

The identified node types and the DUF components assigned to them are depicted in Figure 47. In a typical distributed system, there is one management node and a number of system nodes.

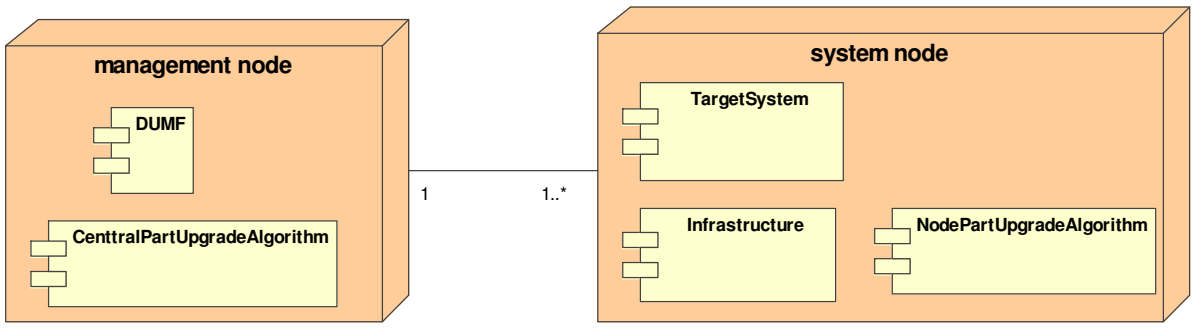


Figure 47. Deployment diagram for the DUF components.

For deployment purposes, the package functionalities are distributed into corresponding physical components. The following components are identified: `DUMF`, `TargetSystem`, `Infrastructure`, `CentralPartUpgradeAlgorithm` and `NodePartUpgrade-Algorithm`. Two latter components comprise the functionalities of package

`DUF.Algorithms` and divide the implementation artifacts of these functionalities into the ones to be deployed once in the system and the ones to be deployed on each node of the target system.

It is important to note that whereas component DUMF is design and implemented independent of the underlying middleware platform, the other components include some platform specifics. These platform specific implementation artifacts are defined in the separate model subpackages of the corresponding DUF packages. Thus, components `TargetSystem` and `Infrastructure` contain implementation classes from these subpackages, as well.

Furthermore, for the sake of framework extensibility, both each of the upgrade management policies and upgrade algorithms is proposed to be packaged in a different component. In this way, it is possible to deploy an upgrade algorithm or a policy independently and on-demand provided that the `DUMF` and `Infrastructure` components are available in the system. Whereas management policies are deployed always onto the management node, the upgrade algorithms may be deployed both onto some system nodes and the management node. Hence, it is enough to package a policy into only one component whereas an upgrade algorithm may need two components.

An example deployment of DUF components is shown in Figure 48. The deployment diagram depicts only the dynamic components of the DUF facility, that is the component that may be deployed on demand. One upgrade initiation policy is deployed in the management node and its runtime instance in running as part of the DUF. An upgrade algorithm is also deployed in the system. Whereas one part is running in the management node, the other algorithm components are deployed in all the system nodes named `brunali`, `devore` and `dividices`. Though basic components of the DUF are not shown in the diagram for the sake of simplicity, they are deployed on the corresponding nodes as depicted in Figure 47. This example deployment configuration has been used in the practical experiments described in chapter 8.

.

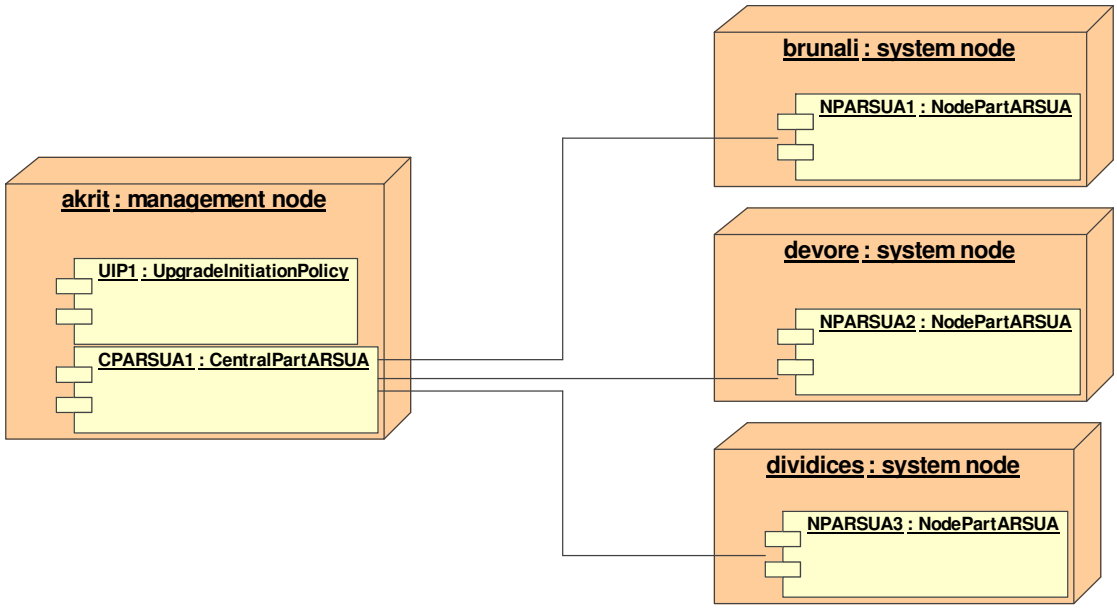


Figure 48. An example deployment of the dynamic DUF components.

6.4 Summary

In this chapter a framework for managing dynamic upgrades in distributed systems was presented. This framework allows for policy-based management of upgrade process in a distributed system. In particular, it enables:

- Defining upgrade management policies determining the parameters of upgrades including the upgrade triggering condition.
- Managing these policies so that they can be added or removed on demand.
- Evaluating policy conditions at system's run time.
- Enforcing upgrade management policies by triggering corresponding policy actions.

With regard to the requirements stated in chapter 5.1, the following of them are fulfilled by this part of the Dynamic Upgrade Facility:

- **Automated Upgrade Management** (R12) is realized by inserting management policies by the service deployer. The dynamic upgrade management framework automatically evaluates the policy conditions and autonomously takes management actions related to upgrades. The upgrade process is automated in that system determines which upgrade algorithm is to be applied to the given upgrade target and may perform corrective actions to recover from failures to a certain degree without human interventions.
- **Support for multiple simultaneous upgrades** (R13). The system supports running multiple upgrade process and their coordination if needed. To achieve that a number of upgrade management policies have to be defined, each of which may be concerned with one upgrade process. These policies may describe when to start an upgrade process

7 Performing Upgrades: The Algorithms

This section is concerned with the algorithm allowing for replacing an instance of a component at runtime. Several variants of the algorithm are briefly presented that are applied to components with different replication characteristics. The variants comprise an upgrade of:

- Non-replicated component,
- Actively replicated component, and
- Passively replicated component.

The following subsections describe the design details of these algorithm variants.

7.1 Non replicated component

In this variant, the component to upgrade is not replicated, i.e. there is only one copy of a component instance. Performing upgrade of the component requires some break in the component activity so that a part of the system functionality is not available for some time.

7.1.1 System Assumptions

Concerning the system model, it is assumed that:

A1. The code modules of the target component are available and can be downloaded and installed in the environment where the old runtime instance of the component is running.

A2. The component state is transferable and the execution state can be described using the concept of upgrade points (see section 6.2.6.2).

A3. The component reaches an upgrade point in a acceptable period of time.

7.1.2 Algorithm Overview

The basic idea of the algorithm is intuitive: the running instance of the old component version is replaced with a instance implementing the new code so that the component state is transferred from the old instance to the new one. The process is instantaneous from the point of view of the rest of the system that depends on the target component.

This algorithm variant has the following steps:

1) Install the new version of the component in the same runtime environment.
2) Deactivate the old component, transfer the state, rebind the connections.
3) Activate the new version of a component.
4) Uninstall the old version of the component.

The component is unavailable during step 2. This break may be short but exists always in this variant. Two instances of the component, the old and new version cannot be running at the same time since (Figure 49):

- the state of the old component cannot be changed after the state is sent to the new component
- the new component cannot be activated and process incoming requests before it is synchronized with the state of the old component.

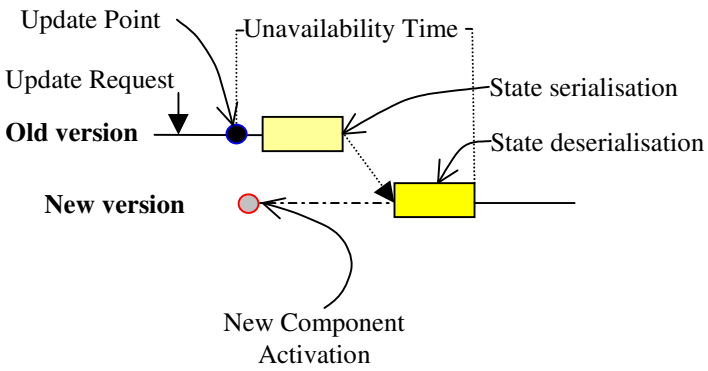


Figure 49. Upgrade process of a non-replicated component

7.1.3 Algorithm Design

The design of the upgrade algorithm integrated in the facility is presented in Figure 50. The algorithm logic is represented in class `NRUA` (a short for Non-Replicated Server Upgrade Algorithm) which implements interface `UpgradeAlgorithm` described in section 6.2.4. The definition of the algorithm is kept in the algorithm repository and is used by an object of class `UpgradeProcess` when the latter is initiated. A concrete instance of the upgrade algorithm is associated with an instance of class `UpgradeProcess`.

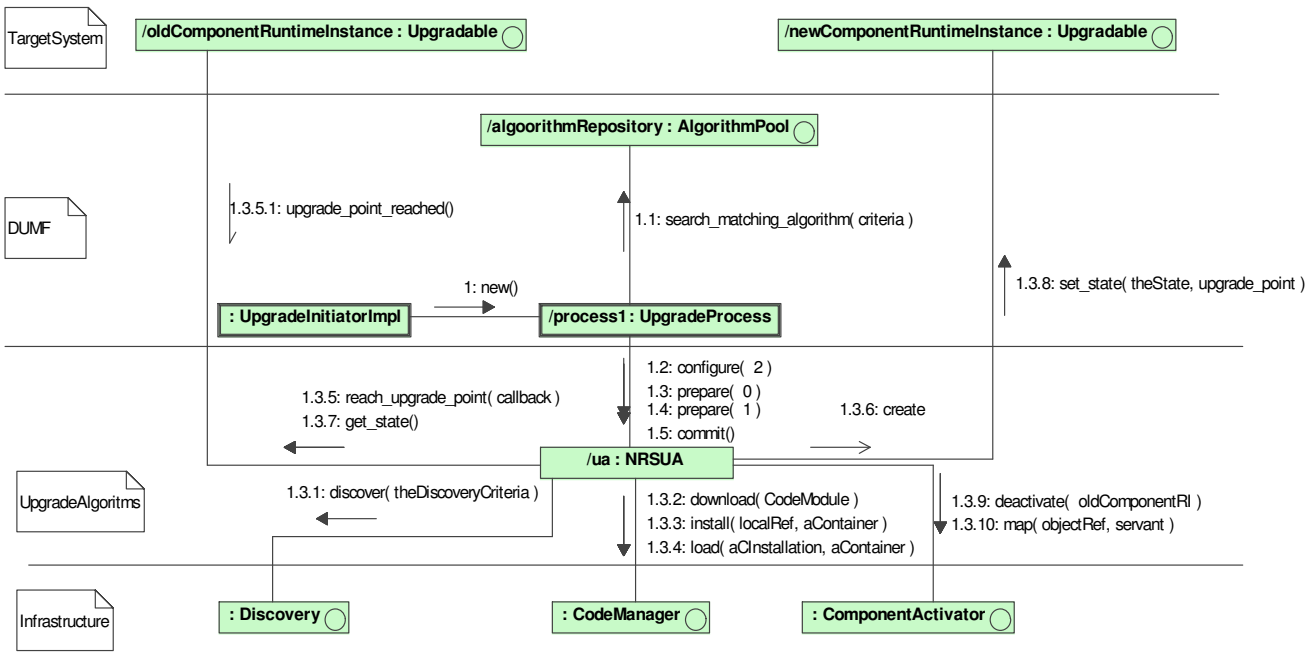


Figure 50. Coordinating an upgrade process.

Figure 50 describes the algorithm logic in terms of interactions between the classes involved defined in packages DUMF, `DUFInfrastructure` and `UpgradeAlgorithms`. The class interactions are numbered as in the collaboration diagram depicted in Figure 50.

Seq. No.	Interaction Description
1.	As soon as the triggering condition of an upgrade initiation policy is positively evaluated by the `UpgradeInitiatorImpl` , a corresponding upgrade process is triggered. The upgrade process is represented by a class `UpgradeProcess` and such an object is created.
1.1.	The upgrade process searches for a suitable upgrade algorithm for its upgrade target. It contacts the algorithm repository implementing interface `AlgorithmPool` and requests an algorithm matching the given criteria. The search criteria include information on the upgrade target type. In this case, it is an algorithm of class `NRUA` that is returned from the repository.
1.2.	The process configures the returned instance of the `NRUA` algorithm providing the necessary details on the upgrade target.
1.3.	The algorithm is started by triggering its first phase.
1.3.1	In this phase, the algorithm discovers the location and access details of the upgrade target contacting `Discovery` service offered by the Infrastructure.
1.3.2	The algorithm requests the old runtime instance of the component to upgrade to reach an upgrade point and get a notification on that.
1.3.3	The new code of the component is downloaded to the location where it is going to be instantiated. It means typically the same location where the old code has been running.
1.3.4	The code downloaded is installed in the target container, the execution environment using Infrastructure's `CodeManager`.
1.3.5	Additionally, the fetched code is loaded into a running instance of the target container.
1.3.2.1	As soon as the old component instance reaches an upgrade point, a notification is sent to the upgrade algorithm through interface `UMFCallback`.
1.3.6	Now the algorithm may create an instance using the new component code.
1.3.7	It may perform the state transfer accessing the state of the old runtime instance of the component.
1.3.8	The state is passed then to the newly created instance of the new version of the component.
1.3.9	The old component may be deactivated, removed so that the resources it consumed are returned to the container and the old code unloaded and uninstalled.
1.3.10	The algorithm finally updates the dispatching information in the container so that the client requests aimed at the old component instance are directed to the new component instance.
1.4.	After the last (and the first one) phase terminates successfully, the upgrade process can signal a commit. No rollback is possible from that point in time on. The upgrade target has been successfully upgraded and the new component runtime instance processes client requests instead of the old one.

Table 9. Class Interactions during the NRSUA algorithm.

7.1.4 Discussion

The presented algorithm allows for upgrading a single instance of a component. The algorithm is based on a simple replacement of an old component instance with a new one using the support mechanisms provided by the Infrastructure. The algorithm does not support fault tolerance as the component instances are assumed to be co-located in the same container, which is considered the smallest unit of failure. This algorithm reduces the system availability by making the component unavailable during the replacement phase.

To overcome this drawback, the system needs some kind of redundancy. A common form of redundancy in distributed object-based software is the technique of object replication. The following sections describe upgrade algorithms for replicated upgrade algorithms.

7.2 Upgrading an actively replicated object

A different variant of the upgrade algorithm can be applied in an environment in which a component to upgrade is replicated in the active scheme. In this case, a number of component replica are running in parallel and processing incoming requests at the same time upgrading its state.

A straight-forward solution would be to could block processing requests by all the replicas, replace all the replicas at the same time and reactivate the upgraded replicas. The big advantage of this algorithm is that is reduces unavailability of the system, as the component is not available during an upgrade of all its replicas. A better algorithm that reduces the unavailability of the component is outlined. The presented algorithm assumes that the state transfer process during the upgrade is much longer than the time needed to resynchronize the upgraded replicas as well as the one that was used to serialize its state, with the active ones after a time in which an upgrade is performed. The rationale is here that a state transfer between different versions of the component is a time consuming operation, whereas a synchronization of replicas can be achieved by processing the requests that were received during the state transfer.

The sections below present the details of the upgrade algorithm for actively-replicated objects [113]. Section 7.2.1 gives a brief overview of the system model for the upgrade target. Section 7.2.2 presents the assumption on the system, section 7.2.3 gives the basic idea of the algorithm. Section 7.2.4 presents the details of the algorithm design and finally section 7.2.5 concludes with a brief analysis of the algorithm.

