AGH — University of Science and Technology
in Kraków, Poland

Faculty of Electrical Engineering, Automatics, Computer Science
and Electronics
Institute of Computer Science

Adaptive Deployment of Component-based Applications in Distributed Systems

A dissertation for the degree of
Doctor of Philosophy

author: MSc. Eng. Jacek Cała
supervisor: Prof. Dr. Eng. Krzysztof Zieliński

January 2010

Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

Wydział Elektrotechniki, Automatyki, Informatyki i Elektroniki
Katedra Informatyki

Rozmieszczanie adaptacyjne aplikacji komponentowych w systemach rozproszonych

Rozprawa doktorska

autor: mgr inż. Jacek Cała
promotor: prof. dr hab. inż. Krzysztof Zieliński

Styczeń 2010

To my Mother

Abstract

Deployment of distributed applications in heterogeneous environments is an interesting yet complex area in the software life cycle. A proper deployment infrastructure can alleviate many important issues related to software execution and management such as finding suitable location for application components and automation of low-level deployment tasks. It also promotes component-based software design and enables creating more sophisticated dynamic and adaptive solutions. Applying adaptation to the software deployment process has great potential. Generally, it allows reacting to context changes and reorganizing application components to improve their execution. Specifically, it may support ubiquitous environments, autonomic computing solutions and highly available systems. This work presents design, implementation and evaluation of the Adaptive Deployment Framework (ADF) created in the course of our research in this area.

The key role in adaptive deployment plays the model-based approach to software deployment. By separation between a model of software and a model of execution environment, it improves reusability and enables automation of the deployment process. However, many of the existing model-based solutions are limited to the spatial distribution of application components in the execution environment. We extended the notion of deployment and defined three basic deployment dimensions: spatial, temporal and semantic. Deployment can be considered in each of these dimensions separately but also the dimensions can be combined together creating more elaborated deployment scenarios.

One of the important requirements that enable adaptive deployment is availability of reconfiguration mechanisms. What mechanisms are needed, however, depends on the way how deployment update is performed. We distinguished four possible redeployment techniques: *full*, *deep*, *shallow* and *runtime redeployment*. In this thesis we concentrate on runtime redeployment which is supposed to guarantee the most *agile* adaptive deployment system. To realize runtime redeployment we designed, implemented and evaluated runtime component migration mechanism. It is the foundation for the ADF framework. We found that the component level and particularly the CCM model, used as a basis for application design, is very well suited for migration and enables effective deployment adaptation.

For the purpose of evaluation of our Adaptive Deployment Framework we designed and implemented Force-Directed Deployment Planning (FDDP) a novel approach to deployment planning. It demonstrates that adaptive deployment can be successfully used to improve application performance.

Acknowledgements

I would like to thank my supervisor Prof. Krzysztof Zieliński for his ideas and constructive advice that allowed me to complete this thesis.

To my colleagues from Distributed Systems Research Group for creating friendly work atmosphere and specially to Dr. Łukasz Czekierda to whom I am deeply indebted for his invaluable help in the most critical moments of writing this dissertation.

I am also very grateful to my wife Ania for her continuous support and motivation on the long journey.

Last but not least, I wish to thank Prof. Paul Watson for his surprising indulgence during my work at Newcastle University.

Jacek Cała

Contents

1 Introduction 1

1.1 Motivation . . . 2
1.2 Approach and Challenges . . . 4
1.3 Thesis Statement and Objectives . . . 7
1.4 Thesis Contributions . . . 8
1.5 Roadmap . . . 8

2 Background and Related Work 10

2.1 Definition of the Deployment Process . . . 12
2.2 Deployment Automation . . . 15
2.2.1 Deployment Automation on a Single Machine . . . 15
2.2.2 Deployment Automation in Distributed System . . . 18
2.2.3 Deployment Automation in Virtualized Environments . 25
2.2.4 Deployment Automation in Grids . . . 27
2.2.5 Deployment Automation in SOA . . . 28
2.3 Deployment Planning . . . 29
2.3.1 Definition of Deployment Planning . . . 30
2.3.2 Planning Dimensions . . . 31
2.3.3 Complexity of Deployment Planning . . . 35
2.4 Adaptive Deployment . . . 39
2.4.1 Definition of Adaptation . . . 40
2.4.2 Benefits of Adaptive Deployment . . . 41
2.4.3 Reflective Systems . . . 43

2.4.4 Autonomic Computing 44
2.5 Adaptive Deployment Platforms 45
2.6 Conclusions 53

3 Towards Adaptive Software Deployment 59
3.1 Plain Deployment Platform 61
3.1.1 Support for Component-based Applications 61
3.1.2 Support for Deployment in Distributed Systems 62
3.1.3 Support for Deployment Planning Dimensions 64
3.1.4 Support for Virtualization 69
3.2 Support for Adaptation 72
3.2.1 Requirements for Adaptive Deployment 72
3.2.2 Monitoring Facilities 73
3.2.3 Reconfiguration Mechanisms 74
3.2.4 Adaptation Control Loop 77
3.3 Summary 78

4 Adaptive Deployment Framework 79
4.1 The Model of Deployable Components 80
4.2 Overview of the Framework 81
4.3 Plain Deployment Infrastructure 83
4.3.1 Repository Manager 84
4.3.2 Target Manager 84
4.3.3 First-stage Planner 85
4.3.4 Deployment Plan Execution 88
4.4 Adaptive Deployment Infrastructure 91
4.4.1 The Management Layer — Sensors and Effectors . . . 92
4.4.2 The Adaptation Layer 99
4.5 Framework Usage Scenario 102
4.6 Summary 105

5 Monitoring and Management Infrastructure 107

5.1 Support for Runtime Component Migration 108
5.1.1 Suspension and Dealing with Requests 110
5.1.2 Factory Support for Reconnection 112
5.1.3 Life Cycle of a Mobile Component 115
5.1.4 Passivation During Synchronous Requests 119
5.1.5 Summary 121
5.2 COPI-based Application Monitoring 123
5.3 Component Instance Identification 124
5.4 Summary 127

6 Evaluation of the Framework Building Blocks 129
6.1 Configuration of the Testing Environment 129
6.2 Testing Applications 130
6.2.1 Traffic Generator 130
6.2.2 Asymmetric Ray Tracing 131
6.3 Evaluation of the Plain Deployment Infrastructure 132
6.3.1 Conformance to the D&C Specification 133
6.3.2 Performance of the Deployment Infrastructure 136
6.3.3 Possible Extensions 139
6.4 Performance of the Migration Mechanism 140
6.4.1 Effectiveness of the Migration Mechanism 140
6.4.2 Influence of Migration on Communication Performance 142
6.4.3 Influence of Migration on Processing Performance . . . 143
6.4.4 Overhead of the Migration Infrastructure 147
6.5 Overhead of Monitoring Infrastructure 148
6.6 Summary and Conclusions 150

7 Adaptive Deployment with Force-Directed Algorithms 152
7.1 Overview of FDA algorithms 153
7.2 Force-Directed Deployment Planning 154
7.2.1 Graph Representation of the Deployment Problem . . 155

7.2.2 Forces in FDDP 156
7.2.3 Mapping of Observables on Model Parameters 159
7.2.4 Experimenting with the FDDP Model 160
7.3 Evaluation of the Adaptive Deployment Framework 161
7.3.1 Using the ADF Framework 161
7.3.2 Costs of Runtime Adaptation 163
7.3.3 Application Performance 165
7.3.4 Adaptation to an External Disturbance 167
7.4 Limitations of the FDDP Planner 170
7.5 Summary and Conclusions 171

8 Conclusions and Possible Research Directions 173

A IDL interfaces 176

B Description of an Execution Environment 180

C Support for the Planning Dimensions 184

Bibliography 187

Acronyms 199

List of Tables

2.1 Main challenges for adaptive deployment framework and how they are met by existing solutions. 54

3.1 An example of mapping of an execution node and software component entities for selected virtualization levels. 71

4.1 Preferred BFS heuristics for initial deployment planning. . . . 86

4.2 CIM-based sensors provided by the framework. 95

6.1 The software and hardware configuration of the testing environment. 129

6.2 The number and percentage of the entities and operations defined in D&C that are implemented by our deployment infrastructure. 136

6.3 Time required to complete a single run of the ART application with and without deployment updates. 137

6.4 Time required to update deployment of the ART application depending on the target execution nodes. 138

6.5 Time required to perform subsequent migration steps when moving a component. 141

6.6 Time required to perform a ray tracing task depending on chunk size, `Worker` location and its mobility. 144

6.7 Execution time of the ART application for the original and mobility-aware OpenCCM platforms. 148

6.8 Execution time of the ART application depending if the WBEM infrastructure and CIM-based sensors were enabled. 149

6.9 Execution time of the ART application depending if the COPI infrastructure and COPI-based sensors were enabled. 149

6.10 Migration time of the `Runner` component with `HomeSensor` disabled and enabled. 150

7.1 Execution time of the ART application with the ADF infrastructure disabled and enabled. 164

7.2 Selected static and initial deployments together with the measured execution time of the ART application. 165

7.3 Execution time for selected static deployments of the ART application in comparison to the application managed by ADF. 170

List of Figures

2.1 Deployment planning effectiveness in contrast with flexibility of component deployment. 11

2.2 Installation of application components in a distributed execution environment. 12

2.3 Activities in adaptive software deployment. 13

2.4 The life cycle of a CIM Software Element. 23

2.5 The multilevel virtualization model. 25

2.6 Four steps of deployment of SOA applications. 29

2.7 Three dimensions of software deployment planning. 31

2.8 A coarse illustration of the software package structure defined in D&C. 36

2.9 Building blocks of the Autonomic Computing model. 45

3.1 The key aspects leading towards the adaptive software deployment platform. 60

3.2 The global view on deployment activities. 62

3.3 The local view on deployment activities. 63

3.4 Separation between models of the software application and execution environment enables automatic planning. 64

3.5 General view on the spatial planning dimension. 65

3.6 General view on the semantic planning dimension. 66

3.7 General view on planning in the temporal dimension. 67

3.8 A sample application with temporal dependencies represented as a DAG and using the `TemporalCollocation` structures. . . . 68

3.9 The multilayer structure of deployment models in virtualized environments. 70
3.10 The state diagram of the adaptive deployment process showing four adaptation techniques. 75

4.1 Illustration of the general concept of a component in the CCM model. 81
4.2 The global view on the architecture of our ADF framework. . . 82
4.3 The local view on the architecture of our ADF framework. . . . 83
4.4 Implementation of the `RepositoryManager` element. 84
4.5 A collaboration diagram showing deployment plan execution. 89
4.6 The interface enabling adaptation of application deployment. 90
4.7 Two layers of the adaptive deployment framework. 91
4.8 The model of adaptive deployment framework showing separation between application and environment layers. 92
4.9 The general design of the sensor and effector components. . . 93
4.10 A sequence diagram of successful migration between HERE and THERE locations. 97
4.11 The relation between a CCM component, container, component server and a hosting process. 99
4.12 The general design of the `AdaptationManager` component. . . 100
4.13 The collaboration diagram showing how migration process is observed by the `AdaptationManager`. 101
4.14 Two-step interaction between the `AdaptationManager` and the deployment infrastructure during reconfiguration. 103
4.15 The flow diagram showing how ADF may be used to manage deployment of a software application. 104

5.1 Four possible cases of when dealing with requests during object passivation. 110
5.2 The most common scenarios of collaboration between migration effector and mobile component factories. 116
5.3 Two options of handling passivation requests when a composite operation is in progress. 120

5.4 A solution to the passivation problem while a composite operation is in progress. 122
5.5 The general view on the basic and extended level of COPIs. . . 124
5.6 The detailed view on request transmission between client and server components when the COPI infrastructure is enabled. . 124
5.7 The extensions creating a link between a running component instance and a deployment plan. 126

6.1 The topology of the testing environment. 130
6.2 The architecture of the Traffic Generator application. 131
6.3 The architecture of the Asymmetric Ray Tracing application. . 132
6.4 Extensions of the *Execution Management Model* related to runtime application reconfiguration. 134
6.5 Target environment used for testing time required to prepare reconfiguration of the ART application. 138
6.6 Influence of the reconfiguration preparation step on adaptation agility. 139
6.7 The deployment of the Traffic Generator application among five Sun Blades servers. 141
6.8 Deployment of components when testing influence of migration on call performance. 142
6.9 Influence of migration on the number of operation invocations per second and invocations' RTT. 143
6.10 Deployment of components during the test that verified influence of component migration on its processing performance. . 144
6.11 Performance overheads incurred by the ART application related to a single `Worker` move. 144
6.12 Influence of migration on application processing time. 146
6.13 Deployment of the Asymmetric Ray Tracing application to measure overheads of the our migration-aware CCM platform. 147

7.1 Examples of graphs that represent an execution environment and a component-based application in FDDP. 155
7.2 An example of a graph representing application deployment in FDDP. 156

7.3 An illustration of repulsive and attractive forces between vertices of a graph representing an execution environment in FDDP. 157
7.4 The GUI of the prototype adaptation manager component. . . 162
7.5 Deployment of the ART application together with ADF to verify costs related to its operation. 164
7.6 A selected deployment of the ART application used for testing effectiveness of our ADF framework. 166
7.7 Execution time for different deployments of Asymmetric Ray Tracing. 167
7.8 Deployment of the application that played role of an external disturbance. 168
7.9 Distribution of application components in the first phase of application adaptation. 168
7.10 Distribution of application components that avoids an external disturbance. 169

C.1 Extensions to the D&C models enabling semantic deployment planning. 185
C.2 Extensions to the D&C models enabling temporal deployment planning. 186

Chapter 1

Introduction

Computing infrastructure has evolved over the last decades, moving from a mainframe-centric, batch processing model, through two- and three-tier client-server architectures, to recent very diverse distributed computing models. From the software perspective they include Component-Oriented Architecture (COA), Service-Oriented Architecture (SOA), Message-Oriented Middleware (MOM) and many others. In these diverse and often complex distributed environments the problem of software deployment becomes an important factor in deciding on software usability, performance and dependability. Anyone who tried to install, configure and run a Java Enterprise Edition application sever such as JBoss[1] or Glassfish[2] faced hundreds of pages of installation and administration guide. They include knowledge of how to install and connect the software with the rest of the infrastructure such as a database engine to achieve the desired functionality and performance. As Szyperski aptly expressed,[3] "deployment exists in the software life cycle to bridge the gap between what a software developer could not know about the execution environment and what the environment's developer could not know about the deployable software." Leaving this gap leads to many problems with software execution, performance and security. Conversely, proper tools can produce an application deployment well-suited to the execution environment and, as a result, can ease starting the system.

Deployment is a process that makes software provided by a producer available for use by a consumer. In its basic form it consists of software retrieval, installation and execution activities, whereas the full deployment process also includes software updating, reconfiguration and removal. Unfortunately, deployment in open and heterogeneous distributed systems is

[1] `http://www.jboss.org/jbossas`

[2] `https://glassfish.dev.java.net`

[3] C. Szyperski, Foreword to Proceedings of Component Deployment, IFIP/ACM Working Conference, Berlin 2002.

a non-trivial task. Unlike closed platforms (such as embedded systems or simple mobile phone OSs) that operate on a predictable set of applications and workloads, open systems offer users much more freedom in running software. This causes often unexpected changes in resource availability as different, unknown a priori applications compete for the same resources. If, in addition, the execution environment comprises heterogeneous resources, deployment becomes a computationally complex task because matching application components against execution hosts depends exponentially on the number of components to be deployed.

In this dissertation we show that by combining a component-based approach, a well defined deployment model and an adaptation mechanism we can alleviate these problems. The conducted research resulted in creating a comprehensive deployment model and its prototype implementation in the form of an adaptive deployment framework. The prototype offers tools for automated application deployment and runtime reconfiguration in open, distributed and heterogeneous systems. In this work we present design and implementation of our framework and its evaluation that focuses on application performance.

1.1 Motivation

The fact that deployment of distributed systems becomes a problem is clearly visible if we consider even simple client-server architectures with thick- and thin-client approaches.

Historically first, the thick-client architecture gives users higher usability and a better experience of software by making use of remote (server) and local (client) resources. For this reason the thick-client approach is the architecture of choice in the case of interactive applications, which are particularly susceptible to round trip time delays. Unfortunately, the thick-client approach suffers from a severe disadvantage — the problem of high management costs that stems directly form the lack of proper deployment models.

The answer to this problem has been the introduction of the thin-client architecture that provides a centralized software service. It requires much less effort to manage and maintain a system because the problem of deployment is limited to only one, central software repository.[4] Then, instead of struggling with management of a number of foreign customer sites a service provider is responsible for only one, their own system. This makes thin-client architectures a successful solution extensively used around the Internet to-

[4] We do not necessarily mean a single physical repository but rather a logical entity that behaves like a single physical repository.

day. Unfortunately, the weak point of this approach is lower usability and performance and often greater development effort. Although solutions such as Asynchronous Javascript And XML (AJAX), Shockwave Flash or Silverlight are trying to diminish these problems,[5] the thin-client approach is not a silver bullet. There are still many applications e.g. text, voice and video communication and peer-to-peer file sharing that are rarely used in a web browser environment.

Moreover, both thick- and thin-client approaches address only a fragment of the whole area of distributed system architectures. Today's distributed services are rarely built from only a client and a server but usually comprise many more application components. The component-based architecture gives greater flexibility in application design and shifts application development, deployment and management to a more granular level. Additionally, many component-based platforms offer multithreaded operation what is in line with today's multicore hardware platforms. Unfortunately, the higher flexibility offered by component-based design is also the main factor of deployment complexity. We argue, however, that a proper deployment infrastructure is the answer to the aforementioned problems for three reasons.

Firstly, the infrastructure can ensure that selected application components are running in the most (or close to the most) suitable locations. Secondly, it makes management of distributed software much easier by automation of many low-level deployment tasks. Thirdly, a well-defined deployment model allows for more sophisticated dynamic solutions such as on-demand deployment, deployment controlled by an application itself and also adaptive deployment. The last one enables reacting to changes in application execution context and can support many of the software adaptation needs. For example, data centres must survive hardware component failures or sudden surge in resource consumption and need to move computation to available nodes in runtime. A software deployer might not know a priori what is the best distribution of application components in an execution environment and would like to arrange them in runtime while visualising their interactions. Under high load some application components may need to be separated, whereas usually they perform better when collocated. For these and many more examples, adaptation of deployment can be the enabling technique. This motivated us to design and implement the Adaptive Deployment Framework.

[5]Recently, applications developed using these solutions are known under the common term Rich Internet Application (RIA).

1.2 Approach and Challenges

A software deployment may be related to many different software technologies and may be considered on different levels of system virtualization. As it is hardly possible to build a complete solution that fits any of these conditions, it is important to provide the assumptions we adopted when designing and implementing our framework. Following are the key points that have had the most significant influence on our approach and the research challenges we met.

Component-based middleware. Our solution is based on the CORBA Component Model (CCM) defined by OMG in [103]. This model offers many strong and valuable features that are of key importance for distributed software systems and its deployment such as: dependency injection, late binding, two-phase initialization, and event-driven programming. CORBA components can also be characterized by properties imposed by Szyperski in [119] as units of: *composition*, *state encapsulation*, and *independent deployment*. This makes a CCM-based application a good candidates for adaptive deployment.

The CCM technology locates our framework for adaptive application reconfiguration in the middle between low-level adaptation based on Virtual Machines (VMs) (as presented by Kotsovinos [70] and Kosiński [69]) and high-level adaptation of objects or language components (proposed e.g. in [64, 90] or by the ProActive project[6]). Comparing to the VM adaptation, our solution provides greater flexibility and lower overheads because a CCM component is a much smaller unit of reconfiguration than a Virtual Machine. Comparing to the objects and language components adaptation, the CCM technology concerns heterogeneity of software and hardware platforms providing a more general solution. Moreover, as Quema noticed in [108], large number of fine-grained classes generated during object-oriented modelling induces a large number of dependencies between them, thus making it difficult to take classes out of the context in which they were elaborated. This is especially important when deployment is considered.

At the middleware level an important research challenge was to propose a reconfiguration mechanism that will provide enough flexibility for adaptive deployment infrastructure and will fit the CCM model. The mechanism we realized is the runtime component migration that enables components to migrate between hosts without significant disturbance to overall application processing.

[6] `http://proactive.inria.fr`

Model-based deployment. The key implication of this deployment approach is clear separation between a model of application and a model of execution environment. The model-based deployment allows representing a structure of an application and execution environment in a declarative manner by means of an Architecture Description Language (ADL). It explicitly models components, its configurations, connectors, and requirements as well as execution entities, network interconnections and environment resources. The declarative approach allows hiding low-level aspects of deployment and freeing users from most of the work related to application management. In consequence, it enables not only automation of deployment planning but, what is more important for us, adaptation of the whole deployment process.

One of the most complete attempts to define a deployment and configuration process is the Deployment and Configuration of Component-based Applications (D&C) specification proposed by OMG in [100]. It defines many aspects of component deployment such as: component configuration, assembling, packaging and many others. The ADL proposed by the specification provides a general and expressive means for modelling software applications and execution environments. However, the D&C specification does not address issues related to dynamic deployment and dynamic reconfiguration. It also leaves undefined resource and requirement definition languages. Therefore, our research challenges were to propose and implement extensions of the D&C specification that support deployment reconfiguration as well as suggest a description language for environment resources and component requirements definition.

Runtime deployment planning. In open, heterogeneous and distributed environments optimal or suboptimal static deployment planning is futile because it is a computationally complex problem. Its complexity stems from several key facts: large problem search space that grows exponentially with the number of components, diversity of resources, changing component requirements that depend on application workload and changing resource availability that depends on workload consolidation. Consequently, the deployment process needs an approximate planning approach.

We based the design of our adaptive framework on the conviction that application reconfiguration and the process of its deployment very much depend on each other. Reconfiguration to be effectively realized needs a proper deployment infrastructure, whereas deployment supported by an approximate planning algorithm needs reconfiguration to apply changes in runtime. For that reason, one of the important qualities of the planner is its low computational complexity. Although there exist approximate solutions that solve the deployment planning problem in a polynomial time, we rose to the challenge and proposed a novel algorithm that suits open, heterogeneous

and distributed environments. FDDP is our solution to the planning problem based on the force-directed methods. Apart from the desired low complexity it may also be used as an engine for a visualization tool. FDDP produces nice layouts of application and environment graphs and provides users more insight into interactions between application components and execution environment.

Deployment planning in the spatial dimension. As presented later in Sect. 2.3.2 we defined three dimensions of deployment planning: spatial, temporal and semantic. Building a complete infrastructure for distributed applications and heterogeneous execution environments that embrace all these dimensions is very interesting task, yet complex and requiring a lot of effort. It is especially true when runtime deployment planning is considered because each of these dimensions requires a different approach to planning, monitoring and reconfiguration. Therefore, in this work we limited design and implementation of our adaptive deployment framework to the spatial dimension only. For planning in spatial dimension we monitor low-level resource utilization and data flow metrics, whereas e.g. the semantic dimension would require observing high-level Quality of Service (QoS) parameters.

Best-effort resource management. Considering resource management, the D&C specification defines a static resource reservation and management approach. This is, however, better suited to the stringent memory and performance constraints of Distributed Real-time and Embedded (DRE) systems. They often need to meet end-to-end latency or computation deadlines and explicit resource reservation is one of the means to achieve that [115]. In this work, however, we focus on deployment of enterprise applications in an open distributed environments where these constraints are usually much more relaxed. Consequently, we followed another approach to resource management and reservation.

Many previous examples showed that costs of resource reservation and reservation management are not always justified and lose with simple yet effective best-effort solutions.[7] When resource management is considered, the best-effort approach means that no additional management and reservation mechanisms exist. Therefore, in the case of extensive application workload or scarcity of resources the best-effort approach will lead to application exceptions and service unavailability. However, in most cases when availability of resources is high enough to carry existing application workload, no additional reservation mechanisms are required. This is particularly true in the

[7] This is especially visible when comparing networking mechanisms such as IntServ, Token ring, WiFi PCF with DiffServ, Ethernet and WiFi DCF respectively. Although the former provide proper reservation mechanisms, the latter are in common use today.

context of constantly falling hardware prices and increasing communication availability.

The best-effort approach to resource management during deployment is also especially reasonable when adaptation mechanisms are present. Then, instead of enforcing reservation policies, managing of resources can be effected by application adaptation. If, additionally, an application has a fine-grained component-based architecture, an adaptive deployment mechanism have enough means to ensure proper component distribution. What we show in this work is that, in many cases, provided with proper adaptation mechanisms no need for pessimistic resource allocation exists.

1.3 Thesis Statement and Objectives

The motivation, approach and challenges presented above allowed us to express the aim and main thesis of this work:

> *Modern component-based systems can be successfully enhanced with a runtime reconfiguration mechanism and can enable deployment adaptation of component-based distributed systems.*

To verify this thesis we present the design, implementation and evaluation of the adaptive deployment framework for component-based distributed applications following the listed research objectives:

1. To analyse exiting deployment and adaptive deployment approaches and determine their strong and weak sides.
2. To propose a comprehensive model of deployment that enables adaptive and runtime deployment of component-based distributed applications.
3. To determine and implement mechanisms required to realize deployment in the spatial dimension which is a selected subset of the functionality defined by the model.
4. To propose and implement a deployment planning algorithm that suits open, distributed and heterogeneous environments and is able to support runtime application reconfiguration.
5. To build a prototype of an adaptive deployment framework that combines the aforementioned elements.
6. To evaluate effectiveness of the created prototype in improving application performance.

1.4 Thesis Contributions

The research presented in this thesis has generated a number of original contributions that we summarize below:

1. The model of deployment for component-based applications that comprises spatial, temporal and semantic dimensions of deployment planning and includes dynamic aspects of deployment such as adaptation and updates. The model was based on the Platform Independent Model (PIM) defined in the D&C specification.

2. The design, implementation and evaluation of the adaptive deployment framework that enables runtime reconfiguration of distributed applications. The design of the framework follows the Autonomic Computing approach and, therefore, clearly separates between the layer of adaptation logic and the layer of managed resources. This, in turn, facilitates changes and further extensions.

3. The design, implementation and evaluation of the basic mechanisms enabling deployment adaptation in distributed environments such as the runtime component migration, communication interception, application monitoring and environment monitoring mechanisms. The design and implementation of the migration mechanism has been supported with a detailed discussion of key issues and the adopted approach.

4. The design and implementation of FDDP — an approximate deployment planning algorithm based on force-directed methods. The proposed algorithm is not the main contribution of this work, however, it is a novel and promising technique for runtime deployment planning.

1.5 Roadmap

The structure of the remainder of the thesis is organized as follows:

Chapter 2 presents background and related work in the area of application deployment, deployment automation and its adaptation. The purpose of this chapter is to provide research context for adaptive deployment, present important definitions, position our research in the broad area of software deployment and point out shortcomings of the relevant existing solutions. This chapter defines also *deployment planning dimensions* which we regard as important when distributed deployment is considered.

Chapter 3 presents overall concept of our Adaptive Deployment Framework and more closely discusses the key aspects that have had the most influence on its design. The chapter is split onto two parts. The first raises issues related to plain software deployment, whereas the second addresses adaptation in the deployment process. The main outcome of this chapter is formulation of ADF requirements, however, it also introduces the problem of deployment in virtualized distributed systems.

Chapter 4 presents design and overview of the framework implementation. The chapter starts with a brief introduction of the CCM model that was used to build deployable applications and the framework itself. Then overview of the framework is presented with clear distinction between the original D&C deployment infrastructure and our extensions related to deployment planning, reconfiguration and adaptation.

Chapter 5 focuses on basic building blocks that enable deployment adaptation. The main reason for this chapter is to analyse *the runtime component migration* mechanism. It discusses proposed solutions to the fundamental problems inherent to runtime software migration such as reaching quiescent state, the problem of residual dependencies and explicit context dependencies of a component. Further, a brief overview of our implementation of the container portable interception mechanism is presented. The chapter presents also an important issue of *instance identification* in relation to deployment, migration and interception mechanisms.

Chapter 6 evaluates all building blocks that are basis for the adaptive deployment framework. This chapter focuses on overheads and performance issues, however, for plain deployment infrastructure it also includes conformance to the D&C specification.

Chapter 7 discusses the FDDP algorithm and evaluates our deployment framework. The main purpose of this chapter is to show the presented *adaptive deployment framework in action*. A set of experiments investigate the capabilities of the framework to follow changes in the execution environment and to optimize overall application performance. This chapter also shows limitations of FDDP and outlines future research directions that could be taken to improve it.

Chapter 8 concludes the thesis, presents its limitations and suggests the potential areas for future work.

Chapter 2

Background and Related Work

Software deployment, in its most basic form, may be defined to be the process between the acquisition and execution of software. This process is performed by a *software deployer* who is the actor that acquires software, prepares it for execution, and possibly executes it [29]. However, Carzaniga et al. in [19] form the basis for broader understanding of software deployment. Their definition characterizes deployment as a process comprising not only activities related to acquisition, preparation and activation but also updating, and adaptation of software. To avoid ambiguity in meaning we term the former basic form of deployment — *plain deployment* or simply deployment, whereas the latter, extended form — *adaptive deployment*.

Despite that application execution is not a part of the deployment process in neither plain nor adaptive form, the latter definition — by including updates, adaptation, etc. — expands this process over the execution phase. The question is how adaptive application deployment differs from application management if they both are performed in application runtime. We consider adaptive deployment as means to automatize these aspects of application management that are related to component installation and instantiation. The key role is to relieve system administrators from many mundane tasks such as deployment planning, component configuration, system updates and reconfiguration. Therefore, we perceive adaptive deployment as a subset of all tasks related to application management.

In this work we focus on software deployment in component-based distributed systems, hence two important issues need to be considered: granularity of software components and granularity of an execution environment. Technologies such as CCM and Enterprise Java Beans (EJB) define components as fine-grained application building blocks that usually are much smaller than application modules or software packages. These technologies define also more fine-grained execution environment elements as deployment of

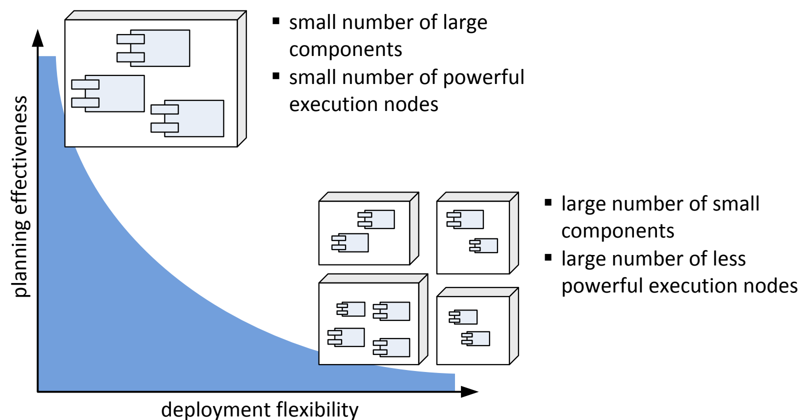

Figure 2.1: Deployment planning effectiveness in contrast with flexibility of component deployment.

components is performed not over an operating system but over a component/application server.[1] Moreover, for a single application there is often a need to run many component servers each having different configuration settings. Consequently, deployment of component-based systems is more flexible comparing to deployment of monolithic applications because a deployer has more freedom of how to distribute many small application components in often multi-node execution environment. Unfortunately, this higher flexibility is also the main factor of deployment complexity. Planning of component distribution heavily depends on the number of components to deploy and the number of execution nodes. It is much more effective for simple applications and environments and grows exponentially with increasing number of components. Figure 2.1 illustrates this relation graphically.

Another complexity barrier to deployment in open distributed systems is heterogeneity of execution environment elements. To deploy an application both components and execution nodes have to be described with a number of properties determining their requirements and resources respectively. Some of these properties often change due to various external and internal factors such as changing number of users or unpredictable node failures. This opens possibility to use adaptation and makes deployment in these environments a challenging task and an interesting research area.

Further, in this chapter we present a more detailed definition of the deployment process. Then, we discuss its automation starting with simple, single node installations and ending with Grids and systems based on the SOA paradigm. When deployment of more complex distributed systems is

[1]Actually components are installed in component containers that are included in a component/application server but this is usually an internal part of a deployment process not visible to a software deployer.

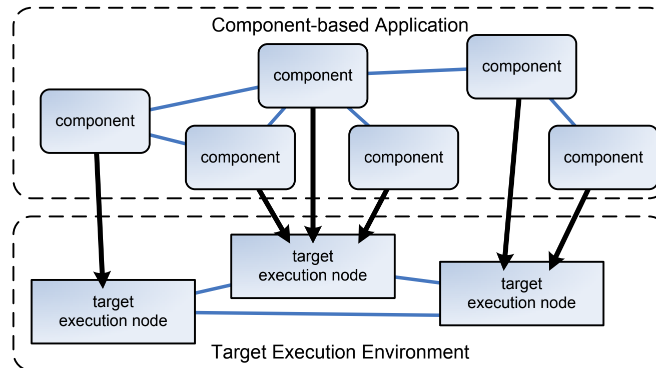

Figure 2.2: Installation of application components in a distributed execution environment.

considered we show, in addition, how it is related to virtualized execution environments. Later, we discuss planning of deployment which is the most complex phase of the whole process and hence requires more attention. Lastly, the background and work related to the main focus of this thesis — adaptive software deployment — is presented.

2.1 Definition of the Deployment Process

As mentioned earlier, plain deployment of an application includes three main actions: software retrieval, its installation and activation. When distributed systems are considered, however, from these three activities the most important is the installation which comprises of two steps. First, planning deals with assigning each application component to an appropriate execution node taking into account the component requirements and node's resources.[2] After deciding where a component will run, the second preparation step involves transferring component artifacts (such as executable, resource and software library files) to the nodes indicated in the plan. In other words, the installation process refers to matching the structure of a component-based application to the structure of a distributed execution environment. This is roughly illustrated in Fig. 2.2.

The plain application deployment is, however, too limited when considering real case component-based applications deployed over a distributed heterogeneous environments. Applications have to be updated and may be adapted and reconfigured, therefore, in this work we define a more complete *adaptive deployment* process:

[2]Planning assumes that a component is a unit of *independent deployment* what is in accordance with the general definition of component outlined by Szyperski in [119].

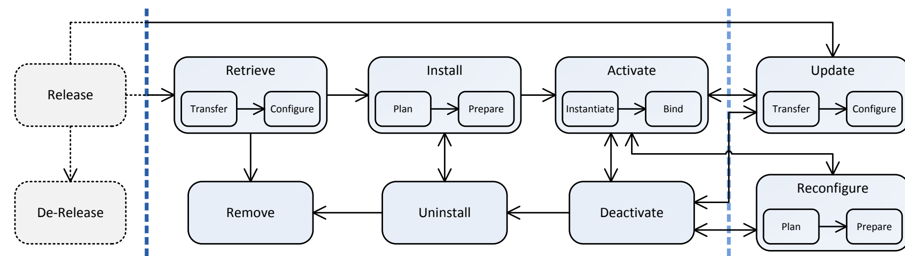

Figure 2.3: Activities in adaptive software deployment.

> Adaptive deployment is a continuous process performed at a software consumer site that starts after the software is published by a vendor and leads to application execution. When the application is running the adaptive deployment provides measures to perform application updates and reconfiguration.

Figure 2.3 presents a more detailed view of the adaptive deployment process in the form of an activity diagram. The two activities at the left side are not directly included in this process although they create a context for further activities. Software *Release* is performed by a software producer who prepares an application package comprising binary artifacts and a software description. The package is then published using common means like CD, web site, etc. During software maintenance phase the producer may occasionally release a new version of the software what evokes *Update* activities at the consumer site. The *De-release* step is done whenever the producer ceases further development and support of the published package. The other activities presented in the figure form the adaptive deployment process performed at the software consumer site. Following we present their more detailed description; we grouped together interrelated activities.

Retrieve/Remove — Retrieving is an act of transferring the published software package and bringing it into a component software repository at a consumer site. The location of this repository is not necessarily related to the place where the software will actually execute. The *Retrieve* activity includes also a configuration step which aims to make the software ready for installation. This step is specific to a particular consumer who, using the properties defined in the package, can tune up the software functionality to their needs. For example, configuration may include setting the font size in GUI components.
An inverse of retrieving is software removal that refers to deleting the software from the consumer repository. When the application was previously installed to ensure correctness the remove step assumes previous application uninstallation.