7.2.1 System Model

In the context of this paper, the software system is modeled as an actively replicated server [99], processing requests from multiple clients, see Figure 51. The clients send their requests by means of a group communication system which guarantees reliable broadcast of their requests to all the replicas of the server, collectively called a group. After the client requests are delivered by the GCS in a total order, each replica processes them in a deterministic way. The GCS collects the replica responses, agrees on the final response and sends it back to the corresponding client.

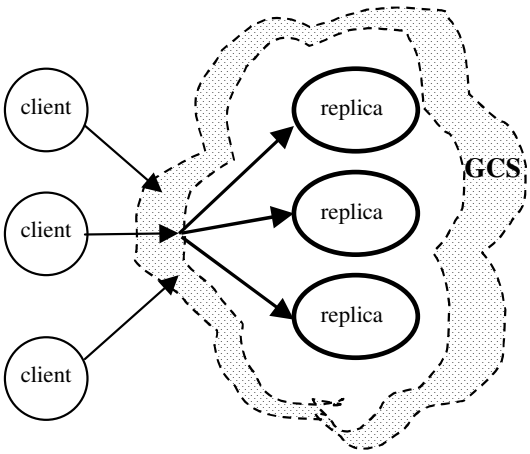


Figure 51. The state machine system model.

7.2.2 System Assumptions

The system model presented above is used for further investigations. In this section the assumptions of the model are explicitly described and extended with additional assumptions made on the upgrade algorithm.

Concerning the system model, it is assumed that:

A4. server computations are deterministic.

A5. server has a state that is transferable.

A6. server is actively replicated with level n.

A7. clients access the server functionality sending requests and receiving responses.

A8. replication transparency to the clients is provided by the Group Communication System.

A9. GCS is capable of detecting crash of replicas.

A10. GCS supports state transfer between the replicas in the server group.

Additionally, the following assumptions on the replication management are stated:

A11. GCS is extended so that it can recover from crashes by starting a replica on an available host and making it join the server group.

A12. Recovery policy is defined and enforced by the extended GCS.

Finally, the assumptions on the upgrade algorithm itself are proposed:

A13. Server upgrades are atomic with respect to each other, i.e. two upgrade processes cannot interleave.

A14. Replica replacement makes the replica temporarily unavailable.

A15. Replica code is replaceable while it is not processing a user request, i.e. an upgrade is possible before a replica starts, and after it finishes processing a client request.

A16. The algorithm has to tolerate replica crashes during the upgrade.

A17. The system resources are limited. In particular, it is not feasible to substantially increase the original replication level while upgrading the system.

With respect to upgraded software, a new version of the software is supposed to be substitutable with the old one. In particular, it is required that:

A18. the new software accepts the input acceptable by the old version. In particular, the new software offers a compatible interface to the old one;

A19. for any input acceptable by the old software, the new one responds with the same output as the old software would do.

7.2.3 Algorithm Overview

The following section describes the basic idea of the algorithm allowing to upgrade an actively replicated server.

The algorithm is based on the idea that to upgrade an actively replicated object it is enough to upgrade each of its replicas in a sequence. The number of replicas that can be upgraded in parallel depends on the availability requirements, i.e. the minimum replication level.

The steps of such an algorithm could look like this:

1) An upgrade request is reliably multicast to all the replicas of a server to upgrade. A set of replicas to upgrade consists of all the replicas of the server to upgrade.
2) A candidate to be upgraded first is chosen.
3) An algorithm checks whether the actual replacement is possible.
   a) If so, the candidate replica is (stopped and) replaced with its new version. Otherwise, the replica processes client requests and its replacing is postponed until it is possible. At the same time, the rest of the replicas are available to process the client's requests.
   b) After an upgrade of the replica, the state of the new replica state is updated with the state of the active replicas.
4) The upgraded replica is removed from the set of replicas to upgrade.
5) Steps 2-4 are repeated until all the replicas are upgraded.

7.2.4 Algorithm Design

Following the basic idea of the algorithm, a more elaborate design of the algorithm is presented.

7.2.4.1 Algorithm Core

The algorithm is designed in a distributed way i.e. without a global coordinator. All the server replicas perform the same algorithm and are symmetrical in this sense. Figure 52 illustrates the steps of the elaborated design of the algorithm from the view point of a single replica. For the sake of better readability, the upgrade algorithm is depicted in state-oriented way using a SDL notation. This is one of exceptions of not using the UML notation done in this thesis.

After the replica is initiated, it enters its `idle` state, i.e. it is neither processing a client request nor being upgraded. If it receives a client request, it enters `processing` state and after it is `done` it returns to the `idle` state. If it receives an upgrade request, the upgrade process starts.

The replica sets the `triggered` flag to true and enters `idle_upgrade` state. While in this state, the replica may switch temporarily to the processing state whenever a `client_request` is received. Alternatively, the replica may leave this state when a condition `enabling_upgrade` holds.

This condition is a conjunction of two basic checks:

- *is it this replica's turn to start the actual upgrade?* This check can be reduced to a selection of a different replica at each check only once so that any resulting sequence of selections is permutation of replicas in the group. This can be realized through a voting process, in which each replica returns an element of a ordered set, like integer numbers. The selection criterion can be based on the order relation so that the replica with a smallest/greatest number wins. An example of a set matching the requirements above is the set of replica identifiers which are unique and typically realized as integer umbers.
- *can the upgrade of this replica can be performed at the moment?* Hereafter, this condition is expressed in more accurate form by checking whether the current replication level is greater than a given number `r`, `r>1`.

The enabling condition is evaluated regularly and as soon as it becomes fulfilled, the replica continues with the upgrade procedure.

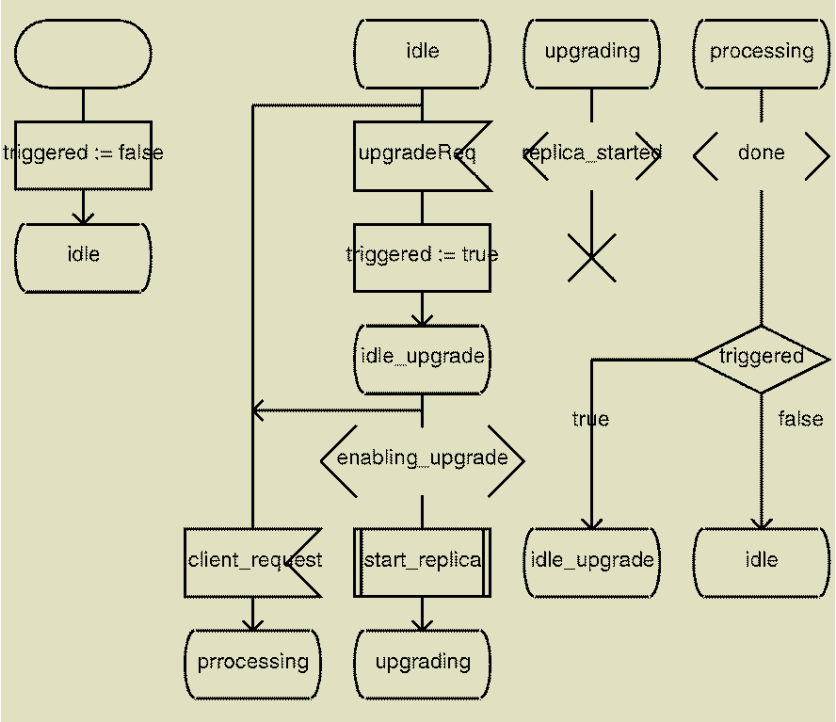


Figure 52. The upgrade algorithm from the viewpoint of a replica.

The replica initiates (asynchronously) a process of starting a replica that replaces this one and enters the `upgrading` state waiting for the success of the operation.

Operation `start_replica()` cooperates with the underlying layers of the system, including GCS, to start a replica with the new version of the code. The operation terminates before the new replica is started. However, it has to guarantee that the replica is started and joins the server group. Otherwise, the upgrade algorithm does not terminate. The GCS is also responsible for transferring the current state of the server group (determined by the rest of active replicas) to the new replica. Now the old replica can leave the group and terminate.

7.2.4.2 Algorithm Integration into DUMF

Figure 53 shows a scenario in which an actively replicated server with two replicas is being upgraded. Hereafter. it is assumed that:

- Minimum replication level is one.
- No crashes occur during the upgrade process.

The figure shows a sequence diagram with invocation calls between the objects involved in performing steps of the upgrade algorithm. The upgrade algorithm, which has been triggered by the DUMF in a suitable point of time, consists of a central object `Algorithm1` registered with the DUMF and a number of local algorithm proxy objects, `LocalARSUA`, running on each of the nodes on which the replicas of the upgrade target are running. These objects perform the steps of the algorithm presented in Figure 52. Additionally, the diagram shows the algorithm interactions with the `DUFInfrastructure`, representing the middleware platform. In the scenario, the algorithm makes use of the middleware service to block incoming requests to the objects being upgraded as well as to activate a new replica and bind it to the address of the replica to replace.

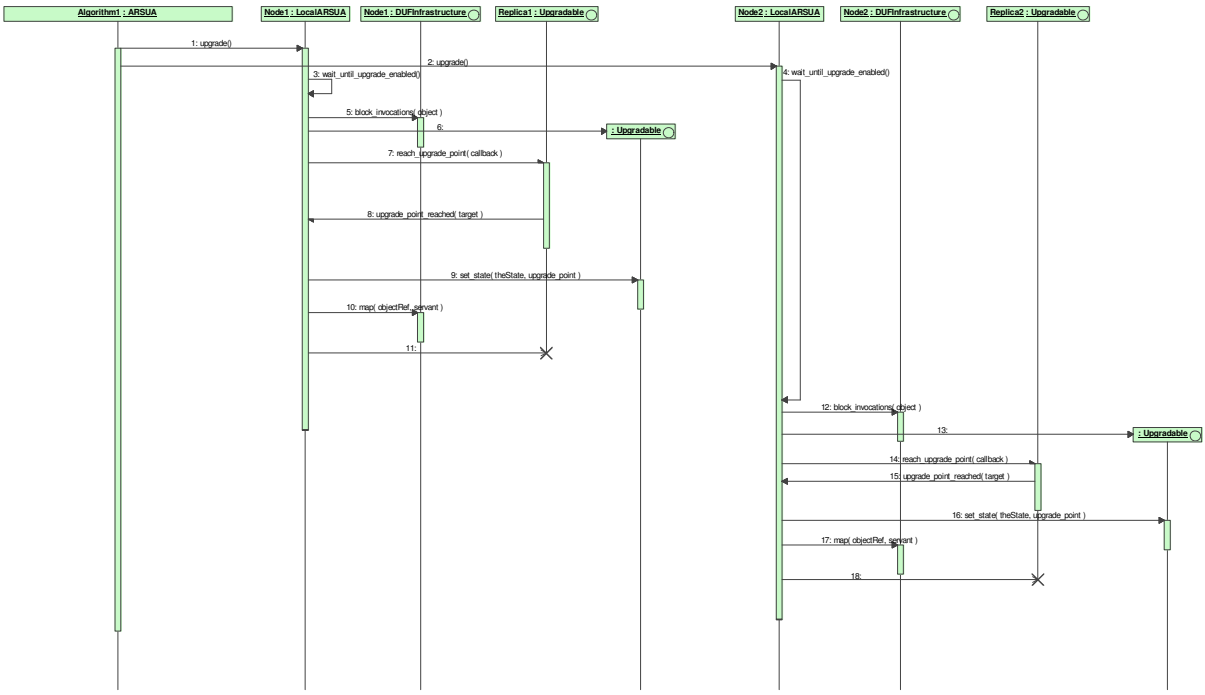


Figure 53. The upgrade algorithm at work: scenario with 2 replicas and no crashes.

7.2.5 Discussion

An upgrade algorithm for an actively replicated server has been presented above. It is to be noted that the algorithm requirements listed in section 5.1.2 are met by the algorithm design as follows:

Regarding *Automating the Upgrade Process* as requested in R6, the algorithm is *triggered on demand* by a sending an upgrade message using the basic system communication mechanism. The algorithm is triggered by a single upgrade request that is reliably distributed to all replicas of the server. This way of starting the algorithm allows for adding some automated upgrade management facility as requested in R12 in section 5.1.5.

The algorithm preserves the system consistency during the upgrade, as requested in R7, in that it allows for state transfers that are handled by the underlying GCS services. One of the GCS services is the state merging service whose task is to transfer the group state to the replicas joining the group. As the replicas with the new component implementation join the group

according to the algorithm, their state is automatically upgraded to the current group state.

Regarding the minimizing of the loss of the system functionality, as requested in R8, the granularity of upgrade is on the level of service components. As the upgrade target is operational at upgrade time, the system functionality is not lost. The upgrade may impact the performance of the system but it does not cause some part of the system to be not operational.

Regarding the minimizing the unavailability periods, as requested in R9, the algorithm keeps a number of replicas of the upgrade target operational all the time during the upgrade. Thus, the upgrade process does not cause parts of the system to be unavailable.

Regarding dependability of the upgrade as requested in R10, the algorithm has been designed to be *self-stabilizing*[12], i.e. can automatically recover following the occurrence of failures. The algorithm is self-stabilizing in that it is decentralized and tolerates replica crashes. As there is no single entity that controls the process of the algorithm, the algorithm continues in presence of transient crashes of replicas being upgraded. Additionally, other crashes are tolerated by the extended GCS enforcing a recovery policy.

The algorithm supports the upgrade transparency, as requested in R11. At upgrade time, the parts of the system that are not upgraded, may access the functionality of the software component being upgraded and after the upgrade is completed in the same way it was before the upgrade. This is due to the fact that the new replicas join the same replica group and can be accessed using the same group identifiers.

The presented version of the algorithm by itself does not guarantee that the replication level is hold after the upgrade. There must be an additional supervising mechanisms that takes care of this.

7.3 Upgrading a passively replicated object

This section presents the initial design of the upgrade algorithm for passively-replicated objects. The basic idea of the upgrade algorithm for a passively-replicated server is quite similar to the upgrade algorithm presented in section 7.1.1. For this the description is focused on the differences.

Section 7.3.1 gives a brief overview of the system model for the upgrade target. Section 7.3.2 presents the assumption on the upgrade target and the upgrade algorithm, whereas section 7.3.3 gives the basic idea of the algorithm. Finally, section 7.3.4 concludes with a brief analysis of the algorithm properties.

7.3.1 System Model

In this section, the software system is modeled as a passively replicated server, processing requests from multiple clients, see Figure 54. A server is represented by a group which is a set of replicas running on different nodes. The group comprises:

- The **primary replica**. This replica receives the requests sent by the clients, processes them and sends the response back. Whenever a request is processed, the primary distributes its current state (or a difference compared to the one it had before the last request arrived) to all the backup replicas.
- A number of **backup replicas**, which do not process the client requests. They upgrade their state with the intra-group messages sent by the primary.

The clients send their requests by means of a group communication system (GCS) which guarantees reliable transport of their requests to and the corresponding responses from the primary replica of the server. In contrast to the active replication scheme, GCS does not have

to guarantee a total order delivery. Neither has the replica to process the requests in a deterministic way.

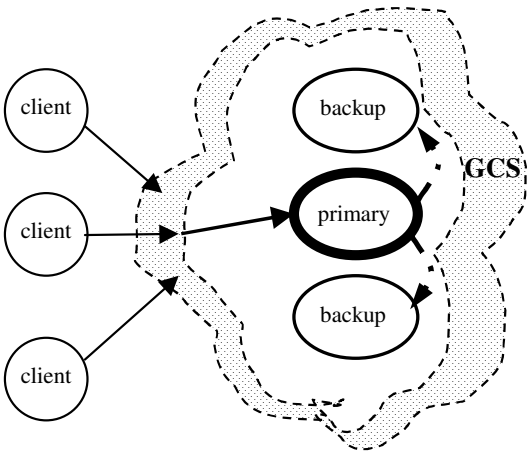


Figure 54. The primary-backup system model.

7.3.2 Algorithm Assumptions

The objective of the upgrade algorithm is to upgrade an actively replicated server, i.e. substitute the executable code of the running processes of server replicas with another version of the executable code.

Having in mind the general objectives of our framework enabling dynamic upgrades from chapter 1 , more detailed requirements on the upgrade algorithm are stated:

R1. The algorithm is triggered on demand by a sending an upgrade message using the basic system communication mechanism.

R2. The algorithm is self-stabilizing [12], i.e. can automatically recover following the occurrence of failures.