Install/Uninstall — Installation is an activity consisting of two steps discussed above. First is planning where in the distributed environment to run component instances taking into account component requirements and execution environment capabilities. This complex step demands a lot of computation and may have significant influence on application effectiveness. Further, preparation performs the actions to make the execution environment ready to run the software. For example, according to the provided plan it may do the copying of component artifacts into appropriate execution nodes. Separation between planning and preparation is deliberate because there are important cases when planning shall not have immediate effect on the target environment e.g. when running multiple instances of the same software.
Once the application has been installed it may be uninstalled. The *Uninstall* activity includes removing software from the nodes where component artifacts were copied.

Activate/Deactivate — The former brings the software to an execution state. It includes two steps: component instantiation and instance binding. Separation of these steps is crucial to seamless software activation esp. when circular dependencies between components exists. In heterogeneous environments the instantiation step may be an elaborate task because it usually depends on properties of the particular hosting node. The *Activate* action is the final task in the plain software deployment. In case of adaptive deployment, an activated application may further be updated and adapted until it is terminated.
The deactivation step refers to shutting down all running component instances.

Update — Enables evolution of the software. This is a special case of retrieval when existing components are exchanged with their newer versions provided by software producer. *Update* may require to deactivate the application, install a new version of some components, and further reactivate the software. Otherwise, it can be performed dynamically in runtime while a previous version is still active.

Reconfigure — Is similar to the *Update* activity as it involves modifying a software system that has been previously installed. It differs, however, from updating in that *Update* is initiated by remote events, such as a software producer releasing a new version of a component, whereas reconfiguration is initiated by internal events e.g. changes in the execution environment at the consumer site. Reconfiguration is similar to the installation step in that it often requires planning of deployment which takes into account new conditions at the consumer site. Planning, in turn, forces rerunning preparation.

Our definition is a combination of the definition presented by Carzaniga et al. in [19] with the one much more formally specified in the D&C specification [100]. The former identifies software deployment as consisting of activities such as: release, install, activate, update, and adapt. The latter focuses on plain deployment only but addresses distributed aspects of this process in relation to software component technologies. Additionally, it provides a lot of details about how to represent a component, an application, a target execution node, how to configure components, and how to perform deployment planning activity. The D&C specification is perhaps the most complete attempt to define a deployment and configuration standard [29], even though it defines merely static aspects of this process. Further in this work we supplement D&C and create a comprehensive model for adaptive deployment of distributed component-based systems. Before this, however, we need to discuss deployment automation that is crucial to enable adaptation of this process.

2.2 Deployment Automation

Automation of deployment is the first step in the way to achieve adaptive deployment. It is also beneficial, however, for improved correctness, speed and documentation of the application instantiation. Even having automatized merely the plain deployment process, we posses a documented receipt that we can follow to instantiate an application many times with only least effort required. Today's deployment tools provide varying levels of automation. Often, they automatize only selected activities of the process, for others requiring human intervention. In this section we present selected tools and approaches to automatic software deployment at different system scales starting from a deployment on a single computer machine.

2.2.1 Deployment Automation on a Single Machine

As long as the destination of deployment process is a single computer system its automation is usually well developed. This is mainly due to simplification of the process because several major obstacles such as system heterogeneity, distribution and deployment planning simply disappear. Moreover, deployment on a single machine does not always include application instantiation which, as a straightforward operation, is left to a user.

Numerous tools support or enable automatic deployment on a single system and we divided them onto three categories:

Package managers such as RPM Package Manager (RPM),[3] dpkg,[4] pkg*[5] are the most widely used low-level deployment tools for Linux and UNIX-like operating systems. These are command-line driven package management utilities capable of installing, uninstalling, verifying, querying, and updating software packages. These tools define a package as a discrete bundle of related files and associated documentation, configuration and meta information such as version, description, and signature. The heart of a package manager system is a database — a software repository containing all of the meta information of the installed packages. This database is used to keep track of the files that are changed and created when a package is installed what enables users to reverse the changes and remove the package later [49].

The managers operate on a package level rather than on a single-file or an entire application basis. The application is modelled as a graph of interdependent elements where dependencies are expressed by referring to a package name or its name and version. The lack of more sophisticated package referencing generates problems known as *dependency hell*.[6] Therefore, numerous higher-level tools for software package maintenance exist such as Yellowdog Updater Modified (YUM),[7] Advanced Packaging Tool (APT)[8] or Portage.[9] Their intention is to provide automated way to retrieve, install, update, and remove whole graphs of packages forming an application. OpenPKG[10] goes even further and resolves consistency issues between different UNIX-like operating systems [84].

Application installers provide a bit more sophisticated application model comparing to the package managers. Windows Installer,[11] InstallShield[12] and similar tools are organized around the concepts of components and features. A *feature* is a part of the application's total functionality that a user recognizes and may decide to install independently, whereas a *component* is a piece of the application or product to be installed which is usually hidden from the user. There is the 1-to-N relationship between a feature and component. When the user selects a feature for installation, the installer determines which components must be installed to provide that feature. Authors of installation packages need to decide how to divide their application into

[3] http://rpm5.org
[4] http://www.debian.org
[5] A set of tools: pkgadd, pkginfo, pkgrm. More info on http://docs.sun.com
[6] Dependency hell can take several forms: long chain dependencies, circular dependencies, conflicting dependencies, Internet access dependencies.
[7] http://linux.duke.edu/projects/yum
[8] http://www.debian.org
[9] http://gentoo-portage.com
[10] http://www.openpkg.org
[11] http://msdn.microsoft.com
[12] http://www.installshield.com

features and components. The selection of features is primarily determined by the application's functionality from the user's perspective, whereas mapping features on components highly depends on application design.

Similarly to package managers the core of application installers is the installation database tracking which applications require a particular component, which files comprise each component, where each file is installed in the system, and where component sources are located. Using application installers, the deployment process comprises two main phases: feature acquisition and installation execution. Additionally, if the installation is unsuccessful, a rollback phase may occur. At the beginning of the acquisition phase, an application or a user instructs the installer to install selected features or an application. The installer then progresses through the actions specified in the prepared installation database and generate a script that gives a step-by-step procedure for performing the installation. Then, during the execution phase, the installer passes the information to a process with elevated privileges and runs the script.

Web-centric Deployment Model is a step forward in deployment automation. It assists not only in configuration and preparation phases but also helps in software transfer. In this way web-centric model allows for fully automation of *Retrieve*, *Install* and *Update* deployment activities.

Several technologies support the web-centric deployment model: previously Java Applets and ActiveX components, and more recently Java Web Start [116] (a reference implementation of Java Network Launching Protocol (JNLP) standard [59]), .Net ClickOnce,[13] and ZeroInstall.[14] The main idea behind the web-centric deployment is in locating an application in a central repository, usually realized as a web server, from where it is acquired and transparently cached on a user computer. Unless trusted, the application is run in a protective environment — a *sandbox* — with restricted access to local resources. This model provides the following benefits over the application installers, package managers and package maintenance tools:

- simple deployment: the only action required is initial execution action which starts a deployment agent retrieving application resources and running the software,
- transparent updates: upon running an application the deployment agent checks the currently cached resources against the versions hosted in the repository and transparently downloads newer versions,

[13] `http://msdn.microsoft.com`
[14] `http://0install.net`

- incremental updates: only the resources that have been changed are downloaded. In case only few of the application's components have been modified, this can significantly reduce the update time,
- incremental retrieval: often there is no need to download the whole application at once. Some non-critical components, e.g. a documentation module, may be downloaded on demand until the first use. This again can reduce time of the initial installation and later updates,
- off-line support: the recent web-centric deployment technologies allow downloading all the application resources locally and run the software off-line without connection to the repository,
- runtime environment deployment: the technologies can also detect missing parts of a runtime environment (Java Runtime Environment (JRE) or Common Language Runtime (CLR)) and automatically install its required version. This further improves user experience of the deployment automation.

Similarly to the previous deployment techniques, the web-centric deployment model allows splitting software into smaller components. This enables incremental retrieve and update but is prone to the aforementioned dependency hell problem. Therefore, usually, the web-centric applications are packaged independently and instead of sharing their components they let duplicates be downloaded and cached separately.

As may be seen, the web-centric model ensures significant automation of the deployment process for single computer machine. The presented mechanisms find their use also for more complex execution environments. In such a case, however, to provide full deployment automation they need to be supported by additional deployment tools. In the following subsection we discuss techniques used to enable deployment over distributed systems.

2.2.2 Deployment Automation in Distributed System

Application deployment in a distributed system consisting of N computer machines is more than N-times as complex as on a single computer. More sophisticated execution environment introduces several additional issues to be resolved.

One of the main problems is heterogeneity of resources, which generates the need for requirement specification, resource and requirement description, and in consequence a deployment planning step. The other important issues are the need for coordination of deployment actions, support for

distributed sequencing, instantiation synchronization and dependency resolution. Moreover, as Gokhale et al. discussed in [56], configuration of application components pose much more problems in distributed systems than on a single machine. They noticed how tedious and error-prone is to manually ensure that all parameters exposed by software components are consistent throughout the whole distributed system.

All these issues clearly indicate that distributed deployment requires support by some kind of automation tool that should cover as much of deployment activities as possible. The three most relevant solutions for this are: *script-based*, *language-based* and *model-based* deployment [120].

Script-based deployment

To achieve distributed deployment the script-based approach makes as much use of the existing technology and tools as possible. The basis for this method is a set of scripts (e.g. written in `bash`[15]) that coordinate main deployment activities. Software distribution may be done using a secure copy tool (`scp`), package installation using presented earlier package managers, and configuration by applying predefined configuration files.

For small scale deployments this approach is easy to understand and convenient for system administrators who use all these tools on a daily basis. However, it is not suitable for more complex applications or more complex execution environments as scripts tend to become long and obscure. It has also limited expressiveness regarding to resource description what makes the automation not always achievable.

Language-based deployment

It overcomes some of the limitations of script-based approach by using a specialized configuration language, parsers and tools to perform deployment tasks. A number of frameworks exist that follow this approach such as SmartFrog,[16] Abacus [127] and GScript [85]. Although they are much different in details they share the same general idea — all deployment activities are described by a deployer in the form of a program that is executed by a dedicated interpreter.

Usually, language-based deployment frameworks consist of three elements: (1) a component model, (2) a language for describing configuration, dependencies and deployment workflow, (3) a distributed deployment and

[15] `http://www.gnu.org/software/bash`
[16] `http://smartfrog.org`

management runtime environment. The component model defines an abstraction layer for management of software being deployed. Each separate software element is represented in the model by a 'configurator' component that encapsulates its current state, configuration and provides a management interface.[17] With the provided language a deployer can make use of this interface to configure the software and create a workflow coordinating deployment tasks. A prepared deployment workflow is then executed by the distributed deployment engine that enacts the workflow to achieve and maintain the desired application state. GScript provides also the ability to express composition in space as well as composition in time. The former involves direct connections between components to allow control and data to pass directly between them. The latter assumes that components do not have to be directly connected. Instead, their interfaces can be invoked by the deployment coordinator.

Several important benefits stem from using the language-based deployment approach. Mainly, it allows deployment engine to continue application management while the system is running. This enables more sophisticated management strategies like software reconfiguration, automation of updates and on-demand deployment. However, language-based deployment modelling does not allow for full deployment automation as it requires a user to be involved in preparation of a deployment workflow at some stage. This deployment workflow depends on the specific application composition and execution environment and can hardly be automatically generated. With language-based approach it is also difficult to address heterogeneity of resources and components. In order to address these issues the model-based techniques may be used.

Model-based deployment

Currently, this is the most advanced approach to deployment. It uses an Architecture Description Language to model structure of a software application and structure of an execution environment. An ADL explicitly represents components, connectors, component configurations and their requirements on one side, and execution nodes, network connections and resources on the other. This clear separation between software and environment models is one of the key advantages of the model-based approach. The environment-side components can in a declarative manner expose their resources, whereas software-side components can declare their requirements. This improves reusability and enables full automation of the process. The model of a software can be reused when deploying the software in different execution

[17] For example, the most sophisticated of these tools SmartFrog distinguishes 5 component states: installed, initiated, started, terminated, and failed.

environments. Similarly, the model of an execution environment may be reused for deployment of many different applications. Moreover, when COA-based systems are considered, their architecture model can naturally be a basis for a definition of the software deployment model. This makes the model-based approach especially suitable for the component-based systems.

There exist many frameworks that follow model-based deployment. MOCCAcino [85], Deployment And Configuration Engine (DAnCE) [31], DeployWare [47] are only a few examples. MOCCAcino facilitates deployment and management of computationally-intensive applications on grids and is specifically suited for Common Component Architecture (CCA). DAnCE addresses deployment of CCM applications. Similarly to our solution, it is based on the D&C specification that standardizes many aspects of configuration and deployment for component-based systems. DAnCE, however, enhances the D&C data models to describe deployment concerns related to real-time QoS requirements and middleware service configuration and deployment [115]. DeployWare is based on the Fractal component model[18] and abstracts concepts of the deployment independently of the underlying paradigm and technology. It provides a Domain-Specific Modeling (DSM) language and a metamodel to mask software heterogeneity. Everything in DeployWare is being modelled as a component: properties are represented as a composite component that contains the configurable properties of a software, dependencies are composites that contain references to other software components, even procedures, such as install, configure or start, are represented as components symbolizing the instructions. These instructions are runnable components that use the DeployWare libraries to realize elementary deployment tasks.

Most of the existing frameworks propose a proprietary solutions to model-based deployment that address different, often specific, needs. There are, however, some efforts made to create a standard for the model-based approach to deployment in distributed systems. Two particularly notable are: the Common Information Model (CIM) standard defined by Distributed Management Task Force (DMTF) in [35] and the Deployment and Configuration of Component-based Applications specification produced by OMG [100].

Common Information Model together with closely related Web-Based Enterprise Management (WBEM)[19] are standards developed by a consortium of major hardware and software vendors called the DMTF. CIM provides the framework by which a system can be managed using common building blocks rather than proprietary software. The standard comprises a meta-schema and a number of management schemas that are the building blocks for manage-

[18] http://fractal.ow2.org

[19] http://www.dmtf.org/standards/wbem

ment platforms and management applications, such as device configuration, performance management, and change management. WBEM, in addition, aims at unifying the management of enterprise computing environments using a set of standard Internet technologies like HTTP, XML and DTD [33].

Software deployment is defined in the application schema. It intends to describe applications with structures ranging from standalone desktop applications to a sophisticated, multi-platform distributed, Internet-based systems. The schema incorporates three major concepts: (1) structure of an application, (2) life cycle of an application, and (3) the transition between states in the application life cycle. The structure of an application consists of the four following components:

Software Element is a collection of one or more files and associated details that are individually deployed and managed on a particular platform. It represents the fundamental building block of the CIM application management information model.

Software Feature is a collection of software elements that performs a particular function or role in the more general Software Product. This level of granularity is intended to be meaningful to a consumer or user of the application. This concept allows software products or application systems to be decomposed into units that have a meaning to users rather than units that reflect how the product or application was built (i.e., Software Elements).

Software Product is a collection of software features that can be acquired as a unit. Acquisition implies an agreement between the consumer and supplier, which may have implications in terms of licensing, support, or warranty.

Application System is a collection of software features that can be managed as an independent unit that supports a particular business function.

Basing on these elements, the CIM model allows managing application life cycle using four activities: (1) deployment, (2) installation and configuration, (3) startup, (4) operation including monitoring. These activities have direct relation to the software elements' states that are captured by the application model (Fig. 2.4).

The deployable state describes the element in its distributable form (for example, in a software repository), as well as the details and operations required to move the element to the next, installable state. The installable state describes the element as ready for installation (e.g. as a zip file that can be decompressed and installed). Also, the details and operations required to

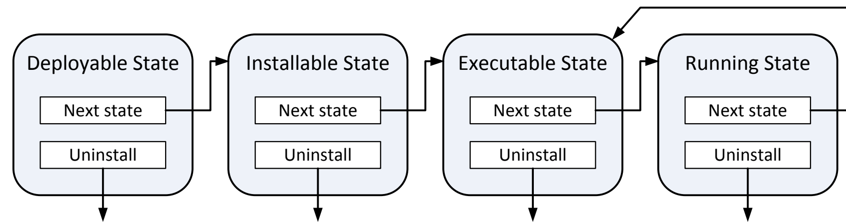

Figure 2.4: The life cycle of a CIM Software Element.

move the element to the executable state or back to the deployable state can be defined. The executable state describes the element as ready to start/run, as well as the details and operations required to move the element to the running state (i.e. the next state) or back to the installable state. Finally, the running state describes the element as it is configured and running.

The second version of the CIM Application Management Model is not expected to capture all the information required to deploy application in a distributed environment. However, the model provides a base upon which additional modelling concepts can be added [34]. The two most important aspects which we regard as missing are: the lack of support for distributed applications and a low abstraction level for software elements. The Application Management Model does not address the problem of deployment of applications build from distributed components that are intended to work over a distributed execution environment. There are no means to describe dependencies between remote components like a connection interface, protocol used in communication or requirements against network links. The elements that are building blocks of an application in CIM are black-box components. It means that relations between them are expressed by the Software Element ↤◇ Software Feature ↤◇ Software Product aggregations only. Therefore, to incorporate aspects of application distribution into the CIM deployment model significant extensions would be required.

Deployment and Configuration of Component-based Applications is defined by OMG and specifies a Platform Independent Model which follows the general idea that deployment process remains the same independently of the underlying technology of software realization. The PIM can further be customized with Platform Specific Models (PSMs) (such as PSM for CCM [103, Chap. 14]) to address deployment aspects specific to a particular software technology. D&C segments the PIM in two dimensions: (1) data vs management and (2) component software vs target vs execution. This segmentation gives six complementary views on the process of deployment and configuration, thereby creating a rich framework for deployment of distributed applications.

The models defined in D&C are supplemented with the definition of abstract actors that manipulate the data, are clients to the interfaces, and enact the various phases of deployment. Usually, part of the actor will be implemented in software tools, aiding a user in development and deployment of an application. There are three categories for actors: development, target, and deployment, mirroring one of the segmentation dimensions. Actors in the first category are concerned with the various phases of implementing a component, starting with an interface design and eventually creating a component package. The basic idea of component-based development defined in D&C is to divide an application into small reusable components that can be connected to other components via ports. A set of interconnected components creates an assembly that can be seen as a component in itself, and therefore can be used recursively. The only actor in the target category — Domain Administrator — describes the local target environment and all its resources. The target environment is composed of nodes, interconnects and bridges. Nodes have computational capabilities and are a target for executing component implementations. Interconnects provide a direct shared connection between nodes (e.g. representing an Ethernet cable). Bridges route between interconnects, modelling both routers and switches. Actors in the deployment category take existing component packages, and deploy them into the target environment in order to create running applications.

Although the D&C specification creates a very good basis for automation of software deployment in distributed systems, it leaves some important aspects undefined. The main issues are the lack of languages for specifying component's selection and resource requirements as well as inadequate specification of target environment management and monitoring. Moreover, D&C does not address dynamic aspects of application deployment, which are crucial to adaptive deployment. The selection requirements express users' needs that are meant to drive the selection among alternative component implementations. Each of these implementations offers some capabilities and planning requires agreement of the vocabulary on both sides. Despite that no such vocabulary is specified, some efforts are made to define it with respect to QoS.[20] The resource requirements express needs of a component implementation against resources of an execution node and are a communication channel between development tools and the management layer of a target environment. Without a standard definition of the resource requirements automation of deployment may be limited because of possible mismatch between component requirements and environment resources description.

Considering monitoring and management of a target execution environment, the D&C specification proposes only basic support expressed in the

[20]Work of G. Deng [31] and more recently "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms" specification [104] address this issue.

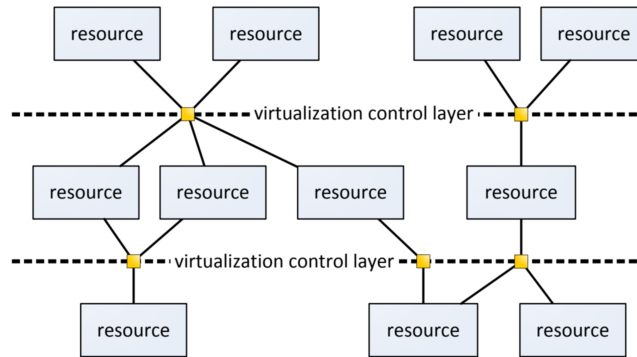

Figure 2.5: The multilevel virtualization model.

form of the `TargetManager` interface. This key area of software deployment also needs extensions.

2.2.3 Deployment Automation in Virtualized Environments

Modern computer systems are now sufficiently powerful to present users with the illusion that one physical machine consists of multiple virtual machines each running a separate, and possibly different, instance of an operating system. However, virtualization can apply not only to the OS layer but to a range of different system layers including hardware-level device virtualization, operating system-level virtualization, and high-level language virtual machines [16].

The fundamental idea behind virtualization is to introduce an additional layer of indirection in accessing resources so that a lower-level resource can be transparently mapped to one or more higher-level resources or, to the contrary, many low-level resources can be mapped to one high-level resource. This concept is illustrated in Fig. 2.5 which shows, additionally, that each virtualization level has its own control layer responsible for management and enforcement of mapping between two adjacent levels. The mapping may be expressed using the three following basic concepts [123]:

partitioning — the ability to run multiple systems on a single physical system in order to share more effectively underlying hardware resources (e.g. OS virtualization with VMware [123] or Xen [129]),

consolidation — opposite to the partitioning, focuses on reducing number of visible and accessible resources simplifying the view of a system at the higher level (e.g. Unix-like disk management[21]),

[21]Using `mount` command in Unix-like systems many physical disk drives can be combined

containment — leads to unification of the infrastructure by covering different lower-level resources with a unified virtual abstraction layer (e.g. Java or .NET Virtual Machine).

Virtualization techniques are widely used in modern computer systems increasing their manageability and flexibility. The choice of virtualization level decides about the availability of management and monitoring mechanisms. It also has direct influence on how resources are described, what their properties are, and in consequence what software requirements may be. Actually, the choice of virtualization level have impact on every phase of design, development and management of software applications including software deployment.

The main concept behind existing deployment solutions that take virtualization into account (such as N1 Service Provisioning System (N1 SPS) [117], InstallAnywhere[22] or Installable Unit Deployment Descriptor (IUDD) [124] and Solution Deployment Descriptor (SDD) specifications [92]) is the ability to represent the *hosted–hosting* relationship between resources. A host represents an execution environment for a hosted resource like, for example, a web application server is an execution environment for a web application. This kind of aggregation can be nested arbitrarily depending on number of virtualization layers in a target execution environment.

The important benefit of deployment in virtualized environments is that it can encompass deployment on all desired layers. Therefore, a typical use case would be to deploy an operating system on a physical machine, then deploy application server on this operating system, and finally install and execute a user application. Automation of this process can greatly simplify administration and maintenance of complex multilayer execution environments. However, to the best of our knowledge, none of the existing platforms or standards supporting deployment in virtualized environments allows for fully automatized model-based deployment in distributed systems. Either they support distributed systems with language-based deployment or focus on model-based approach dedicated to a single machine. Integration of these two approaches, although increasing complexity of deployment, would allow for creating a comprehensive solution for automatic distributed deployment. Furthermore, it would enable investigating adaptive deployment in virtualized environments which is an interesting research area.

to create one file system.

[22] `http://www.acresso.com`.

2.2.4 Deployment Automation in Grids

Despite Grids are distributed and virtualized environments, we distinguish them here due to an important additional aspects they introduce to the problem of software deployment. To the complexity of deployment caused by system virtualization, heterogeneity and variability of resources, Grid-based systems add heterogeneity of organizations. Very often these organizations differ in management policies, rules and regulations under which they are functioning. In result, their collaboration involves such technical challenges as cross-organization authentication, authorization, resource access and resource discovery [48].

Automation of deployment in Grids refers to distributed job scheduling and is now widely available, e.g. in Condor,[23] Globus,[24] Sun Grid Engine (SGE)[25] and ProActive.[26] A job is a segment of work. Each job includes a description of what to do and a set of property definitions that describe how the job should be run. Job deployment in these environments follows the model-based approach: each job is accompanied by a resource requirement descriptor that is analysed by a scheduler after the job has been submitted. Provided with this information the scheduler decides where to send the job for execution.

Usually job submission systems operate on the process level. It means that a job is often a program running directly in the OS. In more complex cases, a job may be a set of program instances, such as *Array Jobs* in SGE, that run independently on different parts of data. It may also be a set of interrelated program instances such as a parallel application or a dependency tree application. The former is composed of cooperating program instances that must all be executed at the same time, often with requirements about how the programs are distributed across the resources. The latter uses a Direct Acyclic Graph (DAG) to model dependencies between tasks. Vertices in the graph represent computation, whereas edges identify dependencies. Often communication interface between jobs in Grid is limited to file exchange or, in case of parallel jobs used in large scale scientific computation, to the specialized Message Passing Interface (MPI) interface. In this work we address higher abstraction level offered by a component-based middleware. We find this a richer and more expressive approach when designing and developing distributed applications.

ProActive is more sophisticated in this respect and allows not only scheduling of stand-alone executables but also *proactive application*s which can

[23] `http://www.cs.wisc.edu/condor`
[24] `http://www.globus.org/toolkit`
[25] `http://www.sun.com/software/sge`
[26] `http://proactive.inria.fr`

model components [94]. Therefore, similarly to our work, ProActive can support application design according to COA. It is, however, restricted to Java language only, whereas in this work we look for a solution which is language independent. ProActive uses also a different and interesting approach to job deployment. Jobs are bound with the execution environment through the `VirtualNode` element, which is abstraction for a location of resources. A single `VirtualNode` can be mapped onto one or several Java Virtual Machine (JVM) processes which can be created on or acquired from a physical node [18]. Nonetheless, the mapping is done manually and the platform does not address the problem of deployment planning.

More recently, there has been a move towards utilising Web Services (WS) to build Grid and other distributed applications. A WS-based application is represented as a set of services that communicate through the exchange of messages using widely accepted standards for describing service interfaces (WSDL) and transferring those messages (SOAP) [128]. However, this kind of applications is much more related to the SOA paradigm and, therefore, we discuss it separately in the following section.

2.2.5 Deployment Automation in SOA

The main idea behind Service-Oriented Architecture is to orchestrate existing loosely coupled services to perform a desired function or a business process. Usually, the orchestration involves enacting an application workflow that delegates subsequent tasks to appropriate services.

Considering deployment of a SOA-based application four actions need to be discussed: *discovery*, *selection*, *service deployment* and *binding* (Fig. 2.6). Discovery involves searching for an appropriate service instances that comprise an application workflow. SOA is based on the premise that most of the applications can be built using existing services and for most of the service types there exists a set of competitive instances available. Therefore, the next step refers to comparing the available service instances and selecting the most appropriate for the application being deployed. Proper selection requires ability to describe service capabilities and application requirements. Occasionally, however, none of the existing services can meet application requirements and then the desired service needs to be deployed on-demand. Even if it is supposed to be less frequent, it might be required for more specialized applications and services. Still, deployment of a service itself does not differ much from the approaches discussed yet. Depending on its design it may involve simple single machine deployment or more complex distributed or Grid deployment. Eventually, when all required services are found and accessible, deployment of the SOA-based application turns to the binding step. This aims at enabling communication between services and

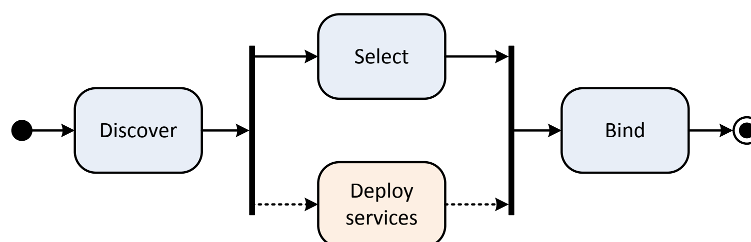

Figure 2.6: Four steps of deployment of SOA applications. The dashed line denotes the less frequent flow is SOA deployment. The highlighted block represents the area of this work.

involves all the problems related to software adapters, message exchange and transformation as well as coherency of service semantics.

Automation of deployment in SOA requires automation of all consecutive steps and, therefore, is more complex than all previously discussed problems. Currently, this is an area of active research and existing solutions address this process partially rather than providing a thorough automatic solution to deployment of SOA-based applications. Chukmol proposes in [22] a framework for WS discovery. Project Dynasoar[27] provides a generic infrastructure for the dynamic on-demand deployment of Web Services. Lécué et al. discuss in [77] issues related to composition of stateful services. It is the matter of future research if a complete answer to the problem of deployment of SOA-based applications can be found. In this work, however, we focused on deployment of a single service. The assumption we made is that the service is a component-based and distributed application.

2.3 Deployment Planning

The solutions discussed already automatize the deployment process to high extent, yet for adaptive deployment to be effective full automation is required. It means that all deployment activities, starting from *Retrieve* and ending with *Update* and *Adapt*, need to be automatized. Even if most of them are straightforward and technical tasks, in case of distributed and heterogeneous systems deployment planning becomes a significant barrier to overcome. As mentioned earlier, to ensure full automation of deployment planning a model-based approach should be considered. Then the planning concerns matching software components against environment resources taking into account component requirements and resource availability. This ostensible simplicity is, however, not always easy to attain and this section presents

[27] `http://www.neresc.ac.uk/projects/dynasoar`

more detailed discussion about the deployment planning problem itself and the sources of its complexity.

2.3.1 Definition of Deployment Planning

Planning of component deployment should not be confused with the classic planning problem that refers to searching for a sequence of actions which shall be carried out to get a system from an initial state to a desired goal state [75, Sect. 2].[28] Considering model-based deployment, we rather follow the D&C specification that defines deployment planning as:

> an activity that takes the requirements of the software to be deployed, along with the resources of the target environment on which the software will be executed, and decides which implementation and how and where the software will be run in that environment.

Therefore, in deployment planning what is interesting is not the sequence of actions to find a viable component placement but rather the goal state itself. This goal state, which is a proper *deployment plan*, is such a placement of components in the environment that satisfies all the requirements of the components and their interconnections not exceeding resource availability of the execution nodes and network links.

Deployment planning should also be distinguished from Component Placement Problem (CPP) [66] and its more general form Application Configuration Problem (ACP)[29] [67]. The main goal of these NP-hard problems is in constructing a deployable application given specifications of environment, components, and user goals. For example, having four components: `MailClient`, `MailServer`, `Encryptor` and `Decryptor`, the ACP and CPP would consider two possible mailing applications. One consisting of only `MailClient`, and `MailServer` and the other comprising all four components. Then if a user goal is to protect message privacy, an ACP/CPP problem solver should decide which one to use. In case the environment has security mechanisms embedded, the valid solution would be the two-component application as

[28]An example of the classic planning problem would be to find a sequence of moves of a mobile robot to change its position from location A to location B. This concept of planning is also used in N1 Service Provisioning System to describe deployment plans. In N1 SPS a plan: *is a sequence of instructions that is used to manage one or more components on the specified hosts. For example, a plan might install three components and initiate the startup control of another component* [118]. It is unlikely, however, that a deployment plan in N1 SPS is created with no human intervention.

[29]The CPP is a restricted variety of the ACP problem in that its goal is to find a configuration in the absence of the time aspect.

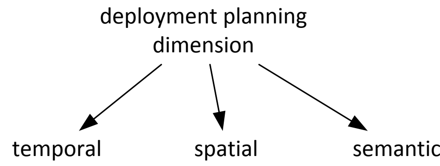

Figure 2.7: Three dimensions of software deployment planning.

`Encryptor` and `Decryptor` components are redundant and decrease application performance. Otherwise, if there exist some insecure links and nodes between client and server components, the valid solution would be the latter application that includes an encryption mechanism. In the deployment planning problem, however, what we consider is a software package with a *precomposed* application. It means that the set of application components and their interconnections are already defined in the package and hence the deployment planning problem is easier to solve than the CPP or ACP which in general are undecidable [67].

2.3.2 Planning Dimensions

In the most common sense, when component deployment is considered, only the spatial dimension is taken into account. This means that deployment planning can only decide *where* in the target environment software components will be located and executed. However, it is possible to provide a broader view on this process and then decompose it on a number of independent dimensions. As shown in Fig. 2.7, we distinguish three dimensions of deployment planning: *spatial*, *temporal* and *semantic*. To the best of our knowledge the literature does not define these dimensions explicitly, therefore, following we provide their definitions together with references to other similar approaches.

Irrespective of the deployment dimension the basis for model-based deployment planning is the structure of an application to be deployed. It may express relations between components, their dependencies, requirements, collocation constraints and requirements of component links. The other inputs to the planning process depend on a selected dimension.

The temporal planning dimension. Historically, when computer systems were dominated by mainframes, time sharing was the main approach to deploy and execute applications. The scarcity of resources posed the problem of running multiple tasks on a single machine concurrently and the answer to this were CPU scheduling algorithms [45, 113].

Planning in the *temporal dimension* is closely related to local scheduling. As an input it takes the structure of an application, a resource pool that determines limitations of the execution environment and a graph of temporal dependencies between application components. Usually, such a graph is defined in the form of a DAG that specifies a precedence order in which components must be deployed and undeployed. Temporal deployment planning differs from scheduling merely in that it considers components of one application rather then unrelated tasks executed in an OS.

The inputs to the basic temporal planning we define as follows:

A — a set of component instances forming an application,
$D = \langle A, O \rangle$ — a DAG defining precedence order of deployment actions; A is the set of component instances — vertices in the DAG; O is a set of arcs that determine the precedence order,
R — a resource pool that application components can demand,

and the result of the temporal deployment planning is plan $P_{temporal}$ that we define as:

$$P_{temporal} = \{(i, T) \; : \; \forall_{i \in A} \exists_{T \in \mathcal{T}} \text{ such that instance } i \text{ is running in time range } T\}$$

while the following constraints are satisfied:

$$\begin{aligned} &\forall_{T \in \mathcal{T}} \forall_{r \in R} \sum_{i \in P(T)} |r_i| < |r| \quad \wedge \\ &\forall_{(i,T) \in P_{temporal}} \forall_{j \in N_D^{-d}[i]} \sup(T_j) < \inf(T) \quad \wedge \\ &\forall_{(i,T) \in P_{temporal}} \forall_{j \in N_D^{+d}[i]} \sup(T) < \inf(T_j) \end{aligned}$$

where:

$\mathcal{T}$ — a set of identified time ranges determining when component instances are running,
T_i — the time range when instance i is running,
$P(T)$ — a set of all instances planned to be executed in time range T,
$|r|$ — amount of resource r available in a resource pool,
$|r_i|$ — amount of resource r required by the instance i,
$N_D^{+/-d}[i]$ — closed dth out-/in- neighbourhood of an instance i, where d is the diameter of D.[30] It denotes all successors/predecessors of i.

[30]Notation according to Bang-Jensen and Gutin [8].

Usually, during deployment planning in the temporal dimension the optimization objective is to minimize the time span required to execute the application:

$$\min(\max_{T \in \mathcal{T}}(\sup(T)) - \min_{T \in \mathcal{T}}(\inf(T)))$$

The spatial planning dimension. With the appearance of workstations and personal computers more and more resources were available and then spatial dimension played the main role. In that time the problem was not *when* but *where* to run software components to achieve the best results. This included all the problems of component distribution, heterogeneity of resources and changes in their availability.

The *spatial dimension* refers to the physical location of components in the execution environment. The inputs to the basic spatial planning are:

A — a set of component instances forming an application,
L — a set of links between the application components,
N — a set of execution nodes forming an execution environment,
C — a set of interconnections forming a network connecting the nodes,

and the result of the spatial deployment planning is plan $P_{spatial}$ that we define as:

$$P_{spatial} = \langle P_A, P_L \rangle$$
$$P_A = \{(i, n) \; : \; \forall_{i \in A} \exists_{n \in N} \text{ such that instance } i \text{ is running on node } n\}$$
$$P_L = \{(l, c) \; : \; \forall_{l \in L} \exists_{c \in C} \text{ such that link } l \text{ uses interconnect } c\}$$

while the following constraints are satisfied:

$$\forall_{n \in N} \forall_{r \in R_n} \sum_{i \in P_A(n)} |r(i)| < |r| \quad \wedge \quad \forall_{c \in C} \forall_{r \in R_c} \sum_{l \in P_L(c)} |r(l)| < |r|$$

where:

R_x — a resource pool of a node or interconnect,
$|r|$ — total amount of resource r available in a resource pool,
$|r(\cdot)|$ — amount of resource r required by a component instance or link,
$P_A(n)$ — a set of all instances planned to execute on node n,
$P_L(c)$ — a set of all links planned to use interconnection c.