R3. The algorithm preserves the system consistency by enabling state transfer from the old version to the new one in case of stateful servers.

7.3.3 Algorithm Overview

In the environment with passive replication, a component is represented by its primary replica processing the requests and the backup replicas regularly upgraded by the primary. An upgrade of a replicated server may include the following steps:

1) Start a replica with the new code in the backup mode B_{n+1}.
2) For all the backup replicas, perform an upgrade of each replica B_i, where $0<i<n$
    - Start B'_i replica with the new code and make it join the group in passive mode
    - Remove the old replica B_i.
3) Force a fail-over to activate one of the backup replicas and shut down the old primary.

Figure 55 shows this version of the upgrade algorithm. The blue rectangles in the picture represent an exchange a replica with its new version. This exchange can be realised as it is described in section 7.1.

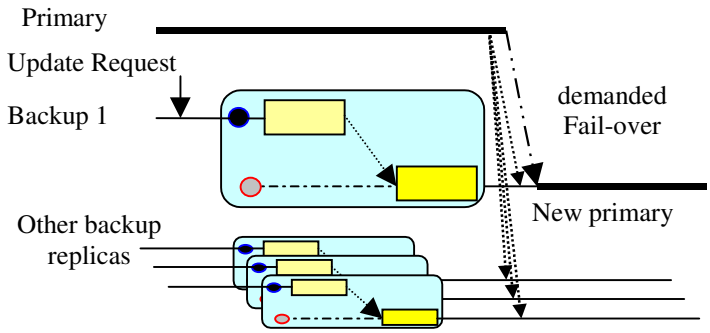


Figure 55. Upgrade process of a passively replicated component

7.3.4 Discussion

An upgrade algorithm for a passively replicated server has been presented. It is to be noted that the algorithm requirements listed in Section 7.3.2 are met by the algorithm design as follows:

R1. The algorithm is triggered by a single upgrade request that is reliably distributed to all replicas of the server.

R2. The algorithm is self-stabilizing in that it is decentralized and tolerates replica crashes. As there is no single entity that controls the process of the algorithm, the algorithm continues in presence of transient crashes of replicas being upgraded. Additionally, other crashes are tolerated by the extended GCS enforcing a recovery policy.

R3. The algorithm allows for state transfers that are handled by the underlying GCS.

7.4 Implementation

One variation of the dynamic upgrade algorithm, suitable for upgrading actively-replicated servers as described in section 7.1.1, has been implemented in the context of Jgroup/ARM. an object-oriented middleware platform. Sections 7.4.1, 7.4.2 and 7.4.4 give an overview of the Jgroup/ARM platform. Section 7.4.3 describes the key concept of the layer which allows for the platform extensions. The Upgrade Layer is described in section 7.4.5 and is the core of the algorithm implementation. The Upgrade Manager is briefly sketched in section 7.4.6. The conclusions on the algorithm implementation are presented in section 7.4.7.

7.4.1 Underlying middleware platform

Jgroup[63] is an extension of the Java distributed object model supporting the group communication paradigm. This middleware platform is aimed at supporting the development of reliable and high-available distributed applications in partitionable environments. Jgroup enables the creation of groups of remote objects that cooperate towards some common goal using a partitionable group communication service. Remote object groups simulate the behavior of standard remote objects by implementing a set of remote interfaces and by enabling clients to remotely invoke the methods defined in these interfaces through the

standard Java RMI mechanism. From the implementation view point, Jgroup is an extendable object-oriented framework that can be extended through adding so called layers to the Jgroup protocol stack.

ARM[59] is a replication management facility built on top of Jgroup to simplify implementation. It handles both replica distribution, according to an extensible distribution scheme, as well as replica recovery, based on a group-specific policy in an autonomous way, i.e. without a need for the manual interaction of the system manager. Additionally, ARM offer a correlation mechanism to collect and interpret failure notifications from the underlying group communication system.

The implementation of the upgrade algorithm extends the ARM framework with the capability of upgrading actively replicated server objects.

7.4.2 Jgroup/ARM Architecture

Figure 56 shows the core components of the Jgroup/ARM framework and its extensions to support dynamic upgrade. A brief description of the core components are given below.

- *Execution Daemon*, ED for short, must be running on all hosts in the system that should be able to host application replicas. The execution daemon is used by the replication manager to create and remove replicas on remote hosts.
- *Replication Manager*, RM for short, is the main component of the ARM framework and its tasks include, replica distribution, failure recovery and interaction with client management applications through the replication manager interface. This component is replicated for fault tolerance, as shown in Figure 56.
- *Upgrade Manager*, UM for short, effectuates upgrade group requests, communicated to it by an upgrade management client. It is naturally co-located with the RM to exploit its database of available groups.
- *Dependable Registry*, DR for short, is a replicated naming service. It enables a dynamic set of replicated remote objects to register themselves under the same name, forming an object group, which can later be retrieved by clients. This enables clients to communicate with the whole group as a single entity. Also the DR is co-located with the RM, since the RM depends on DR for bootstrapping.
- *Application Replica*, R for short, provides the actual service functionality that may be upgraded. The application replica may make use of various services provided by Jgroup by specifying a layer stack, as presented in the next section.

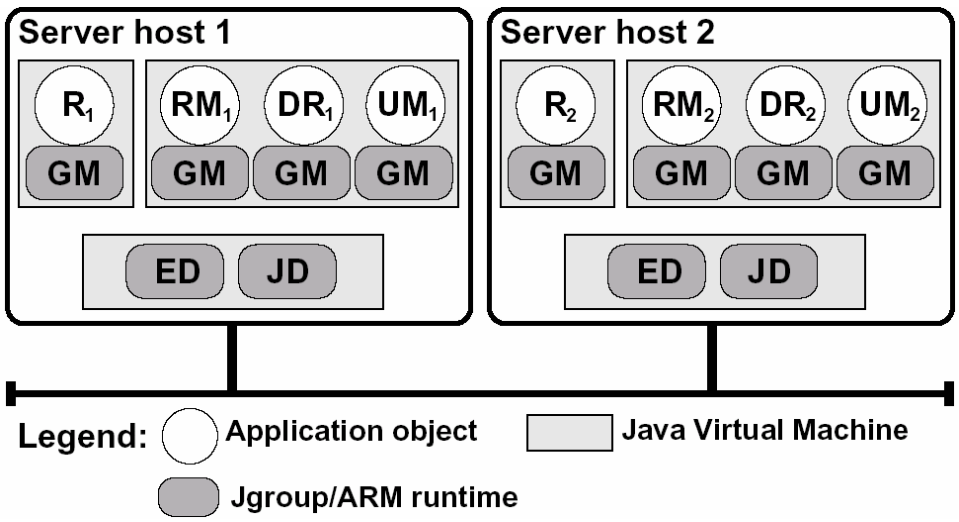


Figure 56. The architecture of Jgroup/ARM

7.4.3 Layers

The Jgroup is designed using a *layered architecture*. A *Layer* in Jgroup is a basic building block that interacts with other layers of the system tin the following way:

- It gets the data input from the lower layer, processes the data and forwards it to the upper layer(s) (`Member` interface), and
- It gets the control data from higher layers and forward it to the lower layers (`GroupManager` interface)

A Jgroup Layer in not a software component as it is understood in this thesis (see section 2.3) as it is not a unit of deployment and management. The definition of the layer is concerned with its interaction paradigm only.

7.4.4 Group Manager

Jgroup *Group Manager*, GM for short, supports dynamic creation of group communication layer stacks, based on a layer stack ordering string associated with each application. The configuration of the layer stack can be expressed in XML, as shown in Listing 4, allowing each application to be configured according to its needs for various Jgroup services, such as recovery, upgrade, group membership and group method invocation services. Each GM layer may interact with any other GM layer, through an interface that each layer exports within the stack.

The *Jgroup Daemon*, JD for short, implements the basic group communication facilities such as failure detection, group membership and multicast, and each application specific GM layer may also communicate with the JD component to perform its tasks.

Listing 4 Example application specification for Jgroup/ARM

```
   <Application name="UpgradableServer" group="103">
   <Class name="test.upgrade.UpgradableServer" args=""/>
   <LayerStack order="PGMS:EGMI:Recovery:Upgrade"/>
   <RecoveryStrategy name="KeepMinimalInPartition">
   <Redundancy initial="3" minimal="1"/>
   </RecoveryStrategy>
</Application>
```

As shown in Listing 4, the `UpgradableServer` application use the PGMS, EGMI, Recovery and Upgrade layers. The PGMS is the group membership service provided with Jgroup; it supplies application replicas with information about the current view of the object group. The EGMI layer is an external group method invocation service, enabling clients to communicate with the entire object group as if it was a single entity. This means that the `UpgradableServer` will export an interface to its clients, enabling them to invoke the server group with what the client sees as a single method invocation. The `RecoveryLayer` also used in the example is a group manager layer that is part of the ARM framework. It is used in conjunction with the RM to ensure that all applications maintain a minimal redundancy level, as specified in Listing 4.

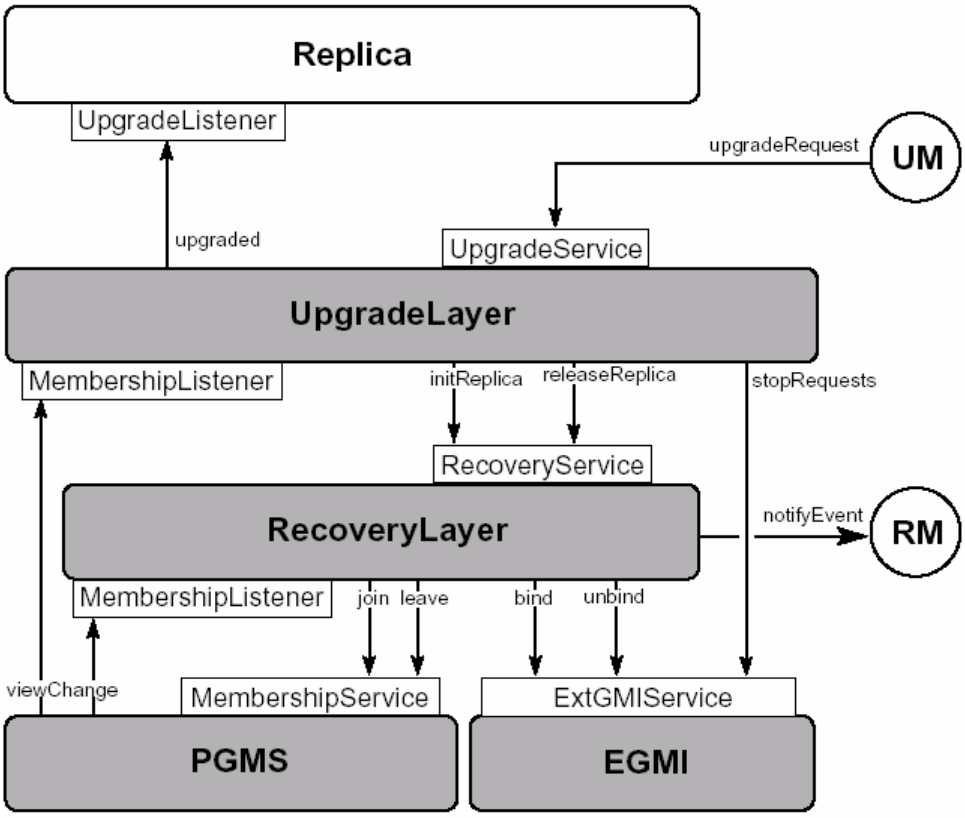


Figure 57. Upgrade layer stack

7.4.5 Upgrade Layer

The upgrade algorithm, as described in section 7.2.4, has been implemented in Jgroup as a layer, called *UpgradeLayer*. Figure 57 illustrates the layer composition and interfaces supported by each of the layer. In the stack protocol configuration, the UpgradeLayer is the last component in the stack and interacts with the application components directly. For an application replica to be upgraded, it must implement the `UpgradeListener` interface (the `upgraded()` method.) The `upgraded()` method is used by the upgrade layer to notify the replica that a new version has been installed, and that the replica may now gracefully shutdown.

Prior to upgrading a particular application, it must first have been installed through the Replication Manager. Figure 58 illustrates the main interactions of an upgrade. The actual upgrade is initiated by the *Upgrade Management Client*, UMC for short, by performing an

`upgradeGroup()` invocation on the UM (step 1), which in turn leads to a `upgradeRequest()` (step 2) multicast invocation on the respective upgrade layers of the group to be upgraded. Next, the upgrade layers of the replicas decide if its their turn to be upgraded; in this case, R_1 is selected for upgrade and the UL performs a `createReplica()` (step 3 and 4) invocation on the local execution daemon. This in turn causes the newly created replica (new software version) to join the group, and thus all replicas (both new and old) install a new view (step 5). Once the UL representing the upgraded replica detect the new version (R_1^u), it will make the old replica leave the group; once it has left, the `upgraded()` method is invoked on R_1.

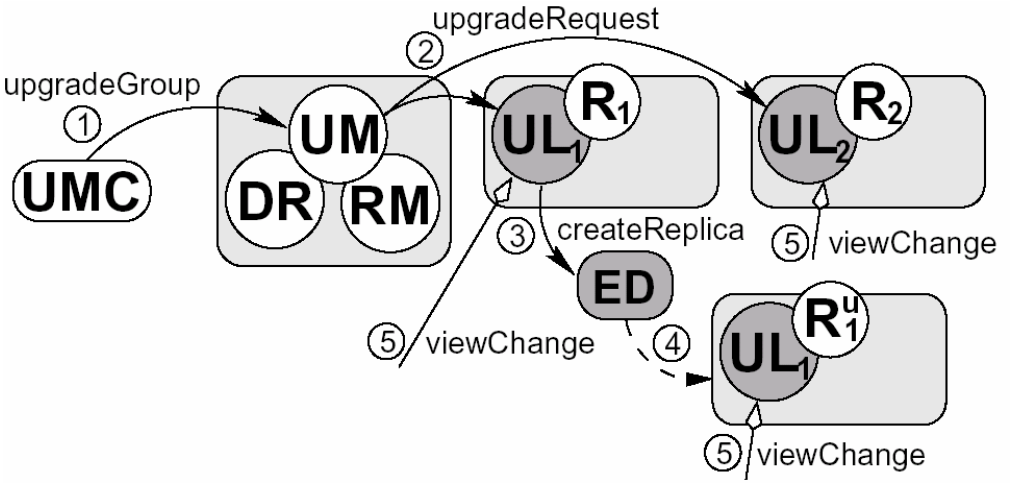


Figure 58. Interactions involved in an upgrade.

One of the main tasks of the upgrade layer is to determine which of the application replicas needs to be upgraded next, following the generic algorithm in section 7.1.1. As shown in Figure 58, the upgrade layer will wait for `viewChange()` events from the PGMS (see also step 5 in Figure 58). The replica to be upgraded next is determined on the basis of the replica positions in the view, e.g. the first member of the group will be upgraded first and so on.

The actual upgrade occurs by invoking the `createReplica()` method on the local execution daemon, and once the new version of the replica has joined the group, the old replica can leave the group. All of this is seamlessly handled by the upgrade layer.

The view originated from the PGMS provides a list of the current group member identifiers. In order to implement the upgrade layer, the member identifier has been extended with a software version number. This is used by the upgrade layer to distinguish between replicas running the old software from replicas running the new software, within the same view.

To prevent client requests from being processed by the replica during an upgrade, the upgrade layer interacts with the EGMI layer, as indicated by the `stopRequests()` method. This is required to prevent returning results to clients while being upgraded.

Assuming that the application is stateful, the new version of the replica must ensure that its state is synchronized with the remaining members of the group, before it can start processing client requests. State synchronization is provided by a separate layer in Jgroup, the *State Merging Service*, thus the upgrade layer does not need to deal with these issues.

7.4.6 Upgrade Manager

As shown in Figure 56, the upgrade manager object is co-located with the replication manager.

This is to simplify the interaction among the two components, yet making it easy to configure the system without the upgrade mechanism. The upgrade manager coordinates the upgrades. It provides an interface for a management client, to initiate upgrade request for a specific application that was previously installed through the replication manager. The `AppInfo` object supplied to the `upgradeGroup()` method encapsulates version information and class location for the new version of the application replica.

7.4.7 Conclusions

This section described one version of dynamic upgrade algorithm that is applicable to actively replicated servers implemented as a proof of concept. The algorithm implementation is based on Jgroup/ARM, which belongs to the class of object-oriented middleware and supports group communication.