Planning in the spatial dimension has no single optimization constraint and usually there exists a set of QoS metrics that are optimized during planning. Moreover, the presented definition describes only the basic form of the problem and more elaborate versions are possible e.g. such that define locality constraints for component instances to separate them on

The semantic planning dimension. Nowadays, with the abundance of computing resources we often meet a situation when for a particular component/service type there is already a set of semantically equivalent components/services running and available. Therefore, the problem we face is not *when* and *where* a component should be executed but rather *which* one to choose to obtain the best results.

When considering inputs in the semantic deployment, we focus on QoS metrics provided by services rather than on exact node or network resources availability. It is hardly possible to investigate service provider's network and analyse resource availability of their execution environment. Therefore, inputs to this problem we define as:

A — a set of component instances forming an application,
S_i — a set of services that are available and semantically equivalent to component instance i,
Q — a set of QoS metrics to be satisfied,

and the result of the semantic deployment planning is plan $P_{semantic}$ that we define as:

$$P_{semantic} = \{(i, s) : \forall_{i \in A} \exists_{s \in S_i} \text{ instance } i \text{ is realized by service } s\}$$

while the following constraint is satisfied:

$$\forall_{(i,s) \in P_{semantic}} \forall_{q \in Q} \; s \text{ satisfies } q_i$$

where:

$q(i)$ — is QoS metric required to be satisfied by instance i.

Semantic planning may be used for describing deployment of applications designed according to the SOA paradigm. SOA assumes that an application is orchestrated using a set of existing services of which some realize the same contract. In this case deployment planning problem is to select which service instances will be used in the actual processing.[31]

Combining dimensions. The presented dimensions can be considered independently of each other or can be combined together creating more elaborated deployment scenarios. For example, the spatio-temporal dimension can address the problem of scheduling of interrelated jobs in grids [113]. Combining the spatial and semantic dimensions one can model the problem

[31] It is worth noting that in SOA the deployment preparation phase may only be limited to interconnecting existing service instances.

of full SOA deployment, which includes both service selection and service deployment. Ultimately, when all three dimensions are considered together it is possible to model very sophisticated deployment scenarios with runtime services' selection and undeployment.

The only aspect which we regard missing in the multidimensional deployment planning is context awareness. Contextual deployment refers to the problem of software deployment in reaction to changes in its execution environment and binds deployment actions with the state of selected context variables. The need for context-aware deployment stems from two important facts: ubiquity of computing devices and interests in autonomic computing. In both these cases changes in execution context are drivers influencing application behaviour, and deployment is one way to express this influence. For example, if we would like to use all mobile nodes in a network to run a single application such as text communicator, context-aware deployment can easily support this need. Whenever a mobile node is detected in the network the deployment infrastructure can initiate deployment of application components to this node. Similarly, when a node leaves the network the infrastructure located on the node can undeploy all unneeded components. Awareness to context is one of the prerequisites to adaptive deployment which needs to observe application's execution environment to perform suitable adaptation actions.

However, context-aware deployment evades the model-based planning approach mainly due to complexity of context representation and processing. More natural to address this aspect are language-based techniques such as presented by Dubus and Merle in [38], which uses an Event Condition Action (ECA) language to react on context changes. Similarly, our deployment framework uses combination of the model-based and language-based deployment to address the complexity of adaptive application deployment.

2.3.3 Complexity of Deployment Planning

In this work we focus mainly on spatial dimension of application deployment. However, the definition of the problem presented above is only its basic form and, therefore, to analyse complexity of this problem we first present the D&C models. They allow for more complete definition of application and execution environment. The D&C data models create a very expressive framework for describing a software package and target environment as needed for planning in the spatial dimension. On the one hand in the software package description it is possible to include (Fig. 2.8): *selection requirements* to identify a component implementation with desired *component capabilities*, *connection requirements* to express required qualities of a network connection, *locality constraints* used for requesting collocation or separation

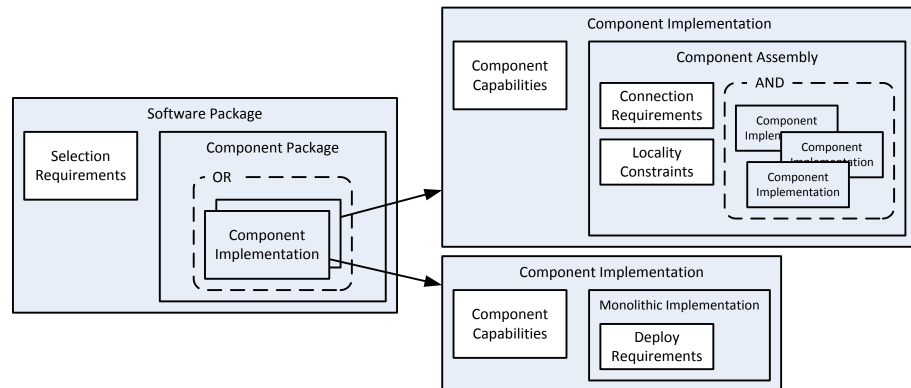

Figure 2.8: A coarse illustration of the software package structure defined in D&C.

of component instances and *deploy requirements* matched against target node resources. On the other hand to describe an execution environment the D&C Target Data Model allows specifying node and network resource capabilities as a set of (`type`, `name`, `value`) tuples. The only missing link in the D&C is a description of user goals which is left as an implementation specific feature. Nevertheless, making use of all the flexibility provided by this framework allows creating very accurate software and site models. The problem is, however, how to effectively plan component deployment given such amount of information and constraints.

Even when most of the constraints in a description are omitted (namely selection requirements, connection requirements, locality constraints and component capabilities) planning of component deployment may be represented as the variable-sized Bin Packing Problem (BPP).[32] In this variant of the packing problem, items must be packed into bins of various sizes and the goal is to minimize the total volume of the used bins. Application components with different requirements are represented by items, and heterogeneous execution nodes with different resources are represented by bins of various sizes.[33]

The basic BPP and wide variety of its variants has been extensively studied since the 60's. The problem is known to be NP-hard [23] but there exist approximation algorithms solving it in polynomial time. Nevertheless, real component deployment problem is much more complex to solve. Adding only locality constraints to the software package description results in the *variable-*

[32]Assuming that the objective of the planning is to minimize the number of execution nodes which will run application components, the deployment problem is a variant of bin packing problem. When the goal is to minimize load of each execution nodes the problem becomes a variant of multiprocessor scheduling problem [79].

[33]It is worth noting that even if components require and nodes offer multiple different resources, the deployment problem does not become the multi-dimensional BPP.

sized bin packing with conflicts problem which is much more difficult to tackle as it generalizes both the variable size bin-packing and graph colouring problems [42].

Complexity of deployment planning comes from the large search space that, in case of spatial dimension, depends exponentially on the number of application components and number of nodes. If n is a number of execution nodes and m is a number of components to deploy, the search space has the size equal to n^m which is the number of permutations with repetition. When planning we need to select m nodes which will host the components from the total set of n and each node may be chosen more than once. Due to the size of the search space solving the deployment planning problem for real size inputs can be realized by approximation algorithms only. However, all the efforts to find an effective algorithms based on BPP and its variants are inadequate. Deployment planning cannot be limited to processing only static software package and execution environment descriptors. The main reason for this is dynamic nature of both application and execution environment.

Planning in Open Distributed Systems

Open distributed systems are highly unpredictable in nature. Unlike closed platforms (e.g. embedded systems or simple mobile phone OSs) that run a predefined set of applications and workloads, open systems allow users more freedom in this respect. Dynamism of the open systems stems from the following key facts:

- resource needs of applications may change depending on their execution phase, user interaction, current workload, etc.,
- availability of resources provided by execution nodes may change due to node failures as well as application and/or server consolidation; usually a node runs more than a single application or operating system concurrently,
- availability of resources provided by network connections may change depending on link failures, applications workload, changing communication patterns, sharing the network between different applications, and many others.

Since all these factors are changing during application runtime, static description of requirements and resources cannot be sufficient. To deal with this issue during planning we divided node capabilities and component requirements into static and dynamic:

static resource — characterize properties of a node which are independent of the number and kind of components executed on this node. An example of a static resource may be the type of CPU architecture or a shared library installed on the node,

static requirement — define constant and essential component needs. If a static requirement of a component cannot be satisfied by the execution environment, the component cannot be instantiated. A kind of static requirements are the type of CPU architecture or the minimum size of memory required to run a component,

dynamic resource — are allocated to the installed components and may depend on the number of components running and their allocation needs. Dynamic resources are further divided into shareable (e.g. memory, CPU, hard disk space) and non-shareable (e.g. an externally connected device such as a scanner),

dynamic requirement — characterize dynamic needs of a component and often depend on external factors hardly predictable in advance. An example of dynamic requirement may be size of memory needed by a component when running.

Using a description language to characterize static environment resources and static software requirements is straightforward. If they are identified, it is enough to include them in a node or component description. Conversely, describing dynamic resource and requirements elicits many important questions such as:

- how to express component requirements if they depend on a number of concurrently processed requests?
- what if they also depend on request size?
- what if they depend on availability of some other resources?

There exist attempts to describe resource usage as a function of some input variables (e.g. [63, 66]). Although they could enable more accurate resource description, their main flaw is in finding functions that would appropriately describe relevant dependencies. For more complex use cases it seems hardly possible because searching for these kind of dependencies requires extensive and expensive tests of each particular software component. Such tests are fully reasonable and often required for real-time embedded systems when the execution environment is more predictable and user must be assured the application will not fail unexpectedly. Generally, however, if an application is assembled from a variety of components provided by different vendors and is

supposed to run in open environments, another approach is needed. It is not possible, or at least not reasonable, to determine what are resource needs of a component for every possible combination of input values and then conduct these tests for each application component separately and together forming an application.

Similarly, major obstacles are encountered when describing dynamic resources of the execution environment. Both, computer machines and computer networks are typically shared between more than one application[34] and usually lack mechanisms guaranteeing the strict timeliness and quality requirements. In this *multiple application deployment* problem it is not easy to predict resource availability and ensure that a newly deployed application will not encounter any resource shortage.

Planning of deployment in open distributed systems requires, therefore, a different method to overcome the aforementioned problems. A promising approach is to introduce dynamic deployment that could adapt to changes in application needs and environment capabilities.

2.4 Adaptive Deployment

As presented in the previous section, deployment planning is a difficult task not only due to high computation complexity but also because of the dynamic nature of a running application and its execution environment. Adaptive deployment is an approach that helps to address both these issues. It allows dividing the problem of deployment planning on an iterative process that can be improved in runtime and, therefore, an application can be deployed more quickly. It also enables an application to be reconfigured in reaction to the changes in environment and workload.

However, adaptive deployment has much more potential. Development of software adaptation is driven by three main factors in the computing field: (1) emergence of *ubiquitous computing*, which dissolves the boundaries for how, when, and where humans and computers interact, (2) increasing interest in *autonomic computing*, which focuses on systems able to manage and protect themselves with as little as possible human guidance, and (3) the need for *highly available systems*, that ensure operational continuity for extended periods and must accommodate change during those periods [60, 88]. Adaptive deployment can well support all these areas.

Previous work clearly indicate that introduction of adaptability to the deployment process substantially improves its usability and increases ap-

[34] Recently, this trend is even more noticeable with the advent of system virtualization techniques such as VMware (`http://www.vmware.com`) and Xen (`http://www.xen.org`).

plication flexibility. S. Zaidenberg et al. present in [130] how reaction to context changes ease deployment in ubiquitous environments. Similarly, in [38] Dubus and Merle show that dynamic deployment allows programmers to express contextual dependencies in the application structure. For example, appearance of a mobile node in a network may invoke a deployment action that installs and launches a desired component on the node. In this way the application can "expand" or "shrink" as nodes are appearing and disappearing in the network. P. Backx et al. in [7] present a prototype platform for autonomic adaptive deployment which aims at reducing network overhead. This solution uses a simple heuristic to move components towards or away from clients resulting in gains up to 20%. Moreover, the experimental results which we present in [4] show that even simple algorithm can substantially improve execution effectiveness. Comparing to even distribution of software components in the environment the reduction of the execution time for different use cases was in range between 5 to 60%. In [15] we discuss how adaptability allows controlling of QoS in so called *virtual redeployment* process. Further, Ayed and Berbers in [6] argue that adaptive deployment makes possible to introduce context-awareness into pervasive applications. Generally, adaptability of deployment allows reacting to context changes and reorganizing the application according to specified rules. It also enables delivering more sophisticated optimization objectives such as maximizing application throughput and minimizing mean response time that are hardly possible with only static and non-adaptive deployment.

2.4.1 Definition of Adaptation

Considering software engineering, there are two different meanings of software adaptation in general. One is closely related to Component-Based Software Engineering (CBSE) and deals with problems of constructing applications from Commercial Off-The-Shelf (COTS) components. This kind of adaptation aims at deriving automatically, from generated or end-user specified composition contracts, pieces of software — *adaptors* — to solve interaction mismatch [17]. The other meaning of software adaptation, and the one used in this work, is close to the general sense of this term which denotes *adjustment* to environmental conditions.[35] Again, this adjustment may be realized in two distinct ways:

a) as adjustment of a sense organ to the intensity or quality of stimulation or,

b) as modification of an organism or its parts that makes it more fit for existence under the conditions of its environment.

[35]Merriam-Webster's On-line Dictionary: adaptation[2]

From the software engineer standpoint the former kind is known as *parameter adaptation*, which modifies program variables that determine its behaviour. This type of adjustment, used e.g. in TCP, can tune parameters or direct an application to use a different existing strategy. It cannot, however, adopt new, unplanned strategies. The latter kind of adaptation, known in computer science as *compositional adaptation*, is able to exchange algorithmic or structural system components improving program's fit to the current environment state. With compositional adaptation, an application can not only tune its variables or select a strategy but it may recompose its components or adopt new algorithms for addressing concerns that were unforeseen in the development phase [88].

When component-based software is considered, the parameter adaptation is focused more on a single component adjusting its operation to the execution environment. Conversely, the compositional adaptation allows (re)organizing application components according to a higher-level adaptation strategy. Deployment of component-based systems is much more related to the latter type of adaptation. Therefore, this work concentrates mainly on compositional adaptation and its applicability to deployment process showing benefits that accrue from this combination.

2.4.2 Benefits of Adaptive Deployment

The computational complexity of planning phase derives from large search space[36] which planning algorithms have to scour for a viable solution. If deployment is static, meaning that each application component is running bound to a fixed location during whole execution time, well prepared plan is crucial and has great impact on application effectiveness. Having adaptive deployment mechanisms available at runtime, complex and time consuming exact planning may be replaced with heuristics, approximate algorithms or simply searching for any valid solution as proposed in [100]. Adaptive deployment allows relaxing requirements on planning algorithms for the quality of solutions found. Although usually a better initial plan will lower reallocation overhead during execution, this must not be a case in general. The simple rule is that for environments where reallocation is more expensive better and possibly more complex planning algorithms should be considered. On the contrary, in environments with lightweight reallocation mechanisms the planning may be limited to only low complexity approximate algorithms.

The next, significant benefit influencing effectiveness and accuracy of deployment planning is hidden behind describing resources of an execution environment and requirements of application components. As discussed

[36]As described in Sect. 2.3.3 the search space in spatial deployment planning grows exponentially depending on the number of nodes and the number of components.

previously, this seemingly simple prerequisite is not easy to achieve in open environments when nodes share resources between different components, applications or even operating systems. If only static deployment is possible, planning requires for all potential resource consumers a priori knowledge of the worst case conditions or specifying intricate resource availability and usage functions such as proposed in [63, 67]. Moreover, it requires some resource reservation mechanism what, in turn, implies that any resource allocation has to be under control of a resource manager. This makes static planning in open distributed systems of low efficiency and often limited usability. With the use of adaptation these constraints can again be relaxed.

By definition, adaptation allows following changes in the environment, hence enables resource management which is more sensible of the current allocation needs. Having adaptation mechanisms there is no real need for pessimistic resource allocation. It is only the matter of an appropriate adaptation algorithm to detect if any changes in deployment are needed to satisfy the current allocation demands. For example, when two components running on a single machine compete for the same resource and this resource becomes depleted, adaptive deployment can redeploy one of the components to another, more appropriate machine. Therefore, when describing component's requirements it is sufficient to specify only its *prerequisites* i.e. these requirements which if not satisfied, will restrain a component from execution. In result component and environment descriptors can be simplified because it is unnecessary to specify dynamic resource requirements and capabilities. Moreover, there is no need to predict resource allocation as well as to search for dependencies between resource usage.

Another advantage of the adaptive deployment process is the ability to be combined with any of the deployment dimensions defined earlier allowing for their more sophisticated event-driven behaviour. Adaptation can take control over deployment actions in response to very different events e.g.:

- (un)deployment of an application component,
- (dis)appearance of a node in the execution environment,
- change in reliability of a network connection,
- (dis)appearance of a service or component,
- appearance of an updated version of a component,
- a user-defined event.

In this way it enables achieving high-level goals such as meeting QoS requirements, improving availability, performance, reliability, etc.

Providing adaptive deployment for a component-based distributed system is, however, a difficult task. One of the major obstacles are dependencies between application components and between the components and their execution environment. Therefore, appropriate techniques and patterns are required to help disentangling these elements and to enable free deployment adaptation. Most commonly used patterns for this purpose are architectural reflection and autonomic computing.

2.4.3 Reflective Systems

The concept of reflection refers to the capability of a system to reason about and act upon itself. The term reflective system, as defined in [81], describes a system which (1) incorporates structures representing (aspects of) itself, and (2) the sum of these structures — self-representation — is *casually connected* to the system it represents. Casually connected means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behaviour, and vice versa [26]. Therefore, it is possible to state that:

a) the reflective system always has an accurate representation of itself,

b) the status and computation of the system are always in compliance with this representation.