The upgrade algorithm is an extension of Jgroup/ARM platform. The extension is enabled through the layer architecture and framework-oriented design of the middleware. ARM provides a possibility of inserting object-oriented entities, called layers into the protocol stack of the middleware platform. The implementation of the dynamic upgrade algorithm is contained in the Dynamic Upgrade Layer.

The implementation of the algorithm is object-oriented using the Jgroup layer concept. A Layer could be considered as a software component as it is interpreted in this thesis 2.3 if it were packaged and deployed in the Jgroup/ARM middleware platform if needed. The application specification could be considered as a simple component deployment language, which allows for describing the system configuration in terms of its components, its initial connections.

7.5 Summary

In this chapter, some upgrade algorithms were presented as the core mechanism to be provided by a dynamic upgrade system. Each of the algorithms addresses upgrades of different upgrade target and differs in the underlying system model. Some additional assumptions and requirements were needed to be added to general requirements on upgrade algorithms presented in section 5.1.2. One of the algorithm variations, i.e. supporting dynamic upgrades of actively replicated components was implemented using the Jgroup/ARM middleware as a proof of concept. The algorithm could be easily added to the middleware used due to its extendable layered architecture Jgroup/ARM .was designed.

8 Solution Evaluation

This section describes the evaluation of a part of the Dynamic Upgrade Framework presented in the previous sections of this thesis. The evaluation is based on a number of practical experiments in which a running Java RMI-based server is dynamically upgraded while processing a stream of client requests. The upgrade target in these experiments is a test server which is actively replicated with the Jgroup2 and upgraded using the algorithm presented in section 7.1.1. During the upgrade, the server is operational and processing a constant flow of client requests. The evaluation shows the upgrade algorithm working in practice and investigates the cost of applying the algorithm in terms of system performance overhead, like decrease in the server response time .

8.1 Aim

The aim of this experiment is to measure the performance of the dynamic upgrade algorithm described in section 7.1.1 applied to an actively replicated server. In the experiment scenario, while the target server is busy processing client's requests sent at regular intervals, it is being upgraded at the same time. With regard to fault workload, the server upgrade is carried out in ideal conditions, that is no server crashes during the upgrade (a fault-free environment).

To express our expectations, the following hypothesis are put forward:

Hypothesis H1.

The response time of a server is compromised by the upgrade algorithm. The performance overhead of the upgrade process does not depend on the server characteristics.

This hypothesis is based on the fact that the upgrade process involves activities consuming the resources of the nodes where the server replicas are running. Thus, performing additional actions decrease the responsiveness of the server as the upgrade process and the execution of the server code triggered by the client requests must compete now for the same resources. However, the performance penalty for the upgrade does not depend on the server characteristics in terms of its response time, replication level or the workload it has.

Hypothesis H2.

The server workload caused by the clients has an impact on the upgrade process time. The bigger it is, the longer an upgrade takes.

The hypothesis above is based on the fact that the upgrade process shares the resources of the hosts with the server replicas runs and that both the upgrade activities and the request processing by the server are performed with the same priority. In this case, the heavier the server workload is, the longer it takes to access the resources needed to complete the upgrade.

Hypothesis H3.

The time of the upgrade process is proportional to the replication level of the server to be upgraded.

As the algorithm implementation allows replacing one replica at a time and each replacement is performed in the same way (and presumably takes similar amount of time), the upgrade time of the whole server consisting of n replicas should be exactly proportional to n.

8.2 Experiment Description

This section describes the details of the experiments performed. The experiment configuration and scenario is sketched in section 8.2.1.Section 8.2.2 presents the experiment testbed, section 8.2.3 describes the benchmarking metrics and the methodology followed when conducting the experiments.

8.2.1 Experiment Configuration and Scenario

In the experiments the following logical subsystems are involved:

- **Test Client** that generate a stream of requests sent to the test server. The test client is implemented as a multi-threaded Java process that can send server requests with a given frequency.
- **Test Server** that is the upgrade target in the experiments. It is an actively replicated server with replication level determined by the experiment parameters. The server runs on top of the JGroup toolkit which is extended by `UpgradeLayer` as described in section 7.4.
- **Upgrade Manager** that is a piece of software that triggers the upgrade process. In the experiment configuration, it is a simple client that sends an upgrade request to the `UpgradeLayer` of each of the server replicas.

The logical configuration of the conducted experiments is depicted in Figure 59.

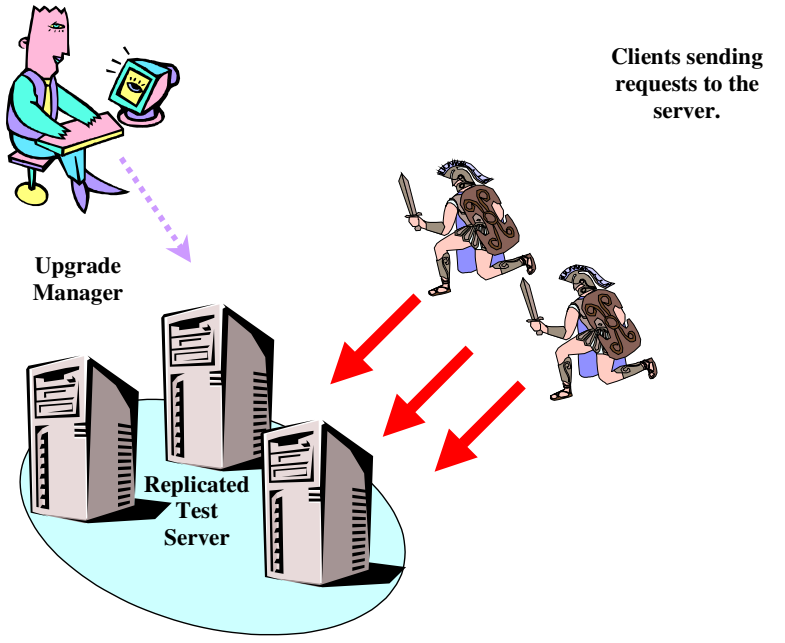


Figure 59. The experiment configuration.

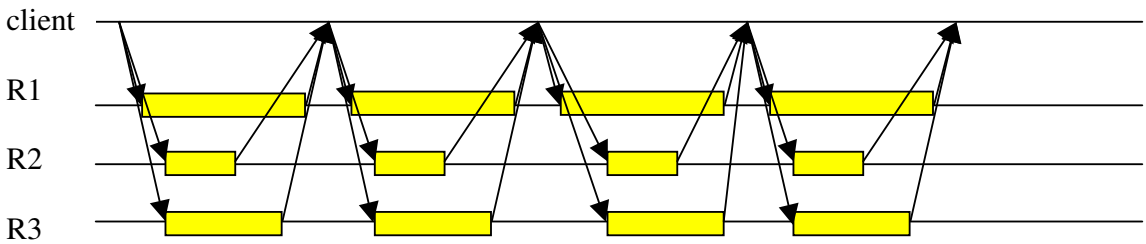
Each experiment is performed according to the following scenario steps:

1. The actively replicated server is started with a initial replication level *R* on a subset of the testbed nodes *N*. The server is stateless and offers one function without side-effects. The return value of that function depends only on the value of its input parameter. The function always terminates deterministically and takes a constant time *TF* to compute the function value.
2. A number of clients are started and send their request to the server, which is also the upgrade target.

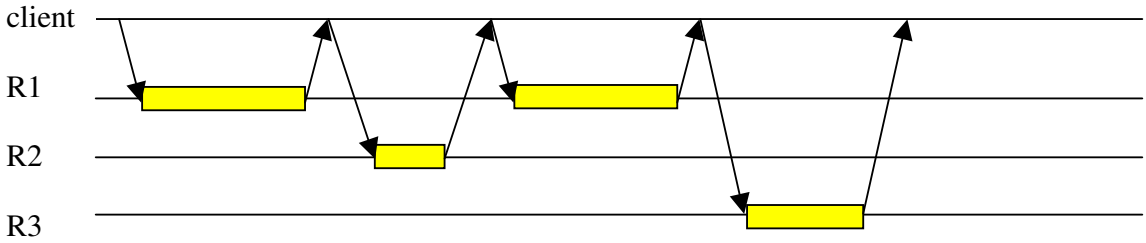
3. After the situation is stabilized after time *T*, the server upgrade may begin. An upgrade request is sent by the Upgrade Manager to all the replicas and the upgrade process commences. While the upgrade is being performed according to the upgrade algorithm, the clients are still sending the user requests. Some server replicas process the requests using the old code, while other are running already the new code.
4. After some later time *T+UT*, the last replica in the server group is upgraded the thus the upgrade algorithm terminates. From now on, the server is processing the user requests using only the new version of the code.

The actively replicated server can process two kinds of method invocations[63]:

- *Anycast invocations.* The semantics of the anycast invocation guarantees that at least one server replica performs such a method. (see the lower diagram in Figure 60 showing a time run of a sequence of consecutive anycast invocations).
- *Mulitcast invocations.* The semantics of the multicast invocation guarantees that such a method is performed by all server replicas. The responses from the replica are then collected and sent back to the client. (see the upper diagram in Figure 60 showing a time run of a sequence of consecutive multicast invocations).



processing a sequence of mulitcast method invocations



processing a sequence of anycast method invocations

Figure 60. The difference in processing multicast and anycast requests by an actively replicated server.

During the upgrade process not all replicas are active, that is processing client requests. In the current implementation of the algorithm, one is being replaced at a time and, therefore, it is not able to process client requests. The other replicas still may process client requests and make the whole server available. Figure 61 shows a sequence of client requests being processed during an upgrade process performed according to the algorithm proposed in previous chapter. For the sake of simplicity, new replicas has the same time lines as the corresponding replicas that they replace. To differentiate a new and an old replicas, the rectangular boxes on the old replica's time line are filled with different filling patterns: a checked pattern for the requests processed by the old replica and a pattern with diagonally-

stroked lines for the new replica. Additionally, the solid rectangular boxes represent the time periods that the replacement process is performed by the corresponding replica.

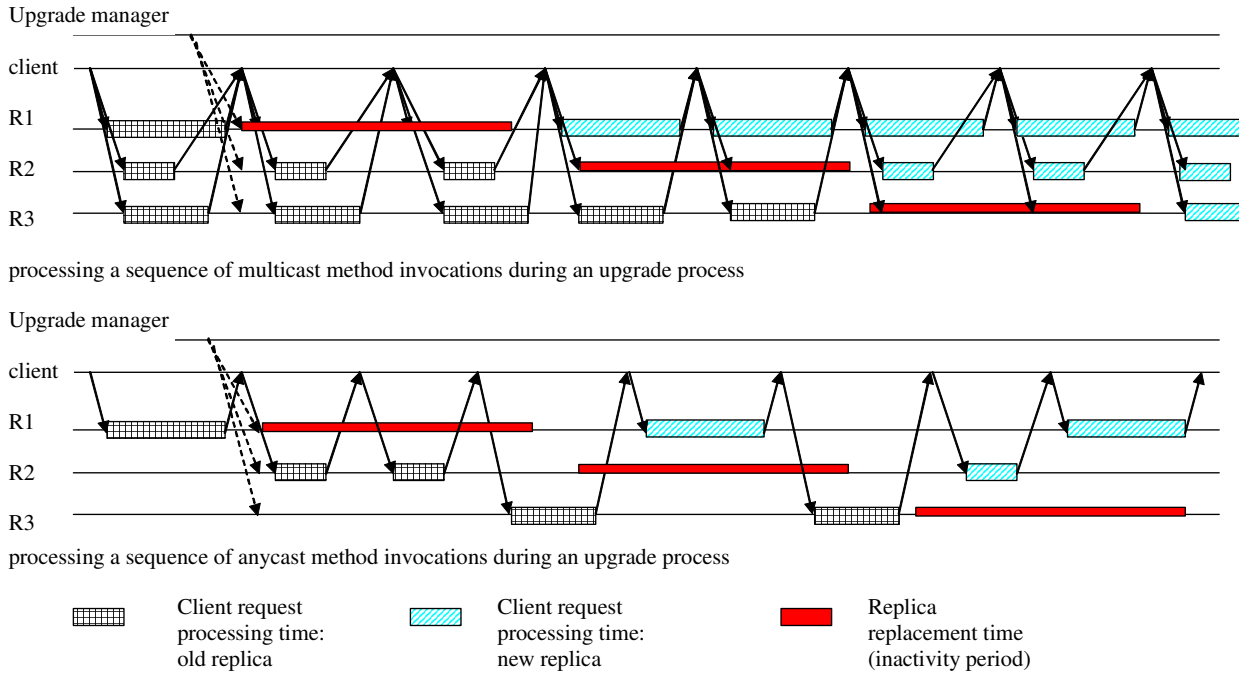


Figure 61. Processing of client requests during the upgrade process.

The following parameter are changed in the experiment series:

- *Server Replication Level.* The value of this parameter describes the number of replicas of the test server. It can be always greater than two.
- *Workload of the test server.* As mentioned in the experiment scenario, the server workload is simulated by sending a continuous stream of client requests to the server at regular time intervals. Such a request send events distribution allows observing the behavior of the systems under constant load and analyze the correlation of the server workload and the system availability characteristics during the upgrade process. In future experiments, more advanced models for request sending may be investigated. The sample values of the request sending frequency are set so that: (1) the maximal frequency f_{max} of request processing is found out (server saturation); (2) the minimal frequency, and the first sample at the same time, is set to some value f_{min}; (3) the other values are distributed in the range $[f_{min}, f_{max})$.

8.2.2 Testbed and experiment constraints

The logical experiment configuration runs on top of the middleware, system software and hardware configuration. The hardware in the experiment testbed consists of:

- 3 PCs, Pentium II 333Mhz,
- 1 PCs Pentium IV 2Ghz,
- 100Mbps Ethernet-based LAN.

Whereas the slower machines are used for test server replicas and the Jgroup/ARM

infrastructure processes, including `ExecDaemons` and `ReplicaManager` (see section 7.4.2), the latter machine is used to run the test client, a test server replica if replication level is equal to four and the scripts automating running a series of experiments (see below).

The following software packages are used to run the experiments:

- Debian 3.0[11], standard Linux 2.4.X kernel[54],
- Sun JDK 1.3.1 for Linux[119],
- ARM/Jgroup 2 extended with the DUF implementation as described in section 7.4,
- Test server replicated with ARM/Jgroup2. The test server is an echo server in that it replies simple responses based on the input parameters with certain delay. The server provides three test methods:
    - **m1** is a *anycast* void method. The implementation of method m1 just updates internal variables, like invocation counters and returns control. The method is also void in the sense that it does not return any return value. The response time of this method is used as a reference for the minimal response time of a anycast method possible in the experiment testbed.
    - **m2** is a *multlicast* void method. The method is also void in the sense that it does not return any return value. The response time of this method is used as a reference for the minial response time of a mulitcast method possible in the experiment testbed.
    - **m3** is a sample *anycast* method with an input parameter and returning some value. The input parameter is a byte array with a constant size (10k bytes). The method performs computations (with a constant time bound) to simulate workload by a typical server implementation.
    - **m4** is a sample *multicast* method with some input parameters and a return value. It performs some computations (with a constant time bound) to simulate workload by a typical server implementation.
- Set of shell scripts that control and automate the experiment runs. The scripts are responsible for starting the experiment in the initial configuration, collecting the logs with the experiment raw data from the testbed nodes. processing and storing them in the format needed for further analysis and presentation (e.g. gnuplot diagrams or excel spreadsheets).

The experiment testbed has the following constraints:

- **Four nodes.** The testbed has only four Linux machines. Running a replicated server with a replication level of greater than four would mean that multiple replicas would be running on the same node. The replicas would be competing for node's resources and this could contribute to noise with regard to to the performance measurements. Consequently, it was decided to perform experiments with server replicated with a replication level of four or less.
- **No total isolation.** However the testbed network is not completely isolated from the environment external to the experiment, i.e. traffic from other machines in the network, the network traffic is much reduced by applying switch filtering the incoming traffic. In this way, the noise coming from non-experiment related sources is rather low. Another issue is the noise coming from other processes running on the testbed machines. To handle that, the number of non-experiment processes on the Linux machines was reduced to minimum. Additionally, the testbed nodes share experiment files through NFS, which may have impact on certain system performance characteristics. Therefore, the experiment results should not be considered as absolute

values. Instead, the experiments are set up to show some relative system characteristics.

Considering the experiment testbed constraints above and the method for generating sample values for the experiment parameters, the following experiment parameter values depicted in Table 10 are assigned for the series of experiments carried out.:

Parameter	**Sample values**
R – level of replication of the target server	2,3,4
X – workload of the test server (req/sec)	1, 4, 8 for experiments with test method m1 and m3 20, 40, 80 experiments with test method m2 and m4

Table 10. Experiment independent parameters.

8.2.3 Benchmark metrics and Methodology

The following are the basic metrics that are helpful to measure the performance of the algorithm.

- **RT**$_x$(t) – **R**esponse **T**ime of test method request x=m1, m4 sent at time slot [t, t+dt), where dt is called *cycle time*. The response time is measured throughout the experiment time: before the upgrade process commences, while the upgrading takes place and after the upgrade is complete.
- **UT**(R, X) – **U**pgrade **T**ime is the time period from the moment an upgrade request is sent to the upgrade target to the moment target's upgrade process completes.

The algorithm evaluation is based on the measuring the benchmarking metrics identified in section 8.2.3 for a number of experiments performed according to the scenario described in section 8.2.1. Each experiment is carried out with different combination of the independent parameters listed in section 8.2.1. Each such experiment is repeated 10 times to reduce the significance of the influence of external factors, including other traffic in the network and other processes accessing the computing resources of the testbed nodes.

The time measurements are done using the standard Java library calls (`System.currentTimeMillis()`) . To reduce the time measurement imprecision, the server response times that are in range of a few milliseconds for method m1, a few ms teens for method m3 and a ms few tens for method m2 and m4, the average response times are computed for a number of invocations on the server within a time period called cycle time. In the experiments done, the cycle time was set to one second.

The upgrade times are measured using the same Java library call. The time is determined by the time point at which the upgrade response arrives a replica of the test server and the time point at which a replica leaves the group of the test server. The upgrade process time is considered the longest[5] upgrade time of a single replica. The measurement method is visualized in Figure 62. The vertical lines symbol the time lines of server replicas. The arrows point the times on which certain events occur to replicas. The yellow rectangular boxes indicate replicas processing the active steps of the upgrade algorithm, whereas dashed lines

[5] Even though that the actual upgrade process time may be longer than the longest upgrade time of a single replica because of different times of an upgrade request arrivals at different replicas, we abstract from this in the experiment measurements. The time difference is an insignicant fraction of the whole upgrade process time.

show passive steps of the algorithm, that is replica waiting periods.

A derived benchmark metrics, Average Upgrade Time, AUT for short, is then an average upgrade time taken for a number of repeated experiments with a given set of parameters. This metrics is used for further investigation for better precision's sake.

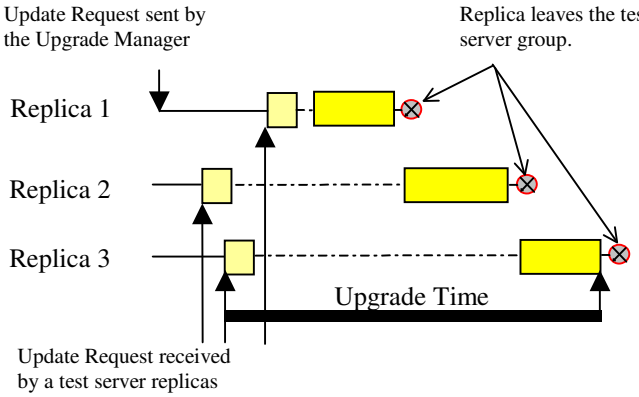


Figure 62. The measurement method for the upgrade process time.

8.3 Results

The results of the experiments are graphically presented in this section as diagrams. First the server response time is presented as it changes in time during the experiment. After the upgrade process times are presented for experiments in different configurations.

8.3.1 Server Responsiveness Analysis

The section presents the results of the server responsiveness analysis. The analysis has been performed separately for the anycast methods in section 8.3.1.1 and for the multicast methods presented in section 8.3.1.2.

8.3.1.1 Anycast methods

In Figure 63[6], the average server response times are shown for an experiment with replication level 2, testing method 1 and request rate 20, 40 and 80 requests per sec. The average response times are computed for a number of 10 repetitions of the same experiment.

The diagram shows that the response times increase during the upgrade process starting at 3s and terminating on average after 15s, that is 18s of the whole experiment time. The response times are especially large at the beginning of the upgrade process (peak around 3s exceeds 12ms) and at the end of the upgrade process (peak around 15s exceeds 8ms). This can be explained by:

- additional activity needed to process the upgrade request including starting a replica with a new code on the same host which the replica to replace runs on, and
- additional communications needed to install a new membership view in the server

[6] Each experiment is named acorrding to the following naming scheme: Ra mb rc, where Ra means Replication level and a can be equal to 2, 3, 4; mb means test method b and b can be equal to 1,2,3 or 4; rc means workload in request sent to the test server per seconds, where c can be 20, 40, 80 for anycast test method and 4, 8, 16 for multicast methods. Additionally, each symbolical variable in this scheme a, b and c can be set to X meaning any value from the allowed range.

group.

Moreover, the diagram shows that the overhead peaks occur at the same points in time in each experiment independent of the traffic generated by the test clients. This phenomenon can be explained by that fact that the time that 1st replacement is started without waiting until the previous client request is fully processed. The implementation of the `UpgradeLayer` of the selected replica for the 1st replacement immediately starts a new replica on the same host and switches the replica state to `upgrading` (see section 7.2.4). The processor on which the selected replica is running is busy with starting a JVM and loading the needed classes. This decreases the response time of the whole server. As soon as the new replica is started and joins the server group, another replica can be chosen to be replaced. The time the next replacement takes place depends mainly on the speed a replica can start and join the group.

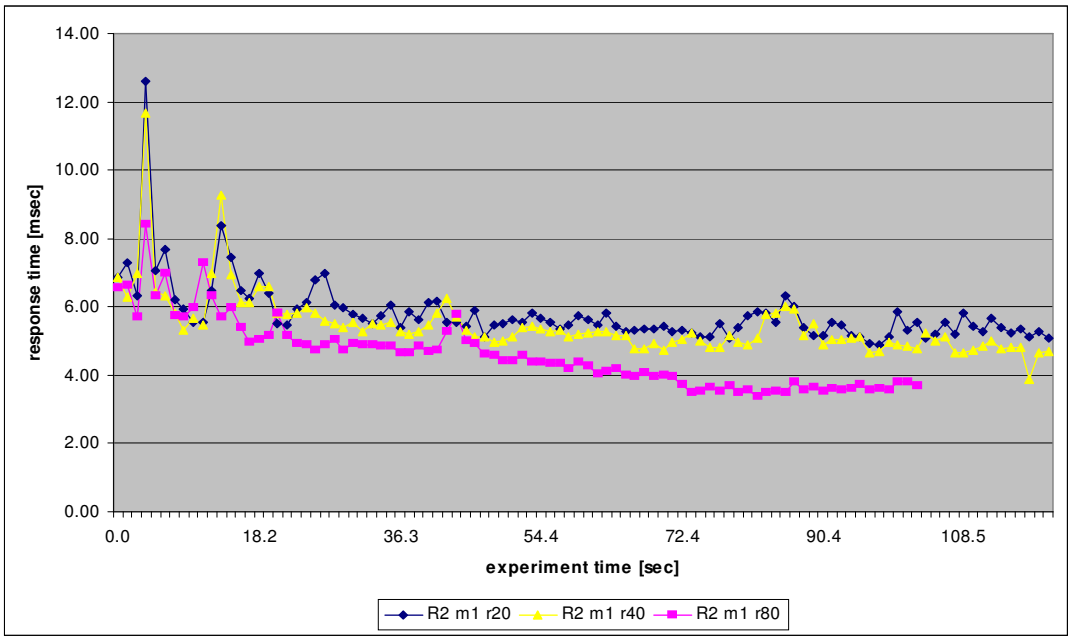


Figure 63. Average Response Times in experiment R2 m1 rX

Figure 64 shows a time dependency of response times of the server's test method 3 to compare. The diagram looks very similar to the previous one in terms of response time variations throughout the experiment. On average, test method m3 has a bigger response time than method 1. At the same time, the overhead introduced by the upgrade process is similar to that of method 1 both in terms of the time dependency (response time peak number and their time points).

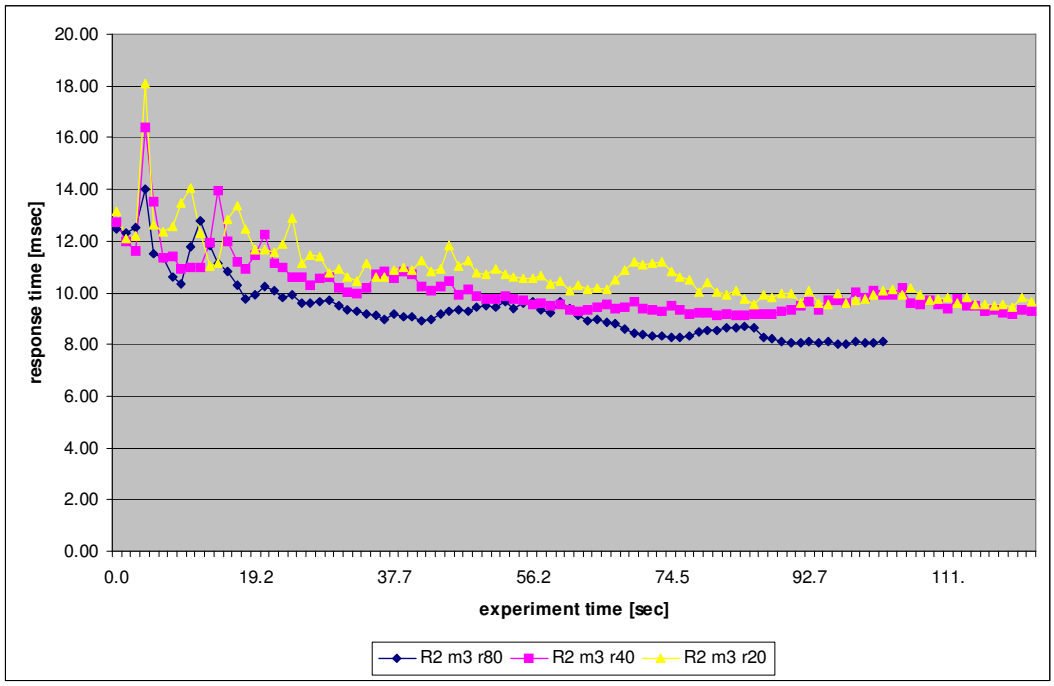


Figure 64. Average Response Times in experiment R2 m3 r20

Figure 65 presents the average response times for test method m1 changing during the upgrade process for a server replicated with a replication level equal to 2, 3 and 4. In this diagram, the rest of the experiment parameters are the same. A significant increase in the response times occurs coherently at the beginning of the upgrade process. The biggest response time at this time point overhead seems to be for the server with replication level of 2. The response time curves also have other peaks at 6s, 11s and 14s but they seem not be coherent. It is related to the fact that the different number of replicas have to be replaced during the upgrade process.

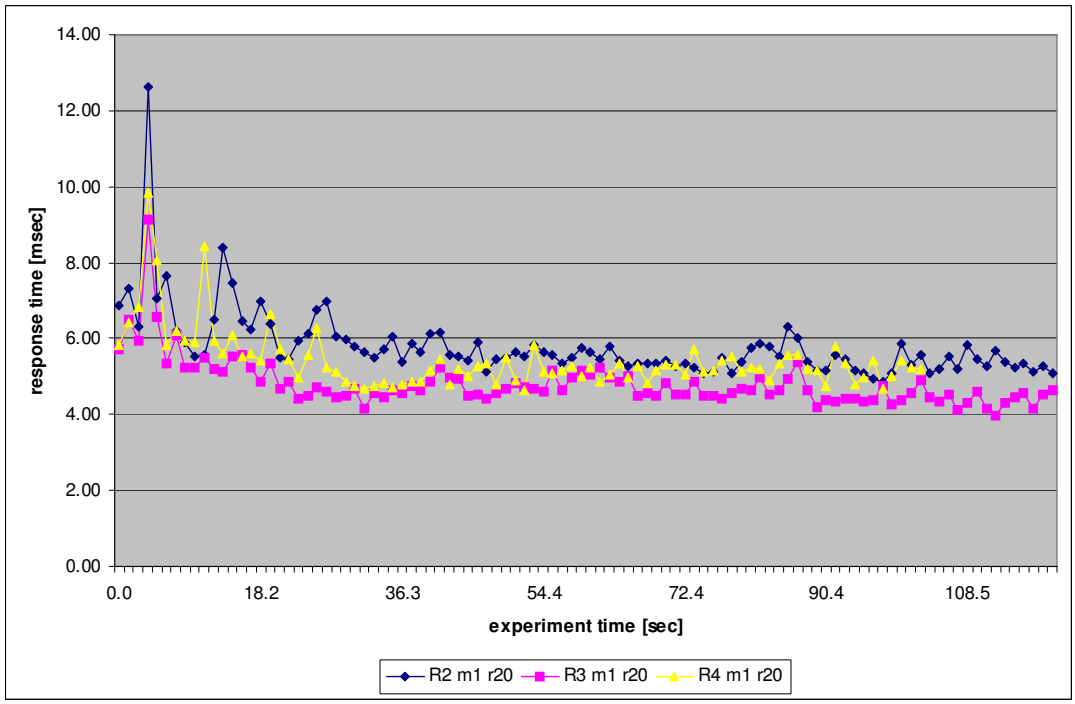


Figure 65. Average Response Times in experiment Rx m1 r20:

Table 11 shows average response times during the upgrade process and after it for the experiments with test method 1 used. The experiments vary in the replication level and the traffic load on the server. The table contains a ratio of the during-upgrade and after-upgrade average times. On average, the response times during the upgrade process are greater by 25% than the response time taken before or after the upgrade process. This can be considered as the overhead of the dynamic upgrade algorithm that is manifested by its negative impact on the test servers responsiveness.

Experiment	During upgrade		Before/After upgrade		Ratio
	Average Response Time	Standard Error	Average Response Time	Standard Error	
R2 m1 r20	6.96	1.77	5.54	0.38	**1.26**
R2 m1 r40	6.63	1.58	5.16	0.39	**1.29**
R2 m1 r80	5.99	0.92	4.19	0.58	**1.43**
R3 m1 r20	5.72	1.07	4.61	0.27	**1.24**
R3 m1 r40	5.41	0.84	4.32	0.36	**1.25**
R3 m1 r80	5.03	0.51	4.04	0.22	**1.25**
R4 m1 r20	6.04	1.21	5.14	0.29	**1.18**
R4 m1 r40	5.74	0.93	5.01	0.22	**1.15**
R4 m1 r80	5.09	0.67	4.08	0.27	**1.25**

Table 11. Average response times during and after the upgrade process for test method 1.

Additionally, Table 11 includes the standard errors of the response times. The greater values of the standard variation for the response times during the upgrade process confirm the observation of strongly varying server responsiveness.

To conclude, the upgrade process adds some overhead onto the server responsiveness. This overhead is of quickly changing intensity and oscillates between 0 to its peak value of over 200%. However, its effective overhead is on average around 25% when measured throughout the overall time of the upgrade process. Additionally, it seems that there is a relation between the number of peaks and the replication level. It can be explained by the fact that the peaks are to be contributed to the resource consuming activity of starting an operating process that each new replica joining in causes.

8.3.1.2 Multicast methods

Figure 66 depicts an equivalent diagram with server response times during a set of repeated experiments where test method m2 is used. Method m2 is a multicast method The experiments shown in the diagram differ in the workload generated. The response times in these experiments are quite correlated and follow the same pattern: the response time of the test server grows reaching its maximum peak of around 100 msec, which exceeds the average response time before and after the upgrade process by a factor of 500%. In this experiment, the number of peaks during the upgrade process is equal to three.