In order to realize these statements reflection offers two activities: *introspection* to let an application observe its own behaviour, and *intercession* to let a system or application act on these observations and modify its own behaviour [88]. The exact meaning of these activities depends on the abstraction level the reflection is performed on, e.g. in computational reflection it is the programming language which provides support for reconfiguration of its objects and object models, while in architectural reflection the reconfiguration refers to the observation and manipulation of the graph of software architecture [37].

Component-based applications are especially amenable to architectural reflection because they are inherently realized as a graph of interconnected components. The problem of self-representation is usually resolved by means of an ADL used in design and development. Therefore, a component-based application to be fully reflective needs to provide the intercession mechanism and ensure that the model is casually connected with the application itself. In other words, consistency between the model and the running system must be satisfied.

Considering deployment of component-based applications, the architectural reflection expresses itself by providing a language to describe deployment in an execution environment and by delivering mechanisms to dynamically reconfigure application in runtime. In this work we show our approach to address these issues. However, given a reflective distributed application there is still a need for support of end-user deployment requirements. Complexity of deployment in heterogeneous distributed environments needs to be hidden behind a convenient interface which allows for high-level control. One way to reach this goal is to use autonomic computing approach.

2.4.4 Autonomic Computing

Autonomic Computing (AC) is inspired by the functioning of the human nervous system and aims at designing and building *self-managing* systems [61]. The main reason for autonomic computing appears whenever the scale and complexity of a system or application grows to the extent that their manual configuration and management is too challenging. Due to our autonomic nervous system we are freed by non-conscious activities from the low-level complexity of managing our bodies to perform high-level tasks — the conscious activities. Similarly, autonomic systems provide a high-level management interface hiding its low-level intricacies [114]. There is, however, important difference between autonomic capabilities in the human body, which are involuntary, and self-managing autonomic capabilities of computer systems. The latter can perform tasks according to a provided, adaptable policy rather than a hard-coded procedure. For example, if an operating system manages task scheduling and creates the illusion of parallel execution, we cannot call it autonomic unless it allows users to change this procedure with another one following different management policy.

Autonomic Computing initiative was first introduced by IBM in 2001. The main building blocks of the proposed model are a *managed resource* and Autonomic Manager (AM) (Fig. 2.9). The managed resource is a hardware or software component that provides any kind of management interface like a computer system, database engine or a mobile robot. It is accessible through a *touchpoint* — a component implementing standard sensor and effector interfaces. This reduces management complexity as, instead of diversity of manageability interfaces associated with various types of managed resources, the AM is provided with a well defined sensor and effector interfaces of their touchpoints.

The Autonomic Manager is a component that implements the control loop consisting of four stages: Monitor, Analyse, Plan and Execute (MAPE). Monitoring provides information from managed resources which is collected and analyzed. As a result of the analysis step, adaptation needs are determined.

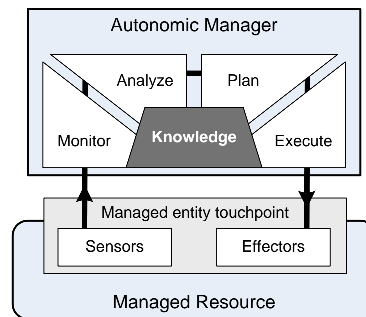

Figure 2.9: Building blocks of the Autonomic Computing model.

Further, a plan which fulfils the needs is created, and executed in the last stage. To realize this loop the MAPE components may use rules, correlations, beliefs, expectations, histories and any other useful information available to the AM [114].

The use of autonomic computing in software deployment and administration is a promising technique which has been noticed recently [10, 12, 24]. There are several important advantages of this approach:

- autonomic administration allows reconfigurations to be performed without human intervention,
- high-level support for deployment and configuration of applications reduces errors and administrator's effort,
- autonomic management is a method for better resource utilization as resources can be allocated on-demand.

With the increasing complexity of software and hardware system architectures, autonomic computing offers yet another advantage. Following the AC approach it is possible to create AM hierarchies dividing the management activities among different manager components. For large systems, this divide and conquer strategy can again improve manageability.

2.5 Adaptive Deployment Platforms

In this section we present selected existing platforms that have been developed to allow the adaptive deployment of distributed applications. Our analysis focus on four aspects which we regard as pivotal when adaptive deployment is considered:

1. Model of components — deployment operates on components, therefore, it is important to examine platforms and determine what a component is, how rich the component model is and if they are language independent.

2. Model of deployment — in this aspect we analyse planning dimensions supported by the platforms, if they support open or closed systems and heterogeneity of resources.

3. Model of adaptation — analysing adaptation we first focus on the level of deployment automation that a platform offers. Automation of deployment is the key enabler of adaptive deployment. Then we discuss what reconfiguration mechanisms the platforms offer and how a user is able to set desired adaptation policies.

This analysis allowed us to show the area covered by the existing solutions, point out their shortcomings and prepare for setting the requirements of adaptive deployment framework.

Prism-MW Platform

In [90] M. Mikić-Rakić presents Prism-MW — a platform and an architectural middleware for improving availability of a large system of small mobile devices. The key driver of the programming-in-the-small-and-many middleware platform is the assumption that the main source of degradation of the system's availability is disconnection. Therefore, the platform employs autonomous, run-time redeployment to increase system's availability by enabling the system to (1) monitor its operation, (2) estimate its new deployment architecture, and (3) effect the estimated architecture. Since estimating a system's optimal deployment i.e. deployment planning is an exponentially complex problem, Prism-MW provide a set of approximative algorithms with different levels of trade-off between complexity and achieved availability.

Prism-MW is an interesting approach showing how dynamic redeployment can increase the availability of systems of small and mobile devices. It proposes a set of models, algorithms, techniques, and tools for improving availability via runtime reconfiguration.

Model of components. Prism-MW defines a proprietary component model dedicated to small mobile devices. It has been optimized to run on resource-constrained devices with low amounts of memory and slow processing speeds.

The basic building block of the model is a `Brick` that encapsulates common features of its subclasses (such as `Component`, `Port` and `Connector`). Components perform computations in an architecture and may maintain their own internal state. Components interact with each other by exchanging events via their ports. Ports are the loci of interaction in an architecture. Each component can have an arbitrary number of attached ports each of which can connect to at most one other port. To support other kinds of delivery semantics (e.g. multicast, broadcast, anycast) the model defines connectors. Connectors are used to control the routing of events among the attached. In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at run-time.

Being dedicated to small devices, Prism-MW is simple and requires only minimal effort to master its basics. However, its simplicity reflects itself in too much simplification sometimes. Prism-MW defines the non-recursive model of a component what can result in a large number of components comprising an application. Moreover the relation between a component and a thread of execution resembles the active-object approach proposed by ProActive.[37] Therefore, components in Prism-MW are very fine-grained and can be compared to a language object rather than to a larger building block of an application. Another simplification in the model is the ability to communicate by event exchange only. Although this model can be used to address other patterns of communication such as Remote Procedure Call (RPC) or data streaming, it would require additional development effort to achieve this.

Prism-MW is language independent. It's core has been implemented in Java JVM but subsets of the functionality have also been implemented in Java KVM, C++, EVC++, Python, and Qualcomm's Brew.

Model of deployment. Prism-MW does not directly address the problem of component deployment in heterogeneous environments. Instead a user can use a separate tool Prism-DE to execute a prepared initial deployment plan. Prism-DE can ensure that a set of topological rules is consistent with a modelled topology. However, it uses very simplified model of a component and does not define resources except for their identifier and IP address. Moreover, it is unclear how, if at all, this information is used in runtime to ensure correctness of a new application deployment.

Model of adaptation. The Prism-MW platform is dedicated to improve system availability. It provides a set of algorithms for runtime deployment

[37] `http://proactive.inria.fr`

planning that differ in complexity and quality of the solution. The platform allows for automatic switching between the algorithms basing on collected historical data. However, given limited view on adaptation, Prism-MW does not allow users to specify their requirements or influence algorithm behaviour.

Considering level of deployment automation, Prism-MW together with external tools (namely Prism-DE and DeSi) offers automation of particular deployment activities. DeSi enables initial planning allowing an engineer to rapidly explore the space of possible deployments for a given system, determine the deployments that will result in greatest improvements in availability, and assess and visualize system's sensitivity to changes in specific parameters and deployment constraints. Prism-DE allows executing the created initial deployment plan i.e. it runs an application according to a specified plan. The extensions to the Prism-MW platform allow for runtime deployment planning and application reconfiguration.

Although the whole deployment process requires user intervention and, therefore, is not fully automatized, in runtime an application can be automatically reconfigured. Similarly to our solution, the basic reconfiguration mechanism used in Prism-MW is runtime component migration. It is not clear, however, how Prism-MW addresses the problem of reaching quiescence state, state portability and reconnection. Due to simplified model of communication that is limited to events transferring, we assume the problem of migration is reduced to serialization of the component state together with queued events and moving these data to a target location.

Jade and TUNe

Jade is a framework to ease deployment of Java 2 Enterprise Edition (J2EE) applications. It provides automatic scripting-based deployment and configuration tools for clustered applications using autonomic computing approach [5, 9]. The main principle of Jade is to wrap legacy software elements in components and administrate this software infrastructure as a component architecture. To model the infrastructure Jade relies on the Fractal[38] ADL model [12].

In order to implement wrappers encapsulating existing software and to implement reconfiguration programs in Jade, the administrator of the environment has to learn yet another framework. Therefore, more recently work on Jade evolved to the TUNe platform which aims at providing a higher level formalism for wrapping, deployment, and reconfiguration. The main motivation is to hide the details of the component model the framework relies

[38] http://fractal.ow2.org

on and to provide a more intuitive policy specification interface [12]. TUNe introduces three main elements: (1) a wrapping description language used to specify the behaviour of wrappers, (2) the Unified Modeling Language (UML) profile for describing deployment schemas which they use instead of Fractal ADL and (3) the UML profile to describe reconfiguration state diagrams which operate on previously described deployment schema.

The main application target for Jade and TUNe is the administration of servers distributed over a cluster of machines or a grid infrastructure.

Model of components. Jade and TUNe address the management of very heterogeneous software which includes web servers, servlet containers, EJB containers and database engines. Due to diversity of the managed software the key design choice in the frameworks is to rely on a component model to provide a uniform management interface for any managed resource. Both use Fractal as a component model which is a modular, extensible and programming language independent. The Fractal components have a reflective structure that is organized into membrane and content. The membrane of a component defines its abstractions, interfaces and its meta-level methods, arranged in specialized controllers and providing the introspection and reconfiguration operations [112]. The content may include either the code directly implementing functional component behaviour (primitive) or other components (composite).

Model of deployment. To model software components both Jade and TUNe use Fractal ADL. TUNe provides also a specialized Wrapping Description Language (WDL) which is used to specify behaviour of wrapper components to hide complexity of Fractal ADL. Moreover, in TUNe deployment is based on a UML profile for describing deployment schemas. The deployment schema describes the general organization of the deployment (types of software to deploy, interconnection pattern). Neither Jade nor TUNe, however, does not model execution environment except for basic information such as `hostFamily` attribute which gives a hint regarding the allocation of nodes [21]. This limits applicability of the solutions to clusters of similar machines rather than heterogeneous systems.

Model of adaptation. Both frameworks are based on the principles of reflective and autonomic software. Any software managed with Jade and TUNe is wrapped in a Fractal component which interfaces its administration procedures. Considering deployment automation, Jade requires additional operations performed by hand to organise the deployment, such as starting Jade daemons/servers, registering them in a naming service [47]. Moreover,

neither Jade nor Tune address the deployment planning problem what again limits automation of deployment.

Once an application has been deployed it is monitored and reconfigured dynamically. TUNe provides a UML profile to define state diagrams that are input to the manager which automatize the reconfiguration process. The diagrams facilitate expressing reconfiguration goals such as self-repair in the case of server failure or self-optimization illustrating dynamic (de)allocation of nodes. Jade and TUNe address reconfiguration of legacy software, therefore, the use of more advanced reconfiguration mechanisms such as migration is limited.

Grid Component Model and ProActive

Grid Component Model (GCM) [24] is an extension to the Fractal component model in order to better target Grid infrastructure which is characterized by highly dynamic, heterogeneous and networked target architectures. A prototype implementation of GCM is currently available as part of the ProActive library from the GridCOMP project.[39]

The Grid Component Model may be perceived as a competitive with the CCM model.

Model of components. GCM builds on the Fractal component model and exhibits three important features: hierarchical composition, collective interactions and autonomic management. Similarly to Fractal, a GCM component is composed of two main parts: the membrane and the content (shortly presented above). The GCM supports variety of communication patterns such as: many-to-one and one-to-many communication, and different communication semantics including asynchronous remote method invocation, events based, and streaming based communication. GCM components support also autonomic behaviour as well as functional and non-functional adaptability. This entails the ability to add, replace and remove dynamically entities, and also dynamically changing the bindings between components. Overall, the GCM is a very rich component model for distributed applications.

Model of deployment. Deployment of components in GCM is relying on virtual nodes. A virtual node is an abstraction of an execution node which allows for separation between logical and physical infrastructure. Virtual nodes are used in the code or in the ADL description to abstract names, creation and connection protocols to physical resources. They may encapsulate

[39] `http://gridcomp.ercim.org`

resource properties and cardinality to ensure that during deployment planning components bound with a particular virtual node are located on an appropriate physical resource.

However, more sophisticated description of application requirements and environment capabilities are left for the future. GCM envisage extensions with respect to instance topology of nodes, including point-to-point QoS, hardware or OS constraints, interconnect preferences. In the future, it might also introduce constraints at the level of the component itself [24, Sect. 6].

Model of adaptation. GCM is designed around the concepts of Autonomic Computing and reflective software. The model envisage several levels of autonomic managers embedded in components, that take care of the non-functional features of the component programs. The bottom level is represented by components having no autonomic control at all. At the next level lie components exhibiting a passive behaviour: they have (non-functional) server interfaces only, for both introspection and intercession. The top level in the range consists of the fully autonomic components. They exhibit self-management skills with respect to all or some of the *self-** aspects. Additionally, an interesting concept proposed by the GCM model are behavioural skeletons [2]. They provide a programmer with the ability to implement autonomic managers without taking care of the details related to parallelism.

ProActive, as a platform implementing GCM, offers a vast number of tools that can support different adaptation needs. For example, it allows for monitoring, load balancing and runtime object migration. The migration is transparent and happens while the application containing the active objects is running and without interruptions in the application. Load balancing enables either the work sharing or work stealing approach to share load between nodes. In the low level the load balancing uses migration of ProActive objects between nodes. However, neither load balancing nor migration functionality are available at the GCM level [93, Sect. 29 and 31]. Moreover, both ProActive and GCM lack ready-to-use solutions for adaptation of component deployment.

Internet Operating System

K. El Maghraoui proposes in [82] a framework for dynamic middleware-triggered reconfiguration of applications in Grid environments. The Internet Operating System (IOS) middleware provides a virtual execution environment that hides complex resource management and reconfiguration issues from high-level applications. IOS has been designed with the following key characteristics: (1) architecture modularity to allow for extensible and pluggable

reconfiguration mechanisms to accommodate various execution environments and different applications, (2) generic interfaces to allow various programming models and technologies to leverage IOS dynamic reconfiguration mechanisms, (3) decentralized strategies to achieve scalable decisions, and (4) adaptability to dynamic environments to allow applications to adjust their resource allocations following the availability of resources.

IOS is currently being used as an experimental test bed mainly for scientific applications. It is a step in ongoing research efforts to realize the vision of efficient, seamless, and easy deployment of applications in dynamic grid environments.

Model of components. An IOS-enabled environment is a set of agents that forms a virtual network. The agents are implemented as SALSA actors and inherit all their autonomous features. Simple Actor Language System and Architecture (SALSA) is a language for developing actor-oriented applications [122]. It provides programming abstractions to implement actor primitives such as creation and asynchronous communication. Actors are inherently concurrent and distributed objects that communicate with each other via asynchronous message passing. They encapsulate state and a set of methods and are controlled by a single thread of execution. Moreover, SALSA's runtime environment provides support for actor migration. All this makes SALSA actors very similar to active objects defined by ProActive. Both actors and active objects exhibit very fine-grained structure and merely the event-based communication model.

The actor model in IOS has been also used to manage existing MPI applications. For this purpose, the platform provides IOS/MPI proxy that implements an actor emulation mechanism whereby MPI processes are viewed by the IOS middleware as actors. This allows MPI processes to take advantage of the autonomous nature of actors such as universal naming, asynchronous message passing, and migration capabilities. The presented approach resembles the previously discussed AC model which has also been used by Jade and TUNe to manage software.

Model of deployment. The grid in IOS is considered as a set of resources modelled as a graph where the vertices represent computational resources and the edges represent the network connectivity between them. However, the work focuses mainly on load balancing issues using the work stealing approach. To be usable this approach requires that the execution environment is prepared to run mobile actors a priori. As IOS concentrates on runtime aspects of reconfiguration, the deployment problem not considered at all.

Model of adaptation. IOS provides mechanisms that allow analysing profiled application communication patterns, capturing the dynamics of the underlying physical resources, and utilizing the profiled information to reconfigure application through migration or malleability. For runtime entity reconfiguration IOS uses SALSA actors and the PCM library. The Process Checkpointing and Migration (PCM) library enables dynamic reconfiguration of iterative MPI programs by providing the necessary tools for checkpointing, and migration.

From the user point of view the focus of IOS is on load balancing of iterative applications. Reconfiguration in IOS has been designed to work with the applications with regular distribution of data.[40] The platform provides different strategies for load balancing such as random work-stealing, application and network topology sensitive work-stealing. However, the proposed splitting and merging policies cannot directly be used for other application models and non-uniform data distribution. Devising malleability strategies for non-iterative applications is left for the future. Moreover, the current PCM extensions do not consider multi-threaded MPI systems what again limits the area of application for adaptive mechanisms offered by IOS.

2.6 Conclusions

With the increasing complexity and distribution of computer systems there is also a growing demand for automatic and adaptive deployment tools. This chapter reviewed state-of-the-art solutions in the area of software deployment and adaptive software deployment. Although many efforts are undertaken to resolve the issues involved in application and system deployment, they are frequently limited in scope. Table 2.1 presents the area of major challenges and which of these are met by the adaptive platforms presented in the previous section.