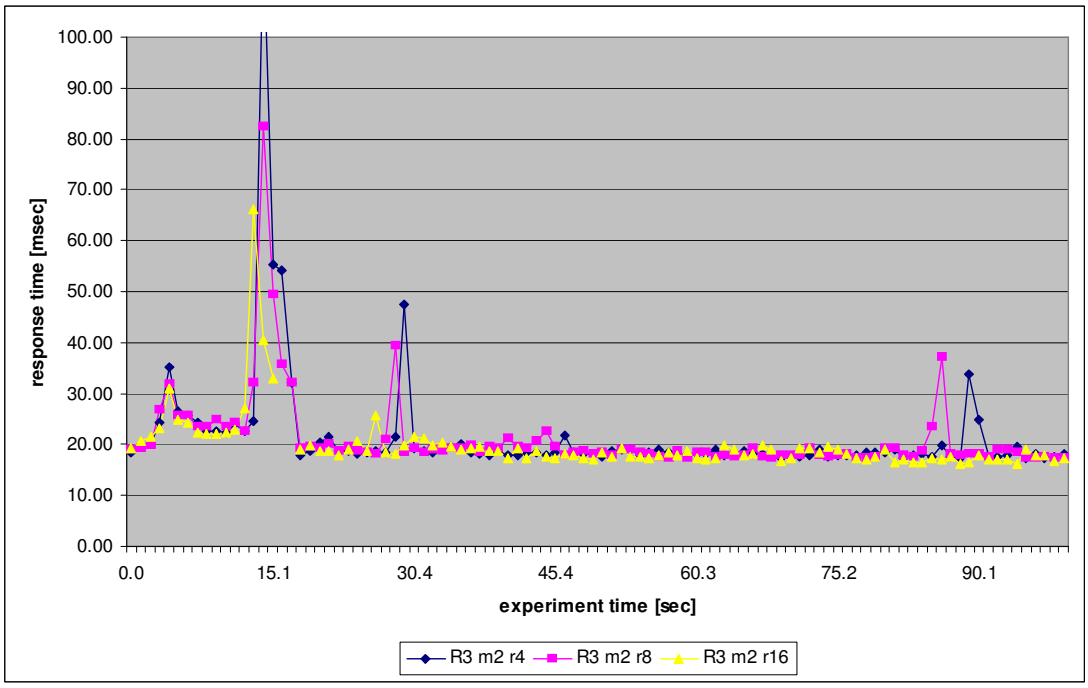


Figure 66. Average Response Times in experiment R3 m2 rX.

Table 12 shows average response times of method 2 both during the upgrade process and before/after it. The last column includes a ratio between these times for each experiment. The average ratio for all experiments amounts to 1.66. Compared to the analogous results for method 1, the ratio is much higher (66% against 25% overhead) and indicates a higher performance overhead of the upgrade process for this test method. Furthermore, considering that fact (see sections below) that the upgrade time for method 2 is also much longer than for method 1 and 3, it can be stated that the overall performance overhead of the upgrade algorithm with regard to servers processing multicast methods is significantly higher than the one regarding servers processing unicast methods.

Experiment	During upgrade		Before/After upgrade		Ratio
	Average Response Time	Standard Error	Average Response Time	Standard Error	
R2 m2 r4	37.46	22.66	20.08	4.99	**1.87**
R2 m2 r8	31.20	13.40	18.96	5.52	**1.65**
R2 m2 r16	27.36	12.54	17.79	2.75	**1.54**
R3 m2 r4	35.80	25.79	19.06	3.75	**1.88**
R3 m2 r8	32.31	15.54	19.09	3.29	**1.69**
R3 m2 r16	29.36	12.32	18.26	1.44	**1.61**
R4 m2 r4	78.03	21.65	47.85	5.78	**1.63**
R4 m2 r8	59.76	11.73	38.42	3.34	**1.56**
R4 m2 r16	60.52	11.00	38.92	1.93	**1.55**

Table 12. Average response times during and after the upgrade process for test method 2.

8.3.2 Upgrade Time Analysis

Another system parameter measured in the experiments is the upgrade time. Its value should be as small as possible in order to limit the performance overhead added by the upgrade process as well as to reduce the risk of failures during the upgrade. The following sections present the average values of the upgrade time, AUT, measured in the experiments.

Figure 67 shows the average upgrade times [in ms] for the experiments with clients sending test method 1 to the server. The times have been presented for upgrade processes with different server replication levels (bars painted with different shades) and for different rates of client requests sent to the server (depicted on the x axe).

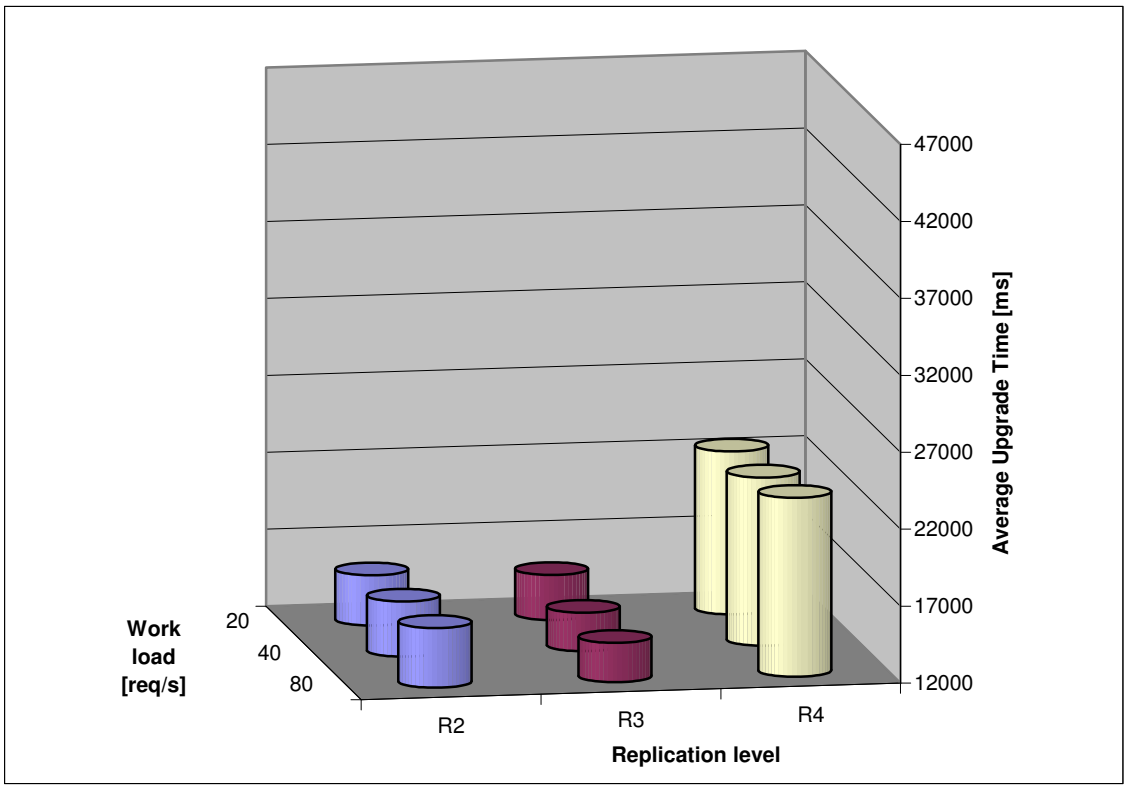


Figure 67. Average upgrade times for experiments with method 1.

Figure 68 presents average upgrade times for experiments with test method 2. The vertical axe represents the upgrade times in milliseconds, whereas the horizontal axe represents the server load in the number of requests processes per second. Different shades of the diagram columns represent different replication level of the test server, with the left-hand blue column meaning replication level of two, the central column – replication level of three and the right-hand column – replication level of four.

The diagram presents suggests a correlation between the upgrade time and the load of the server during the upgrade. Namely, the heavier the load is, the longer it takes to complete the upgrade process. Another dependency that can be seen in the diagram is a positive correlation between the upgrade time and the replication level: the higher the replication level is, the longer the upgrade process takes. This observation seems to support hypothesis H1 put forward in section 8.1

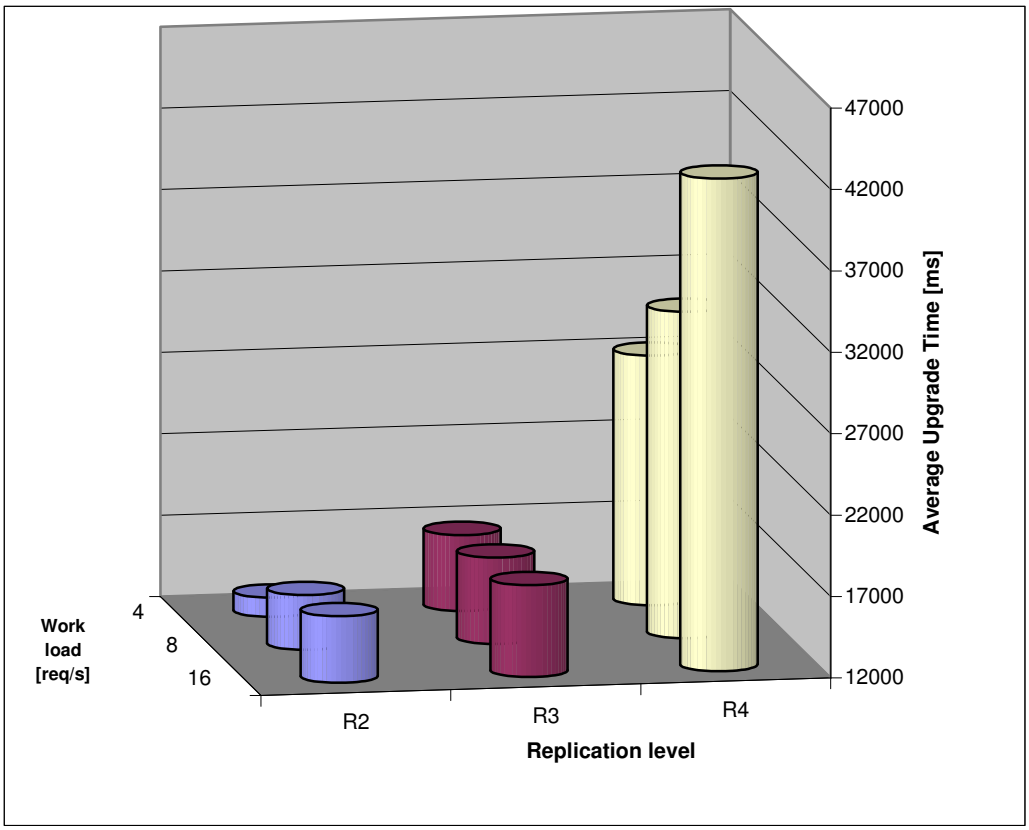


Figure 68. Average upgrade times for experiments with method 2.

8.4 Conclusion

To estimate the performance overhead of the upgrade algorithm introduced in previous chapter of this thesis, a number of practical experiments with the algorithm prototype were performed as described in this chapter.

The results of these experiments confirm the hypothesis formulated at the beginning of the chapter as follows:

With regard to Hypothesis H1, the upgrade process has a negative impact on the server responsiveness during the upgrade process. Depending on the method invocation semantics, this overhead differs from 25% for the anycast invocations to 66% for the multicast invocations.

Therefore, it may be worth of changing the multicast type of traffic to anycast one for the time the dynamic upgrade is to be performed if it is feasible. Using the existing JGroup middleware platform it is possible if both invocation types are foreseen for the server method at compiling time of the server code because the method signatures differ for these method invocation styles. Using the Java reflection support and java byte code analysis and manipulation mechanism provided by the Byte Code Engineering Library [4] it could be possible to develop tools for generating the corresponding method syntax definitions. The difficulty would lie in the automated detection of side-effects and their resolution when transforming an anycast to multicast method or vice versa.

Secondly, Hypothesis H2, which states that the time taken by the upgrade process applied to a replicated server depends on the replication level, can be confirmed. On average, it takes approximately double as long to upgrade an actively replicated server with replication level of 4 compared to a server with only two replicas.

The upgrade time for the sample replicated server used in the experiment was in range between 13 and 42 seconds. During this upgrade time, the server reliability is decreased and the system more susceptible to crashes as the replication level is smaller than out of that time. To reduce the crash risk, this upgrade time may be shortened by defining an upgrade enabling condition so that more replicas are upgraded in parallel. On the other hand, when the more replicas are being upgraded at a time, the less replicas are available and the server reliability is reduced. This trade-off may be solved by starting additional replicas with new code before replacing the old ones. However, the server responsiveness is then compromised as the increased replication level causes additional penalty as even more replicas have to compete for the same hardware resources if provided additionally. The rule of the thumb is to perform upgrades with at least two replicas not being upgraded. Additionally, the upgrade should be then performed during the time that the crash probability is smallest, for instance when the server workload is low.

In the experiments carried out, it was not possible to fully verify , that is the server workload caused by the clients has an impact on the upgrade process time. The server is loaded from small to mean load and the host resources are ample to handle the additional activity of upgrading the server. Only in case of a test server with four replicas, the upgrade time depends on the workload. Some more experiments with heavy load on the server are needed to confirm the presumed correlation between the upgrade time and workload.

9 Conclusions

This thesis has investigated the problem of upgrading distributed software components on the fly. Software upgrades have been considered from several perspectives: as a special case of deployment, as an approach to runtime management and as a mechanism to increase availability of time-critical systems.

In this work, the topic of dynamic upgrades of distributed software was tackled by identifying the range of aspects of the core problem, such as defining the upgrade process, coordination and management of dynamic upgrades, upgrade validation. The problem analysis was initially based on the investigation of the existing approaches described in the research literature. This resulted in a comprehensive comparative study of the state-of-the-art systems supporting dynamic upgrades. The study allowed to define in a set of general requirement on the platform and the software systems to upgrade.

The set of requirements was then taken as the input to develop a more generic model for middleware platforms aimed at supporting upgrades on the fly. The model was called Deployment and Upgrade Facility and formulated using the UML notation. It is modeled following the Unified Process with help of different models. In use case model, the main functionalities of a middleware platform to support dynamic upgrades are specified using the use case notation. In design model, the architecture that is derived out of the model is presented in term of the objects and their relationships and dynamic interactions. The applicability of the model was verified by deriving a concrete design of an object oriented framework supporting deployments and upgrades, The framework was implemented in a series of prototypes extending the mainstream middleware platforms, including OMG's CORBA and JAVA RMI (Jgroup).

9.1 Contributions

The major contributions of this thesis are:

- *A model of a system supporting deployment* of distributed applications and in particular, dynamic upgrades of distributed components. The model results from a comprehensive analysis of issues to be addressed when building software capable of being upgraded on the fly. The model is described in chapter 5 and contains the following parts:
  - *Use case model* describing the actors relevant to deploying and upgrading a distributed system, as well as, the core capabilities a system has to provide to enable deploying, and in particular upgrading distributed components.
  - *Component Model*, which defines the component and its features with regard to the implementation, deployment and runtime phases.
- The ***D****eployment and* ***U****pgrade* ***F****acility* which is an object-oriented framework capturing design solutions and patterns to construct dynamic upgrade support systems. The framework is based on the model above and extends the model with a design of interactions . It has been described in chapters 6-8 of this thesis. The major features of DUF are:
  - *Support for dynamic upgrades of distributed applications*. The framework enables software upgrades by replacing software component on the fly. The software components may be distributed in multiple containers running on

various hosts.

- *Solution extensibility*. The DUF is designed as an object oriented framework with extendibility in mind . The extensibility is expressed in the following aspects:
  - it is possible to extend the facility with new upgrade algorithms, some of which have been presented in this thesis. In particular, upgrade of non-replicated and replicated services is enabled.
  - it is possible to introduce new upgrade management schemes by inserting new management policies.
- *Advanced Upgrade Management*. The DUF allows for managing upgrades in complex distributed systems. The management subsystem enables setting up the upgrades in the identified dimensions.
- *Technology-independence*. The framework does not depend on the technology-specific realization of the supporting mechanism.

The DUF has been extended by implementation of the supporting mechanisms that were implemented in a series of prototypes using a concrete middleware technology, including CORBA and Jgroup, a Java RMI extension.

- *Dynamic upgrade problem taxonomy and a state-of-the-art solution survey.* The thesis identifies and classifies problems related to building dynamically upgradable distributed systems and the dynamic upgrade support systems. The proposed taxonomy is a result of comprehensive comparative analysis of existing systems and ones under development/specification , which support dynamic upgrades in distributed systems.

The results of the thesis have been published in international conferences and discussed with world experts in the area [109][110][112][113]. The solutions presented in the thesis were submitted as contributions to the international research projects, in which the author of this thesis actively participated, including Eurescom P910[17][18], Eurescom P924[111][20] and IST FAIN[22][23].

9.2 Novelty

The approach to designing a dynamic upgrade support system presented in this thesis is novel in many ways:

- *Consideration of dynamic upgrades from the deployment perspective.* The thesis provides a comprehensive view on dynamic upgrades in distributed applications, as a special case of the software deployment process. The view encompasses the deployment aspects and runtime issues related to software upgrades and extends the scope of most of the previous work on dynamic upgrades.
- *Separation of the technology independent part and technology specific part*. The approach of this thesis is to divide the solution for dynamic upgrades into three parts: the model, the framework for Deployment and Upgrade Facility and the supporting mechanisms. Whereas the model and framework are technology independent, the realization of supporting mechanisms is specific to the underlying technology.
- *Heterogeneity Support.* The DUF supports dynamic upgrades of distributed, heterogeneous services. The services may consists of multiple components implemented using different technologies spread across a number of containers and hosts. Thanks to the platform-independent model proposed in this thesis and the

openness of the COBRA middleware technology chosen for the prototypical implementation (open interfaces, interoperability and portability), it was easy to provide an implementation of the DUF which supports upgrades of heterogeneous services.

9.3 Goal fulfillment

The first goal of the thesis has been to construct a model that describes capabilities needed to support dynamic upgrades in distributed software systems. To meet this goal (P1 in section 1.2), the use case model has been specified in chapter 5. The model includes an underlying component model defining how to construct distributed software out of components (goal P2). The model defines the core capabilities of the system and actors interacting with the system and is completely independent from the implementation technology (goal P3) The model has been validated by building a set of prototypes described in section 5.4 and chapters 6-7. using existing middleware platforms (goal P4). Finally, to meet goal P5, the model is specified and presented using the Unified Modeling Language, a standardized notation, which is widely used by the industry and research community working on the object-oriented and distributed systems.

The solution developed in this thesis fulfills all the goals set in section 1.2. It provides a support for performing and managing dynamic upgrades of distributed software components and is compliant to the model presented in chapter 5 (goal P6). The degree to which the support is provided is discussed in the following subsection when presenting the fulfillment of the more detailed requirements on the solution specified in section 5.1.

Furthermore, to meet goal P7, the solution has been practically validated by implementing a series of prototypes using the CORBA and RMI main-stream middleware technologies. It is designed and implemented as a framework that allows for its extensions by both by adding new support mechanisms and management policies for dynamic upgrades (goal P8). The implementation of the solution is divided into the platform independent part and platform specific part (for Jgroup). Thus, its reusability is increased (goal P9). When porting it to another middleware technology, only the platform specific part needs to be implemented. Finally, the performance of the DUF system has been tested in a series of experiments as presented in chapter 8 to meet goal P10.

9.3.1 General Functional Requirements

With regard to the requirement on *Basic Deployment Capabilities* as express in R1. The DUF provides the basic deployment capabilities as described in chapter 5. In particular, it is possible to release, deploy, remove and withdraw a service from the system. The design and implementation of the system demonstrating these deployment capabilities was described in section. 5.4.

With regard to the requirement on *Support for distributed services,* as stated in R2, the DUF supports deployment and upgrades of distributed component-based services. The underlying component model defined in section 5.3 enables a service to comprise a number of software components distributed on multiple containers and hosts.

Concerning requirement R3 on *Support for co-existence of multiple versions*, it is possible to release and deploy multiple versions of a service component in a distributed system on each node, i.e. two or more version of a service component may be deployed on a node.

9.3.2 General Nonfunctional Requirements

Solution *extendibility* has a high priority in the approach presented in this thesis as requested

in R4. The solution is designed as a extensible framework which enables using a number of available upgrade management strategies and upgrade algorithms for various types of upgrade targets, as well as, inserting new dynamic upgrade management policies and supporting mechanisms.

Referring to R5, *DUSS Portability*, a significant part of the Deployment and Upgrade Facility has been developed in a platform independent way so that the underlying capabilities of the middleware platform are abstracted. Additional, the requirement has been addressed by selecting Java as the programming language. This part of this solution can be easily ported to any middleware technology and other hardware platforms, for which an implementation of the Java Virtual Machine is available. Only small part of the implementation code of the support mechanisms is platform dependent and needs to be adapted to different middleware platforms.

9.3.3 Requirements on Upgradable Components

As far as *Orthogonal Upgradability* is concerned, requested in R14, the upgrade capabilities of a component, which belong to the system management issues is separated from the component functionality (business logic) in that another interface is added to the sets of the component's main interfaces related to the business logic of the component. The implementation of the this interface may have to be integrated with the implementation of the component's business logic to insert the so called upgrade points.

Simplicity of development of upgradable components was requested in R15. The component developer has to put some effort to add an implementation of the `Upgradable` interface. For the stateful components, the implementation has to support the state transfer mechanism in that the `get_state()` and `set_state()` method implementation has to be provided. These methods can be partly automated as shown in the work on persistent system, eg. Orthogonal persistence approach. In general, a automatic support for state transfer is not feasible. The DUF handles this requirements by following a framework-like approach providing a number of interface definitions that has to be then implemented by the component developer or generated by the state transfer support tools left out of this thesis.

According to requirement R16, *the set of constraints on the system imposed by a DUSS should be minimized*. The approach taken in this thesis does not impose any constraints on the development process. Neither the system architecture nor programming style is constrained. The design flexibility is supported as much as the component-oriented paradigm allows for that. The DUF facility is a design framework that has some impact on the design of a component. (new interfaces that has to be supported by the upgradable components). Although the framework provides a number of support mechanism implementations, like upgrade algorithms specialized to handle upgrades of various upgrade targets, it is extendable and other supporting mechanisms can be added.

With respect to requirement R17, *dynamic upgrades of heterogeneous services should be supported***.** Deployment and Upgrade Facility is defined on the conceptual level and specified using UML. It has been implemented in the CORBA environment that supports heterogeneity of software components in a system. Thus, it supports interoperability.

9.4 Open issues and Future Work

The following topics have not been investigated at full length in this thesis and are proposed further research:

- **Higher dynamics of change.** As the upgrade algorithms presented in this thesis provide some constraints on the allowed differences between versions of the component to upgrade, some investigation of upgrade algorithms should be carried out

that allow for higher dynamics of change during upgrades.

- **Tool support for automating the preparation of upgradable components**. Some research should be done to investigate in how far it is possible to automate adding the upgradability to a component that has been designed without this goal.
- **Solution Scalability**. The solutions presented in this thesis have been applied to relatively small distributed systems. Further research is needed to check the scalability of the solutions to a large-scale highly-distributed system, running perhaps on hosts connected with a links of significant communication delays.
- **Applicability to other software architectures**. The solution has been used in the multi-tier architecture based on the RPC communication paradigm. The recent tendency in software system development is to design systems in a decentralized and loosely-coupled way that are more suitable for ubiquitous computing paradigm. It would be worth of investigating how the solution presented in the thesis can be applied to software systems built in these architectures.
- **Safe Online Upgrades.** Safety of software system upgrades is a critical issue. The solution presented in this thesis assumes that the service deployer, who is a trusted entity for the target system, may also trust the service code providers with regard to the suitability and correctness of the code. If this assumption is weakened, some additional mechanisms supporting security of the upgrades need to be made available.

Acronyms

This section contains a list of acronyms used throughout the text of this thesis.

ADT	Abstract Data Type
AML	Architecture Modification Language
API	Application Programming Interface
ARM	Autonomous Replication Management
AUT	Average Upgrade Time
COM	Component Object Model
COS	CORBA Object Services
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DU	Dynamic Upgrade
DUF	Deployment and Upgrade Facility
DUMF	Dynamic Upgrade Management Framework
DUSS	Dynamic Upgrade Support System
DUT	Deployment and Upgrade Tools
EE	Execution Environment
EJB	Enterprise Java Beans
GCS	Group Communication System
IDL	Interface Definition Language
ISO	International Standardization Organization
JVM	Java Virtual Machine
MOM	Message-Oriented Middleware
ODP-RM	Open Distributed Processing – Reference Model
OMG	Object Management Group
OSD	Open Software Description
POA	Portable Object Adaptor
RMI	Remote Method Invocation
RP	Reference Point
RPC	Remote Procedure Call
RT	Response Time
SDL	Specification and Description Language
SDR	Service Dependency Resolver
SP	Service Provider
UML	Unified Modeling Language
VE	Virtual Environment
XML	Extendable Markup Language

Glossary

This is a list of terms adopted from the literature or introduced in this thesis. Each entry in the glossary contains the term, its explanation and the literature source it stems from.

Term	Explanation	Source
Actively-replicated Server	A server replicated with a number of replica, each of which is running in parallel and ready to process incoming client requests.	
Anycast Method Invocation	The semantics of an anycast invocation guarantees that the invocation on a replicated server will be executed by invoking the same method on at least one of the server replicas [unless client is completely partitioned from the server in case of partionable environment].	[63]
Architecture	Overall design of a system. An architecture integrates separate but interfering issues of a system, such a provisions for independent evolution and openness combines with overall reliability and performance requirements.	[123]
Component	A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Context dependencies are specified by stating the required interfaces and the acceptable execution platform(s). Component has the following properties: it is a unit of abbstration, deployment and management. In the context of this thesis, the component is mainly considered as a unit of dynamic upgrade.	[123]
Compound Upgrade	A dynamic upgrade that involves multiple upgrade targets.	
Critical Systems	Software systems that are critical to the operation of the organization or its mission.	[30]
Dynamic upgrade (process)	A upgrade process that performs changing a given target system on-line while the system is operating, as compare to a static upgrade in which the system is taken off-line during reconfiguration.	[30]
Multicast Method Invocation	The semantics of a multicast invocation guarantees that the invocation on a replicated server will be executed by invoking the same method on every server replica [contained in the client' partition in case of partitionable environment].	[63]
Object	A model of an entity, which represents either a real or abstract phenomenon. An object can be distinguished from any other object and is characterized by its behavior and state.	[45][46]
OO-Framework	A partial system implementation in object-oriented way formed by a set of interacting classes expressing the behavioral and structural patterns typical for a certain	

	problem domain. The framework is extendable by adding new, specialized classes to turn the framework into a concrete implementation for a system.	
Package	[in UML sense] A collection of arbitrary model elements used to structure the entire model into smaller, clearly visible units. A package also defines a namespace for these model elements.	[81]
Passively-replicated Server	A server replicated with a number of replica, one of which (called the primary replica) is processing the incoming client requests and sending updates to the rest of the other replicas (called backup replicas).	
Platform	A basic technology enabling inter-process and intra-process interactions of software components in the distributed system. Examples of commercial middleware platforms include Sun's EJB[119] and OMG's CORBA[73]. Synonyms: Middleware Platform, Distributed Execution Environment[126]	
Policy	Policy allows for specifying or declaring a condition under which a given management process, in particular a dynamic upgrade, should be carried out.	
Recovery	(1) Recovery is a process of leading to resuming the system operation after an outage. (2) Recovery is the speed with which a system returns to operation following an outage.	[134]
Reference Point	A reference point is the specification of a particular set of conformance requirements. It comprises the set of interfaces that describe the interactions that take place between entities.	[126]
Referential Integrity	Rule which describes the integrity of object or component relations. In the context of an dynamic upgrade if one component is upgraded, that is replaced with another one, the relation realizations (e.g. references) that other components use to access the component upgraded have to be updated correspondingly.	[81]
Runtime Constraints (of a dynamic upgrade process)	Constraints on the quality of dynamic upgrades. They can range from the basic need to have a quality upgrade to a requirements for safe uninterrupted service of an online system, even in the event of an error in the upgraded software, hardware, or process.	[30]
Service Deployment (process)	The process involving actions aimed at making a piece of software representing the service functionality available for using in a given target infrastructure Deployment process involves requirements a number of activities including: deployment requirement matching, code distribution, installation, configuration and activation.	
Service Release	The process of making a service available to be deployed	

	in a given physical infrastructure.
Service Withdrawal	The process of making a service unavailable to be deployed in a given physical infrastructure
Static upgrade (process)	A *upgrade process* that performs changing a given target system off-line. The system is not operational during the *upgrade time*.
Upgrade Algorithm	An algorithm describing the steps to upgrade a given upgrade target.
Upgrade Process	Upgrade process is a management process that deals with exchanging programming artifacts comprising a target system. In the context of this thesis, the target system is component-based distributed system.
Upgrade Time	One of the management dimensions of the upgrade process. Planned Upgrade Time is a point in time, in which the upgrade process for the given upgrade target should start, whereas Real Upgrade Time is the point of time when the upgrade process for the given target starts.
Upgrade Target	The object of a upgrade process. In context of dynamic upgrades of distributed system, it is defined as as set of runtime artifacts (component instances), possibly distributed on various nodes.
Upgrade Transparency	A feature of a DUSS. It masks, from an object, an upgrade being performed on other object(s) in the so called upgrade zone. The upgrade transparency can be considered as another ODP distribution transparency [46].
Upgrade zone	A set of containers, which may be located on various container servers, in which the deployed components are to be upgraded.

Bibliography

[1] Arnold K., Gossling J.: The JavaTM Programming Language, Third Edition, Addison-Wesley Pub Co; ISBN: 0201704331, June 2000.

[2] Bank, D. Shub Ch., Sebesta R.: A unified Model of Pointwise Equivalence of Procedural Computations, ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6, November 1994, pp. 1842-1874.

[3] Bäumer, C. ; Breugst, M. ; Choy, S. ; Magedanz, T.: Grasshopper - A Universal Agent Platform based on OMG MASIF and FIPA Standards, MATA ´99 First International Workshop on Mobile Agents for Telecommunication Applications, Karmouch, A.[Hrsg.]), 1999

[4] BCEL, Byte Code Engineering Library, http://jakarta.apache.org/bcel/

[5] Bernstein, P.: Middleware: A Model for Distributed Systems Services, Communications of ACM, Vol. 39, No. 2, 1996, pp. 86 - 98.

[6] Birman K.: Building Secure and Reliable Network Applications, Manning Publications Co., 1996.

[7] Bloom T.: Dynamic Module Replacement in a Distributed Programming System, Ph.D. dissertation, MIT/LCS/TR-303, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1983.

[8] Bloomer J., Power Programming with RPC, O'Reilly & Associates Inc., February 1992.

[9] Bricker A., Litzkow M., Livny M.: Condor Technical Summary. Technical report, Computer Sciences Department, University of Wisconsin-Madison, January 1992.

[10] Carnegie Mellon, Software Engineering Institute: Software Technology Review, http://www.sei.cmu.edu/str

[11] Debian homepage, http://www.debian.org

[12] Dolev S., Self-Stabilization, The MIT Press, Cambridge, MA, 2000.

[13] Douglis F.: Transparent Process Migration: Design Alternatives and the Sprite Implementation, Software Practice and Experience, 21 (8), 757-785, August 1991.

[14] Eclipse: http:http://www.eclipse.org

[15] Evans H., Dickman P.: DRASTIC: A Run-Time Architecture for Evolving, Distributed, Persistent Systems. ECOOP'97.

[16] Evans H., Dickman P.: Supporting Software Evolution in a Distributed, Persistent System, ERSADS'97, Switzerland, March 1997.

[17] Eurescom Project P910 "Technology Assessment of Middleware for Telecommunications", Deliverable 5: Report on Dependability and Scalability of Middleware Platforms, March 2001, http://www.eurescom.de/public/projects/P900-series/p910/P910.

[18] Eurescom Project P910 "Technology Assessment of Middleware for Telecommunications", Deliverable 7: Report on Telecommunication Application Domains, March 2001, http://www.eurescom.de/public/projects/P900-series/p924/P924.

[19] Eurescom Project P924-PF "Distribution and Configuration Support for Distributed PNO Applications", Deliverable 1: Requirements, Terminology, and Concepts, June 2000, http://www.eurescom.de/public/projects/P900-series/p924/P924.

[20] Eurescom Project P924-PF "Distribution and Configuration Support for Distributed

PNO Applications", Deliverable 2: Notation and semantics for deployment and configuration, July 2001, http://www.eurescom.de/public/projects/P900-series/p924/P924 .

[21] Fabry R.S.: How to design a system in which modules can be changes on the fly?, Proc. Int'l Conf. Software Eng., IEEE-CS Press, Los Alamitos, Calif., 1976, pp. 470-476.

[22] FAIN Consortium: Deliverable D4, Revised Active Node Architecture and Design, May 2002, http://www.ist-fain.org.

[23] FAIN Consortium: Deliverable D8, Final Specification of Case Study Systems, May 2003, http://www.ist-fain.org.

[24] Fatoohi R., McNab D., and Tweten D.: Middleware for Building Distributed Applications Infrastructure: A State-of-the-art Report on Middleware, Nasa Ames Research Center, Technical Report NAS-97-026, December 1997. http://www.nas.nasa.gov/Research/Reports/Techreports/1997/nas-97-026-abstract.html

[25] Finin T., Labrou Y., Mayfield J.: KQML as an agent communication language, invited chapter in Jeff Bradshaw (Ed.), ``Software Agents", MIT Press, Cambridge, 1995.