Rich Component Model. One of the defining properties of components, expressed by Szyperski in [119], is that they are units of independent deployment. This implies that deployment platforms should operate on a component level. However, the definition of ‘component’, or more precisely, ‘deployment unit’ differs substantially between these platforms. Jade/TUNe operates on a legacy software which is managed by wrapper components developed in Fractal, IOS is based on SALSA actors that resemble distributed objects rather than components, Prism-MW proposes a simplified component model which again is similar to distributed objects. In our opinion only GCM

[40] It supports block, cyclic, and block-cyclic data distribution models. More about this can be found in [78].

Table 2.1: Main challenges for adaptive deployment framework and how they are met by existing solutions.

Challenges	IOS	Prism	Jade/ TUNe	GCM/ ProActive	Target ADF
rich component model	✓	✓	✓✓	✓✓✓	✓✓✓
progr. languages independent	–	✓✓	–	✓	✓
deployment dimensions	spatial	spatial	spatial	spatial	spatial, temporal,* semantic*
open execution environments	✓	–	–	✓	✓✓✓
resource heterogeneity	✓✓✓	✓	–	–	✓✓
resource description language	–	–	–	–	✓✓
support for virtualization	–	–	–	✓	–
deployment automation	–	✓✓	✓✓	✓✓	✓✓✓
runtime reconfiguration	✓	✓	✓	–	✓✓✓
user-defined adaptation policies	–	–	–	–	✓
standard-based	–	–	✓	✓	✓✓

*support for this is limited to modelling only.

defines a component model that could be perceived as rich in distributed environments. Our work is based on the competitive CCM model. The main features supported by both CCM and GCM components are: multiple communication ports, different communication styles (synchronous RPC, asynchronous event-based communication and data streaming), multi-threading, interoperability,[41] hierarchical composition.[42] In this work we address the problem of deployment in a selected rich component model.

Programming Languages Independent. All of the reviewed frameworks were created in the Java language. This was also the main development language for GCM/ProActive and Prism. However, GCM is a generic model that could potentially be implemented in other languages, whereas selected functionality of Prism-MW was implemented in C++, Python and other programming languages. A deployment framework should promote interoperability between languages what increases its usability.

[41]Currently, interoperability of the GCM components is limited because the only existing implementation is built in Java.

[42]Although CCM does not allow for explicit component aggregation, a hierarchy of components can be created using the D&C packaging model.

Deployment Dimensions. All of the presented platforms consider only spatial dimension of deployment i.e. mapping of components to execution nodes. Unfortunately, support for the other two dimensions is rarely present in adaptive and non-adaptive platforms alike. Currently, we observe a growing research interest in the area of semantic deployment and, therefore, expect more solutions to become available in the near future. Support for different deployment dimensions greatly increases applicability of a deployment framework and, therefore, is a valuable feature. In this work we mainly support the spatial deployment dimension, however, our deployment models enable representing the other two dimensions.

Open Execution Environments. Deployment in distributed systems becomes especially interesting when the execution environment is not closed and predictable but allows for resource sharing between different applications. Prism focuses on improving software availability in the presence of connectivity loss and does not consider other applications running concurrently. Similarly, Jade and TUNe address the problem of reconfiguration to face high loads and provide higher scalability. However, they focus on adaptation of a single service only. Conversely, both IOS and GCM provide tools to monitor execution environment in runtime and, therefore, can indirectly support open environments. It is a desirable and useful feature, if an adaptive deployment framework can support open environments.

Heterogeneity of Resources. This is one of the most important features of a distributed execution environment. If all resources are homogeneous such as in a cluster system, the problem of distributed deployment is simplified to transferring software artifacts and executing software in a coordinated manner. The key challenges in deployment, like complexity of deployment planning, stem exactly from the resource heterogeneity. Among the reviewed platforms only IOS considered this issue in depth. In Prism heterogeneity of resources is also discussed but the solution is limited to memory resources only. A deployment framework targeted at distributed systems should tackle heterogeneity of resource lest it become a limitation of usability.

Resource Description Language. One of the most developed models of resources CIM has been proposed by DMTF for more than ten years. Nevertheless, none of the platforms use resource description languages to represent application requirements and resource capabilities in a consistent manner. This feature of a deployment platform is especially important in case of the model-based approach. Then use of a proper description language increases portability of application and execution environment descriptors. The same

application package can be deployed in different environments and the same environment can be used to host different applications.

Support for Virtualization. Virtualization in software deployment is a relatively new concept. On the one hand it can greatly improve usability of a deployment platform, on the other hand it increases its complexity. Among the reviewed platforms only GCM/ProActive considers virtualization in deployment. However, the notion of `VirtualNode` is only a limited view on the potential value hidden in software deployment in virtualized systems.

Deployment Automation. To conduct proper deployment adaptation it is needed to ensure automation of substantial part of the deployment process. This includes at least the installation, activation and reconfiguration activities. Most of the reviewed platforms offered automation of these steps excluding, however, initial deployment planning. Only Prism provided tools to ease initial deployment planning, however, they needed human intervention. Our goal is to automatize this three aforementioned steps such that given merely a software and environment description it is possible to launch and adapt application deployment.

Runtime Reconfiguration. Except from GCM/ProActive all other platforms deliver tools for runtime application reconfiguration. In case of IOS and Prism-MW it is the runtime migration mechanism, whereas Jade/TUNe allows changing component attributes or bindings between them what can reflect itself as e.g. add or remove a software replica. The GCM/ProActive platform does not offer runtime reconfiguration mechanisms. Despite the fact that ProActive enables object migration this feature is not available at higher, GCM component level. It is a real challenge to support advanced reconfiguration mechanisms such as migration in rich component models represented by CCM or GCM.

User-defined Adaptation Policies. The reviewed platforms do not allow users to control how the adaptation is performed. Instead, they propose a set of algorithms that improve certain aspects of application behaviour in a predefined way. For example, Prism addresses application availability using three deployment planning algorithms that satisfy the constraints posed by memory and restrictions on the locations of software components. IOS offers a set of work stealing algorithms to control system reconfiguration. However, devising fully autonomic adaptation algorithms that would fit any application and environment seems to be hardly possible. Therefore, to improve usability of adaptation the underlying algorithms should enable users to take part in

the decision process. We aim to provide users with some interface to control the adaptation process.

Standard-based solution. Often the existing deployment platforms propose their own architecture description languages, resource description languages and deployment models. Currently, however, there exist well defined standards that can and should be used instead of resolving the same problems again and again. For example, using a common resource representation proposed by the CIM standard allows avoiding naming coordination problems and improves reuse of application and environment models. It is desirable yet not always simple to adhere to the existing standards. However, to increase portability a deployment framework should promote standard-based development.

* * *

The presented analysis shows the inherent complexity of the software deployment problem in distributed environments. Currently, two major trends trying to escape from deployment intricacies are visible. Firstly, raising hopes are reposed in Software as a Service (SaaS), cloud computing and the “mashup” approach. Secondly, software is more and more often delivered as a black box containing a virtual image of the whole software stack from an operating system to a user application.

We argue that neither of these approaches does not really solve the problem. To effectively deliver reliable services and reliable software a comprehensive deployment platform has to be provided. We cannot simply avoid service deployment in SaaS leaving it as a providers’ issue. Providers need tools to automatically deploy the service and more, they need tools which can easily adapt to current users’ needs and resource availability. However, even if we assume the services are ‘there’, and the providers have ensured to run and maintain them, a challenge which immediately appears is the planning in the semantic dimension. One of the questions it raises is how from the set of equivalent services choose these which are the most appropriate from the user standpoint.

Delivering software in the form of a virtual image has also many drawbacks. From a customer point of view it eliminates the problem of assembling software into a complete application. However, an image has to be preconfigured and, therefore, can satisfy needs of only a certain subset of customers. For others, different software configurations should be offered what ultimately leads to a vast number of virtual images each of which enclosing a distinct combination of configuration settings. Maintenance of such a library of images is a significant effort. Additionally, this full-stack delivery does

not promote proper software sharing and may waste a lot of resources at a consumer site.

Instead of evading the issues connected with software deployment, in this work we analyse the needs and propose a solution to the deployment problem addressing requirements of component-based distributed applications.

Chapter 3

Towards Adaptive Software Deployment

The previous chapter has outlined the main challenges in the area of software deployment. Many of them remain open, hence to bridge this gap we propose the adaptive approach to deployment. We argue that adaptive reconfiguration of software deployment is the key feature that enable effective execution of distributed systems. In the case of static approach, the initial deployment, even if well suited, is not sufficient to follow inevitable changes in application workload and resource availability and may quickly deteriorate. Conversely, adaptation of deployment makes the software able to react to the changes. It also allows for simplification in application and environment definition by avoiding the need to include dynamic resources in their descriptors.

In this chapter we present the overall concept of our adaptive deployment platform for distributed systems. Our discussion concentrates around the following key aspects (Fig. 3.1):

- plain deployment platform
  - component-based application design,
  - distributed application and execution environment,
  - planning dimensions,
  - virtualization,
- adaptive deployment platform,
  - resource monitoring and management,
  - application monitoring and management,
  - runtime deployment planning.

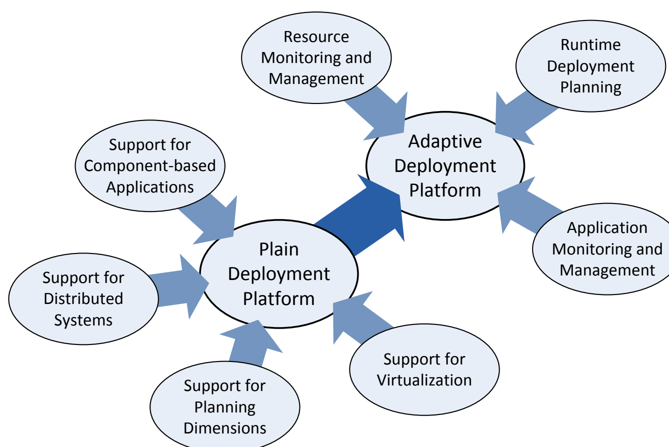

Figure 3.1: The key aspects leading towards the adaptive software deployment platform.

To develop our concept we leveraged the existing knowledge in the area of software deployment and based the solution on the Deployment and Configuration of Component-based Applications (D&C) specification and the Common Information Model (CIM) schema. Both deliver a very general models that are pivotal to software deployment. The D&C specification standardizes many aspects of component deployment including: component configuration, assembling, packaging, package configuration and deployment. Dearle[1] in [29] characterized it as "perhaps the most complete attempt to define a deployment and configuration standard." The strength of D&C is in its compliance with the Model Driven Architecture (MDA) approach. It defines a Platform Independent Model and allows for its further customization with Platform Specific Models.[2] D&C follows the idea that in general the deployment process remains the same independently of the underlying technology of software realization. The CIM schema provides a definition for resource representation that alleviates problems with coordination of resource descriptions. For example, some systems could report its processor name as "Intel(R) Pentium(R) 4 CPU 2.66GHz", whereas others would describe the same CPU as "Pentium(R) 4". This mismatch in description results in problems with deployment planning because it is often not obvious how to verify if a particular requirement can be satisfied by available resources. Kotsovinos noticed in [70] that most distributed deployment platforms either do not tackle the naming coordination problem, or address it by allowing only specific resources to be declared. We ensure portability of application and environment descriptions using a well established resource representation schema defined by CIM.

[1]The author is, inter alia, a co-editor of the Proceedings of Third International Conference on Component Deployment, Grenoble, France, November 2005 [30].
[2]For example, a PSM for CCM is defined in [103, Chap. 14].

3.1 Plain Deployment Platform

For many basic activities an adaptive deployment platform requires support from the plain deployment infrastructure. Therefore, first we present the key points that have the major influence on the overall concept of our adaptive deployment platform.

3.1.1 Support for Component-based Applications

Currently, we are surrounded by various software technologies offering different meanings for the term ‘component’, therefore, it is important to present the definition which we adopted in this work. Szyperski in [119] presents three defining properties of software components — they are units of: *composition, state encapsulation*, and *independent deployment*. The same three properties can be found in the definition provided by the D&C specification which is one used in our work. It describes a component as:

> a *modular* part of a system that *encapsulates its contents* and whose manifestation is *replaceable* within its environment. A component defines its behaviour in terms of provided and required interfaces. As such, a component serves as a type, whose conformance is defined by these provided and required interfaces.

Modularity is the property which guarantees that application is not a large black-box but a composition of smaller elements. Each of these elements encapsulates a part of the application code and possibly state when the application is running. Lastly, the property of being replaceable not only indicates the ability to be replaced by other components e.g. provided by different vendors but also enables independent deployment. If a single component can be replaced in the whole application, it implies that this component can be independently deployed.

Apart from these three key component properties, the other important detail in the presented definition is the specification of component’s behaviour in terms of provided and required interfaces. This manifests itself by: (1) creating clear boundaries of a component, (2) separating a component from its execution context, (3) enabling its replaceability and (4) ability to express dependencies between components in a well defined form. The last property is valuable because many of the existing deployment solutions, even if considering an application as a set of software components, distinguish them merely by reference to its name and/or version.[3] Nonetheless, for adaptive

[3]To name some examples: InstallAnywhere, Java Network Launching Protocol, N1 SPS, Solution Deployment Descriptor, SmartFrog, Microsoft Installer.

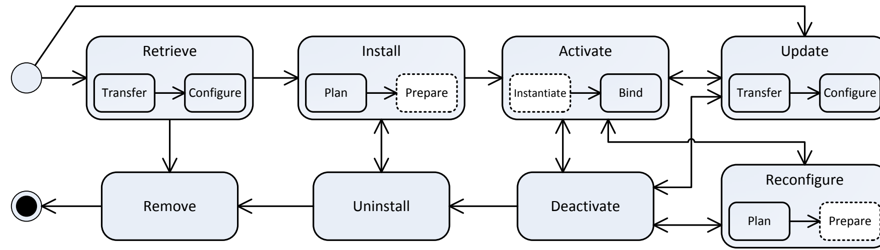

Figure 3.2: The global view on deployment activities.

deployment even more significant is separation between a component and its execution context. It facilitates creating a highly transparent adaptation mechanism. Our main idea is to keep as much of the adaptation infrastructure as possible out of the component business code and free developers from their intricacies. For example, to monitor intensity of communication between components we would like to intercept the traffic in a components' container such that the components were unaware of the estimation being in progress. Similarly, to move a component between execution nodes most of the migration details should be hidden from programmers.

3.1.2 Support for Deployment in Distributed Systems

In order to deploy software in distributed execution environments the general definition of the deployment process presented earlier in Sect. 2.1 needs to be supplemented with additional details. Deployment in distributed environments can be split onto two abstraction levels: global and local. Globally, it encompasses all activities that involve application or target environment as a whole. For example, planning of component distribution in the execution environment is the activity that needs to be done at the global scale. It refers to every application component and decides where in the environment would be the best to run it. Locally, software deployment involves a single execution node and only these application components which were assigned to it. In this case deployment planning may need e.g. to determine how many containers are necessary for the components to satisfy their configuration requirements. Differences between these two levels are depicted in Fig. 3.2 and 3.3.

As presented in Fig. 3.2, deployment performed at the global level includes most of the defined earlier activities. Moreover, their meaning is largely consistent with the provided definition except for `Prepare` and `Instantiate` steps. At the global level software installation and reconfiguration activities coordinate rather than perform preparation, which is actually done at the local level. Similarly, after the preparation step and during

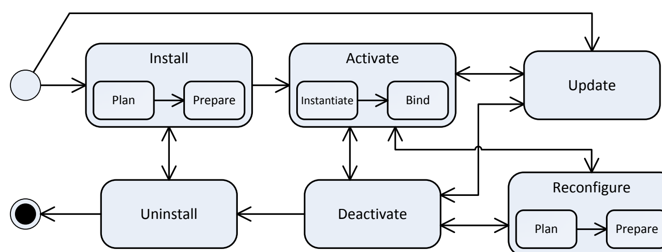

Figure 3.3: The local view on deployment activities.

activation the instantiation of components is also delegated to the proper execution nodes.

At the local level the deployment process is simplified and meaning of its activities is slightly different because it depends on the kind of the target execution environment. First, as shown in Fig. 3.3, it does not embrace `Retrieve` and `Remove` activities. They are done at the global level only. The former refers to transferring software from a producer's site to a consumer's global repository where it is configured, whereas the latter removes the software from the repository. Second, installation does not pertain to the whole environment but only a single execution node. In the case of middleware-based execution environment, which is considered in this work, planning at the local level determines containers required to run software components. Next, the preparation step involves running the containers to make the environment ready to instantiate the components. Later, activation refers to running the software components on the node and performs local component binding. The `Bind` deployment step is also present at the global level where it refers to binding components between execution nodes. Similarly, the `Reconfigure` activity is present on the global and local levels. Locally, however, it refers to changes within a single execution node such as e.g. creating a separate container in the case of high workload to make better use of multicore CPU architecture. The global and local separation of reconfiguration very well corresponds with the hierarchical approach proposed by Autonomic Computing.

Another aspect relevant to deployment in distributed systems is the automation of this process. As discussed earlier in Sect. 2.2.2, three major solutions to this problem are:

- script-based deployment,
- language-based deployment, and
- model-based deployment.

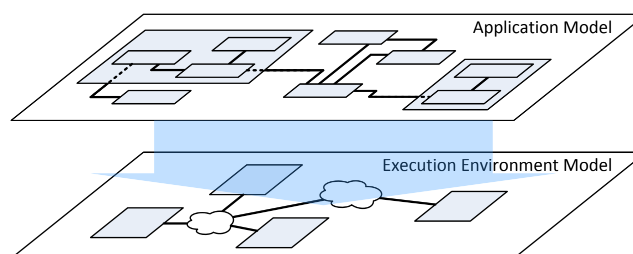

Figure 3.4: Separation between models of the software application and execution environment enables automatic planning.

For adaptive software deployment, however, only the model-based approach is suitable. It enables full automation of the planning phase, whereas the other two solutions would require human intervention in preparing appropriate scripts or deployment descriptors. The key feature of the model-based deployment that enables automation is in creating two separate models that describe software application and execution environment. Provided this, planning can automatically match elements of these models (Fig. 3.4). How in detail this matching is done, however, depends on planning dimensions considered e.g. in the case of spatial deployment it refers to looking for "the best" execution node to run a software component. The next section discusses this aspect more thoroughly.