[26] FIPA, The Foundation for Intelligent Physical Agents, http://www.fipa.org

[27] Galis A., Plattner B, Smith J., Denazis S., Moeller E., Guo H., Klein C., Serrat J., Laarhuis J, Karetsos G. and Todd Ch.: A Flexible IP Active Networks Architecture, IWAN'2000, Tokyo, Proceedings in Lecture Notes in Computer Science 1942 Springer 2000, ISBN 3-540-41179-8, October 2000.

[28] Gamma E., Helm R., Johnson R., Vlissides J.: Design Patterns: Elements of Reuseable Object-Oriented Softawre, Addison Wesley, Reading, 1995.

[29] General Magic's Odyssey, http://www.genmagic.com/technology/odyssey.html.

[30] Gluch D., Weinstock Ch. (ed.): Workshop on the State Of the Practice in Dependably Upgrading Critical Systems, Special Report CMU/SEI-97-SR-014, April 1997.

[31] Goullon H., Isle R., Löhr K.: Dynamic Restructuring in an Experimental Operating System, IEEE Trans. Software Eng., July 1978, pp. 298-307.

[32] Gray J., Siewiorek D.P.: High-Availability Computer Systems, IEEE Computer, September 1991, pp. 39-48.

[33] Grisby D.: ORB Support for the CORBA LifeCycle Service, March 1998.

[34] Gupta D.: On-line Software Version Change, PhD Thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, November 1994.

[35] Gupta D., Jalote P., Barua G.: A Formal Framework for On-line Software Version Change, IEEE Transactions on Software Engineering, Vol. 22, No. 2, February 1996, pp.120-131

[36] Gupta D., Jalote P.: A Formal Framework for On-line Software Version Change, IEEE Transactions on Software Engineering, Vol. 22, No. 2, February 1996, pp.120-131.

[37] Hauptmann S., Wasel J.: On-line Maintenance with On-the-fly Software Replacement, Proc. Of the 3rd Intern'l. Conf. On configurable distributed systems, ICCDS, May 1996.

[38] Haas R., Droz P., Stiller B.: Distributed Service Deployment over Programmable Networks, DSOM01, Nancy, France, 2001.

[39] Heiks M.: Dynamic Software Updating, Dissertation in Computer and Information Science, University of Pennsylvania, 2001.

[40] Hoff, van A., Partovi H., Thai T.: W3C Document: The Open Software Description Format (OSD), http://www.w3.org/TR/NOTE-OSD.html, August 1997.

[41] Hofmeister Ch.R., Dynamic Reconfiguration of Distributed Applications, Ph.D. dissertation, UMIACS-TR-94-8, Department of Computer Science, University of

Maryland. 1994.

[42] Hofmeister Ch.R., Purtilo J.M.: A Framework for Dynamic Reconfiguration of Distributed Systems, UMIACS-TR-93-78, Department of Computer Science University of Maryland, 1993.

[43] IBM Corp.: IBM San Francisco Business Process Components for Java, 1998.

[44] Institute of Electrical and Electronics Engineers: IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.

[45] ITU-T Recommendation X.902 | ISO/IEC 10746-2: "Open Distributed Processing – Reference Model Part 2", ITU-T/ISO, 1995.

[46] ITU-T Recommendation X.903 | ISO/IEC 10746-3: "Open Distributed Processing – Reference Model Part 3", ITU-T/ISO, 1995.

[47] Junit: http://www.junit.org

[48] Kath O.: CORE - Komponentenorientierte Entwicklung offener verteilter Softwaresysteme im Telekommunikationskontext, Band III – Platformunterstützung und Abteilungsregeln für Softwarekomponenten, Humboldt Universität zu Berlin, Institut für Informatik, July 2001.

[49] Kramer J., Magee J.: The Evolving Philosophers Problem: Dynamic Change Management, IEEE Transactions on Software Engineering, vol. 16, no. 11, pp. 1293-1306, 1990.

[50] Lange D.: Mobile Objects and Mobile Agents: The Future of Distributed Computing?, In Proceedings of The European Conference on Object-Oriented Programming '98, June 1998.

[51] Lange D., Oshima M.: Programming and Deploying Java™ Mobile Agents with Aglets™, Addison-Wesley, 1998.

[52] Larman C.: Applying UML and Patterns, 2nd Ed., Prentice Hall, ISBN 0-13-092569-1, 2002.

[53] Liang S., Bracha G.: Dynamic Class Loading in the Java Virtual Machine, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'98, Vancouver, Canada, October 1998.

[54] Linux kernel homepage, http://www.kernel.org

[55] Liskov B., Wing J.: A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems, Vol. 16, No. 6, pp. 1811-1841, November 1994.

[56] Luckham D., Vera J.: An event-based architectural definition language, IEEE Transactions on Software Engineering, pp. 717-734, September 1995.

[57] Magee J., Kramer J.: Dynamic structure in software architectures, Fourth SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, October 1996.

[58] Mattieu B., Carlet Y., Solarski M. et.: Deployment of active services, *World Telecommunications Congress 2002*, Paris, France, September 2002.

[59] Meling H. and Helvik B.E.: ARM: Autonomous Replication Management in Jgroup, In Proc. of the 4th European Research Seminar on Advances in Distributed Systems, Bertinoro, Italy, May 2001.

[60] Mishra S., Schlichting R.: Abstractions for Constructing Dependable Distributed Systems, Dept. of Computer Science, Univ. of Arizona, TR-92-19, 1992.

[61] Moeller E., Solarski M.: "Challenges in Active Service Deployment", ANTA'2002 (The First International Workshop on Active Network Technologies and Applications), Tokyo, Japan, March 2002.

[62] Morgan G., Shrivastava S.K., Ezhilchelvan P. D., Little M. C.: *Design and Implementation of a CORBA Fault-Tolerant Object Group Service*, Proceedings of the Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99, Helsinki, June 1999.

[63] Montresor A.: System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems, PhD thesis, Dept. of Computer Science, University of Bologna, Feb. 2000.

[64] Moore B., Ellesson E., Strassner J., Westerinen A.: Policy Core Information Model -- Version 1 Specification, Request for Comments: 3060, IETF, Network Working Group, February 2001.

[65] Mortazavi M., Lange U.: Design and Implementation of Mobile Communications Services as Applications for the J2EE platform, Java One, Sun's 2001 WorldWide Java Developers Conference, Session 1020.

[66] Moser L.E, Melliar-Smith P.M., Narasimhan P., Tewksbury L. and Kalogeraki V.: *Eternal: Fault Tolerance and Live Upgrades for Distributed Object Systems*, Proceedings of the IEEE Information Survivability Conference, Hilton Head, SC, January 2000.

[67] Mullender S (ed.).: Distributed Systems, Frontier Series. Addison-Wesley, first edition, ACM Press, 1989.

[68] Narasimhanm P., Moser L. E. and Melliar-Smith P. M.: *Exploiting the Internet Inter-ORB protocol interface to provide CORBA with fault tolerance*, Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, OR, June 1997, pp. 81-90.

[69] Narasimhan P., Moser L. E. and Melliar-Smith P. M.: *Replica consistency of CORBA objects in partitionable distributed systems*, Distributed Systems Engineering, vol. 4, no. 3 September 1997, pp. 139-150.

[70] No Magic: Magic Draw, http://www.magicdraw.com

[71] Object Space: Voyager, http://www.objectspace.com/products/voyager/

[72] Object Management Group: CORBA Basics, http://www.omg.org/gettingstarted/corbafaq.htm

[73] OMG Document. formal/02-06-01: The Common Object Request Broker: Architecture and Specification, Revision 3.0, July 2002.

[74] OMG TC Document: ORB Interface Type Versioning Management, Request for Proposal, 1996.

[75] OMG TC Document: orbos/99-10-05: Fault Tolerant CORBA, Joint Revised Submission, 1999.

[76] OMG TC Document: orbos/01-08-08: Online Upgrades, Request for Proposal, Version 1.0, August 2001.

[77] OMG TC Document: orbos/00-09-15: Online Upgrades, Request for Information, Sept. 2000.

[78] OMG TC Document: ptc/00-04-0: Fault Tolerant CORBA, April 2000.

[79] OMG TC Document: orbos/02-01-01: Online Upgrades, Initial Submission, January 2002.

[80] OMG TC Document: mars/02-05-01: Online Upgrades, Revised Joint Proposed Specification, May 2002.

[81] Oestereich B.: Developing Software with UML, Object-Oriented Analysis and Design in

Practice, Addison Wesley, Addison Wesley Professional, ISBN: 020175603X; 2nd edition July 2002.

[82] OpenORB, http://www.openorb.org.

[83] Oreizy P., Medvidovic N., Taylor R.: Architecture-Based Runtime Software Evolution, Intl. Conf. On Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 1998.

[84] Orfali R., Harkey D., Edwards J., Harkey D.: The Essential Distributed Objects Survival Guide, John Wiley & Sons; ISBN: 0471129933, Sept. 1996.

[85] Orfali R., Harkey D., Edwards J., Harkey D.: Instant CORBA, John Wiley & Sons; ISBN: 0471183334, March 1997.

[86] Palsber J., Schwartzbach M.: Three Discussions on Object-Oriented Typing, Report on ECOOP'91 Workshop W5 on "Types, Inheritance, and Assignments", Geneva, Switzerland, July 1991.

[87] Peterson J., Hudak P., Ling G.: Principled Dynamic Code Improvement, Yale University, Research Report YALEU/DCS/RR-1135, July 15, 1997.

[88] Plasil F., Balek D., Janecek R.: Dynamic Component Upgrading in Java/CORBA Environment, Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague.

[89] Plasil F., Balek D., Janecek R.: SOFA/DCUP: An Architecture for Component Trading and Dynamic Upgrading. ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA.

[90] Powell D. (ed.).: Delta-4: A Generic Architecture for Dependable Distributed Computing, Vol. 1, Springer-Verlag, Berlin, 1991

[91] Pyarali I., Schmidt D.: An Overview of the CORBA Portable Object Adaptor, ACM StandardView, 1998.

[92] Rational Rose: Rational Unified Process, http://www.rational.com

[93] Reilly D.: Mobile Agents –Process Migration and its Implications, November 1998, http://www.davidreilly.com/topics/software_agents/mobile_agents/

[94] Riggs R.: Java Product Versioning Specification, part of the Sun's JDK 1.2 specification, Sun, Dec. 1997.

[95] Rosenberry, W., Kenney, D., and Fisher, G.: Understanding DCE, O'Reilly & Associates, Inc., 1992.

[96] Rumbaugh J., Jacobson I., Booch G.: The Unified Modeling Language, Reference Manual, Addison Wesley, ISBN: 0201571684; 1st edition, October 1998.

[97] Sayegh A.: CORBA: Standard, Spezifikation, Entwicklung, 2nd Ed., O'Reilly, Köln, ISBN: 3-89721-156-4, 1999.

[98] Schmidt D., Vinoski S.: Object Adaptors: Concepts and Terminology, SIGS C++ Report, October 1997.

[99] Schneider F.B.: Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial, ACM Computing Surveys, Vol. 22, No 4, Dec 1999, pp. 299-319.

[100] Schulzrinne H.: Internet Telephony: A Second Chance, in Butscher B., Carle G., Kuthan J., Smirnov M., Stüttgen H., Wolf L. (Eds): Proceedings of the 1st IP-Telephony Workshop (IPTel 2000), Berlin, April 2000, pp. 9-12.

[101] Segal M., Frieder O.: Dynamically Upgrading Distributed Software: Supporting Change in Uncertain and Mistrustful Environments, IEEE Conference on Software Maintenance, Miami, Florida, October 1998.

[102] Segal M., Frieder O.: On Dynamically Upgrading a Computer Program From Concept to

Prototype, Journal of Systems Software, 1991, 14, pp.111-128.

[103] Segal M., Frieder O.: On-the-fly Program Modification: Systems for Dynamic Upgrading, IEEE Software, March 1993, pp. 53-65.

[104] Sethi R., Programming Languages – Concepts and Constructs, Addison Wesley, 1989.

[105] Sinnott R.: An Architecture Based Approach to Specifying Distributed Systems in LOTOS and Z, University of Stirling, Department of Computer Science and Mathematics, April, 1997.

[106] Smith P.: The possibilities and Limitations of Heterogeneous Process Migration, Ph.D., The Faculty of Graduate Studies, University of British Columbia, October 1997.

[107] Solaris man pages, Sun Microsystems.

[108] Solarski M.: Dynamic Upgrading of Software Components in TINA-based Systems, ERCIM News No.35, October 1998.

[109] Solarski M.: „Dynamic Upgrade: a method to increase system availability “, Eurescom WS: „Middleware in Telecommunication: Facilitating the Open Services Marketplace“, Kjeller, Norway, March 2001.

[110] Solarski M.: “Upgrading actively replicated objects on the fly”, IEEE CQR’2001 (Workshop of Technical Committee on Communications Quality & Reliability), Tucson, AZ, April 2001.

[111] Solarski M.: “Updating Distributed Components on the fly”, Eurescom P924 WS, Berlin, Nov. 2000.

[112] Solarski M., Bossardt M., Becker T.: Component-based Deployment and Management of Services in Active Networks, IWAN 2002, Zürich, CH, Dec. 2002.

[113] Solarski M., Meling H.: Towards Upgrading Actively Replicated Servers on-the-fly, COMPSAC’02, Workshop on Dependable On-line Upgrading of Distributed Systems, Oxford, UK, August, 2002.

[114] Solarski M., Sinnott R., Durmosch K. and Hafezi A.: *Reference Integrity in Dynamic CORBA Components,* SRDS’99: WREMI'99 Workshop on Reliable Middleware Systems, October 1999.

[115] Somani A., Vaidya N.: Understanding Fault Tolerance and Reliability, Computer, April 1997, pp. 45-50.

[116] Steensgaard B., Jul E.: Object and Native Code Thread Mobility Among Heterogeneous Computers, In Symposium on Operating System Principles, 1995.

[117] Sun Microsystems, Inc.: Java Beans, Version 1.00, October 1996, http://java.sun.com/products/beans

[118] Sun Microsystems, Inc.: Java Language Specification, Second Edition, April 2000.

[119] Sun Microsystems, Inc.: Java 2 Enterprise Edition: Enterprise Java Beans, http://java.sun.com/products/ejb

[120] Sun Microsystems, Inc.: Java Community Process, http://jcp.org

[121] Sun Microsystems, Inc.: Java Specification Request, JSR 117, J2EE APIs for Continuous Availability, April 2001, http://www.jcp.org/en/jsr/detail?id=117

[122] Sun Microsystems, Inc.: Java 2 Standard Edition: Remote Method Invocation, http://java.sun.com/products/jdk/rmi/

[123] Szyperski C.: Component Software - Beyond Object-Oriented Programming, ACM Press, New York, 1998.

[124] Tewksbury L. , Moser L., Melliar-Smith P.: Live Upgrade Techniques for CORBA

Applications, DAIS'2001, Krakow, September, 2001.

[125] Tennenhouse D.L., Wetherall D.J.: Towards an Active Network Architecture, Multimedia Computing and Networking, San Jose, CA, January 1996.

[126] TINA-C, Overall Concepts and Principles of TINA, February 1995.

[127] Tivoli Application Management Specification, 1998, http://www.tivoli.com/o_partners/html/body_ams_spec.html

[128] Udell J.: ComponentWare, Byte Magazine, 19(5), 1994, pp. 45-56.

[129] Van Hoff A. et al, The Open Software Description Format (OSD), http://www.w3c.org/TR?NOTE-OSD.html

[130] Visigenic Visibroker for Java, Programmer's Guide, Version 3.0, September 1997.

[131] Wegner P.: Dimensions of object-oriented language design, Proceedings of OOPSLA'87, ACM SIGPLAN Notices, 22(12), pp. 168-182, October 1987.

[132] Yajnik S., Huang Y.: STL: A toolkit for On-line software upgrade and Rejuvenation," presented at the Industrial Track session in International Symposium on Software Reliability Engineering Nov 1997.

[133] Zeller A.: Versioning Software Systems through Concept Descriptions, Technical Report 97-01, Universität Braunschweig, Jan 1997.

[134] Zhu J., Mauro J., Pramanick I.: R-Cubed R3: Rate, Robustness and Recovery – An Availability Benchmark Framework, Sun Microssystems, Technical Report SMLI TR-2002-109, July 2002.

Index