3.1.3 Support for Deployment Planning Dimensions

Planning dimensions, which we defined in the previous chapter (Sect. 2.3.2), are especially related to model-based deployment. They enable addressing certain deployment characteristics that cannot be pre-programmed (e.g. using a scripting language) but must be expressed in a declarative way. Usually however, the existing deployment models focus on the spatial dimension only, which is important but alone does not allow for more elaborate temporal and semantic dependencies. Using only spatial deployment a planner may not be able to tackle problems such as execution ordering or using an existing service instead of creating a new component instance.

The concept of our adaptive deployment platform includes model-based deployment in all three dimensions defined earlier. This enables fully automatic planning and opens way to implement comprehensive deployment tools for distributed component-based applications. Appendix C describes our extensions to the D&C specification required to model deployment planning in all dimensions discussed in this section.

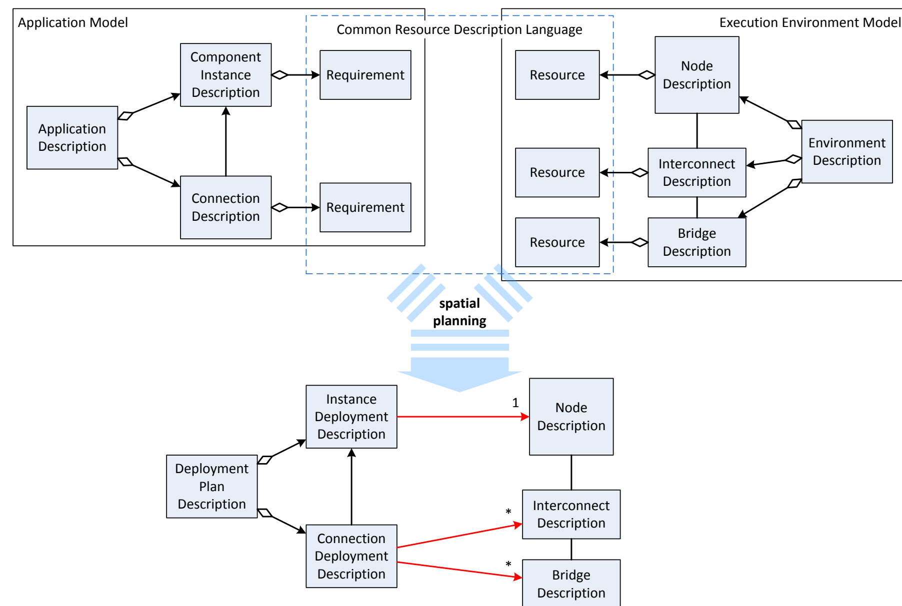

Figure 3.5: General view on the spatial planning dimension.

Spatial Deployment Planning

In order to plan software deployment in the spatial dimension two inputs are required (Fig. 3.5). On the one side, a planner needs an application model consisting of software components that are connected using their provided and required interfaces. Both the components and the connections declare resource requirements that must/should be satisfied by the environment. On the other side, the planner needs a model of target environment consisting of execution nodes that are interconnected with network links and bridges. All environment elements declare resources that can be offered to components and their connections.

Planning of deployment in the spatial dimension tries to find “the best” assignment of each application component to an execution node taking into account requirements of the component itself and all its connections. For the planner to do the proper matching both requirements and resources need to be described in the same resource description language (the CIM schema can well be used for this purpose). The output from the planner is a deployment plan that defines the assignments for all components and connections. Each component instance in the deployment plan is matched with exactly one execution node, whereas a connection may traverse multiple interconnects and bridges.

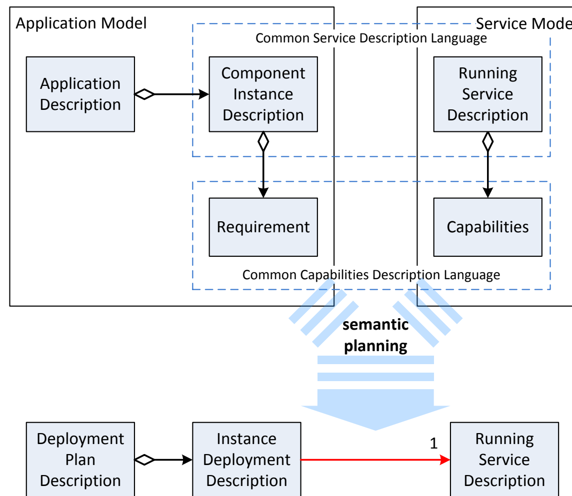

Figure 3.6: General view on the semantic planning dimension.

Semantic Deployment Planning

The basic elements needed for semantic planning are twofold: (1) an application model including component requirements and (2) a service description comprising its capabilities (Fig. 3.6). Descriptions of a component instance and a service allow for matching functional features i.e. to determine semantic equivalence between them, whereas non-functional characteristics such as mean response time or a number of processed transactions per second are expressed by requirements and capabilities. Planning in the semantic dimension tries to find "the best" available service that can be used as an application component. However, the planner to be able to match between component and services needs proper description languages such as IDL or WSDL for functional description and e.g. "UML Profile for Modelling QoS and Fault Tolerance Characteristics and Mechanisms" [104] for non-functional description. The output from the semantic planner is a deployment plan that for each application component provides a reference to a running service.

A service is usually a black-box entity which internal structure is only rarely disclosed. From the point of view of the planner, services can be assessed merely by observing their exposed features and behaviour. Therefore, the model required to plan deployment in the semantic dimension is simplified when compared to the spatial dimension. Complexity of the semantic planning is, however, comparable to planning in the spatial dimension.[4]

[4]To prove this it is enough to notice that for each application component: (a) every machine that hosts a semantically equivalent service could be a potential execution node for that component and (b) every potential execution node with the component deployed becomes an available service to be considered by the semantic planner.

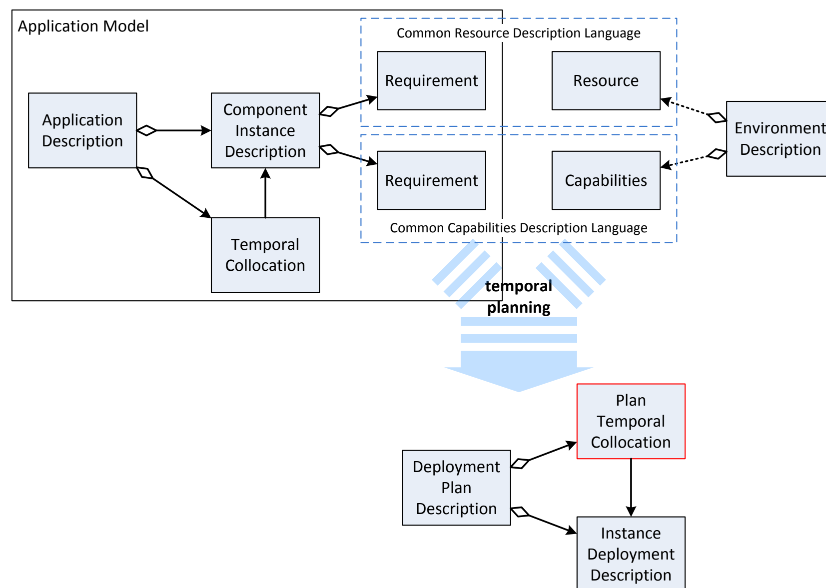

Figure 3.7: General view on planning in the temporal dimension.

Temporal Deployment Planning

In order to support the temporal deployment planning we propose a simple solution based on the work from the job scheduling area [3, 85, 107, 113]. On the one side, the planning uses the temporal collocation structure that enables creating order of deployment actions, on the other side it is related to some resource and/or capabilities available in the execution environment (Fig. 3.7). The result of planning in the temporal dimension is a plan than comprises temporal collocations between components modified according to the constraints implied by matching requirements to resources and/or capabilities. In the same time the temporal collocations from the plan are consistent with the deployment order defined in the application model.

We distinguished three kinds of temporal collocations. The `StartToStart` and `FinishToFinish` allow creating synchronization barriers, whereas `FinishToStart` provides means to create deployment task sequences. Using these collocation kinds[5] together with temporal collocation structures, a packager and planner can easily represent a broad range of Direct Acyclic Graphs modelling relations between deployment of component instances. In Fig. 3.8 we present an example of an application with such dependencies. The collocation structures bind together a number of component instances with a selected collocation kind.

[5]All of them are borrowed from the project management discipline where they are used in defining task dependencies.

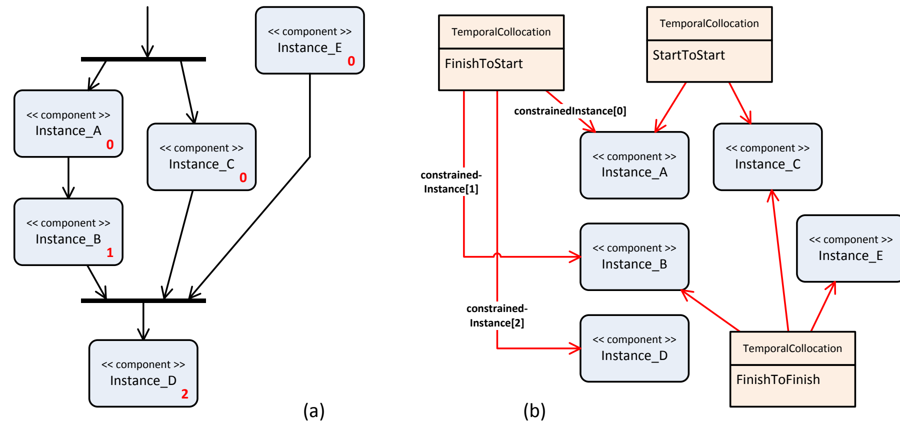

Figure 3.8: A sample application with temporal deployment dependencies represented (a) as a DAG and (b) using the `TemporalCollocation` structures; the numbers in red show a viable sequence of deployment tasks.

Purely temporal dimension planning operates on a single machine what effectively becomes equivalent to the local task scheduling problem. When combined with other dimensions it considers resources of execution nodes and/or capabilities of available services and address the problem of distributed job scheduling. It is important to note, however, that temporal deployment planning is not a way to achieve processing of workflows and should not be compared to approaches such as Spatio-Temporal Component Model (STCM) proposed in [11]. This is because the intention of a deployment engine is not to participate in data flow between components but only to instantiate and interconnect them according to a specified order. Workflow modelling languages aim at offering fine-grained structures that create a Turing-complete language,[6] whereas our intention is for the temporal deployment planning to operate on coarse-grained deployment activities (such as `Prepare`, `Instantiate`, `Bind`).

Apart from the deployment model, which is relatively easy to equip with structures for modelling temporal dependencies, an important issue is also plan realization. Execution of a plan comprising temporal collocations requires a significant change in a plan executor facility. When the spatial or semantic deployment planning are considered the plan executor can be a passive entity that simply instantiates and binds application components in reaction to user requests. Conversely, temporal deployment requires not only the executor to start and stop component instances but also a mechanism to constantly monitor current components' status. Only, in reaction to the changes in components' deployment states can the plan executor enforce

[6]For example, a widely accepted Business Process Execution Language (BPEL) is considered to be Turing-complete [40].

appropriate deployment order. Later in this work we present a mechanism enabling realization of temporal deployment plans.

3.1.4 Support for Virtualization

Virtualization is very common in contemporary computer systems and, therefore, becomes an important aspect that any deployment framework should consider. At least two major benefits stem from supporting virtualization in the deployment process. Firstly, it can isolate resources between applications making the execution environment more predictable what, in consequence, simplifies the planning problem. Secondly, it gives measures to configure and manage software systems in more granular way what enables interesting adaptation techniques. However, this is also an earnest endeavour to implement an infrastructure that enables distributed deployment on many different virtualization levels. In this section we shortly present the main issues that need to be addressed to consider virtualization in the distributed deployment process. More detailed discussion would require separate, extensive research which is out of the scope of this work.

Proper description of an execution environment is an essential part of a deployment framework, especially now with the advent of system virtualization techniques such as VMWare,[7] Xen[8] and Solaris Containers.[9] Virtualization may express itself not only on the operating system layer, though. A web application server can host a Java Enterprise Edition (JEE) application, a database server can host a data table, and a JVM process can host a Java application. Virtually anything that can receive an installable object may be considered an execution environment [62]. Consequently, a deployment platform needs to have the ability to support this feature of software systems which sometimes are perceived as software components and sometimes as an execution environment (Fig. 3.9).

Unfortunately, to the best of our knowledge, none of the existing ADLs support describing virtualized distributed systems. Usually, they provide a simple two-layered model with a component-based application being deployed over a distributed execution environment. Although the application models often allow creating recursive component hierarchies of arbitrary depth (e.g. D&C, Fractal), this kind of recursion is related to the same, single virtualization level. We term it *horizontal component aggregation*. Only the Solution Deployment Descriptor specification [92] addresses multilayered environments but it is limited to non-distributed systems.[10]

[7] http://www.vmware.org

[8] http://www.xensource.org

[9] http://www.sun.com/software/solaris/virtualization.jsp

[10] SDD provides "ability to describe software solution packages for both single and multi-

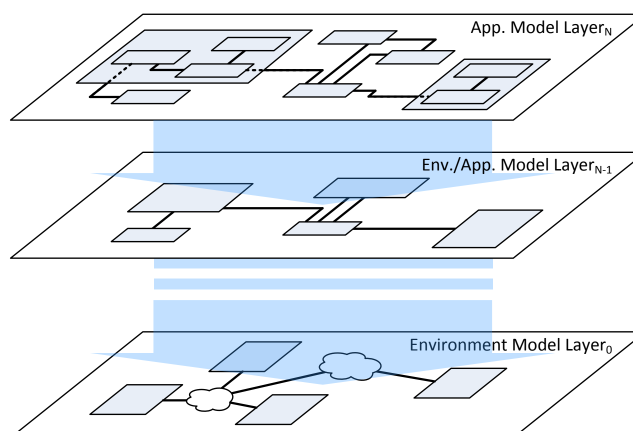

Figure 3.9: The multilayer structure of deployment models in virtualized environments.

As proposed in our previous work [16] it is possible to divide execution environment on several virtualization layers. Table 3.1 presents some of the most common levels and proposes appropriate mapping for selected types of execution nodes and related software components. This relation means that the node at the selected level can host components of the related kind. In other words, the component can be deployed on the corresponding node. The table is by no means complete and may further be extended in case of more sophisticated execution environments. For example, having hosts equipped with Solaris 10 OS we could add the project level just between the operating system and process levels.[11]

Following we present a brief discussion on the factors that need to be explored when designing a deployment model for distributed virtualized environments. More detailed analysis requires a separate extensive research and is out of scope of this work.

recursive execution environment — due to multilayered structure of virtualized systems the environment model may not only describe a flat structure of execution nodes and network connections but needs to be made recursive. In this way it will be possible to model all the nested virtual layers in the environment,

vertical component aggregation — a natural consequence of the recursive

platform heterogeneous environments". However, it does not allow for modelling network environments. Support for multi-platform systems manifests itself in the ability to include in a SDD package different versions of the software, each designated for different platform.

[11]Solaris *projects* group processes into a manageable entities controlled by Resource API. Therefore, an OS administrator is able to control access to resources for all processes included in a project at once.

Table 3.1: An example of mapping of an execution node and software component entities for selected virtualization levels.

Virtualization Level		Execution Node	Component
4	sub-process	component container	CORBA component
3	process	component server	CCM container
2	operating system	OS instance	CCM comp. server
1	OS virt. layer	hypervisor	OS instance
0	computer	host	OS instance

execution environment model is recursion in the application model, too. However, the widely accepted horizontal component aggregation is not enough to address different virtualization layers. It only allows hiding an internal structure of a component behind a well defined high-level component interface. Apart from this, there is a need for expressing *hosting–hosted* relations in order to model *container–component*-like dependencies,

automation of deployment — given the recursive definition of the execution environment and application the question is how to automatically deploy subsequent software layers of this environment. Provided that at an appropriate virtualization level a node can be seen as a component instance (c.f. Tab. 3.1) it seems possible to exploit the deployment platform to perform on-demand deployment on all subsequent levels starting from the lowest level defined,

classification of virtualized nodes — enabling recursion in the environment model, nodes can be nested one in another. Unfortunately, heterogeneity of distributed systems allows that different hosts can have different number of virtualization layers and different layers can host different component types. Consequently, to enable automatic planning, a deployment planner must have means to distinguish between different node types.

The issues presented above do not exhaust the problem of support for virtualization in application deployment but only reveal complexity of the subject. There are many other more detailed issues like resource representation or locality constraints, which we do not discuss here as virtualization is not the main topic of this work. Indeed, support for virtualization in deployment is an interesting research area that is worth closer investigation.

3.2 Support for Adaptation

The basic approach to model-based application deployment assumes off-line spatial planning and static deployment of components over a target execution environment. Our extensions proposed in the previous sections of this chapter extend this basic approach enabling deployment planning in temporal and semantic dimensions. We discussed also deployment in multi-layered virtualized environments. However, in the case of open heterogeneous and distributed systems the discussed extensions to the model are still inadequate. Following is the summary of the main issues that need to be considered as well:

- sharing of resources between different applications or different components of the same application makes the execution environment constantly changing what hampers precise description of the execution nodes and makes static deployment planning fragile,
- requirements of application components often depend on some external variables such as the number of users connected or the amount of input data, which usually cannot be foreseen before and even during application execution,
- high complexity of the deployment planning problem makes exact planning impossible in real case scenarios when the number of components and execution nodes easily exceeds ten.

In result, static deployment over open distributed execution environments is a difficult problem and may easily lead to poor application performance or inefficient resource consumption. To overcome this we enrich the deployment model with elements that enable compositional adaptation of applications. We follow the assumption presented by Maghraoui in [83] that reconfigurable systems enjoy higher application performance because they improve system utilization by allowing more flexible and efficient resource usage.

3.2.1 Requirements for Adaptive Deployment

We based our solution to adaptive deployment of component-based applications on the two paradigms presented earlier in Chap. 2: the architectural reflection and autonomic computing. The reflection leads to separate an application and execution environment from their self-representation and to provide appropriate observation and manipulation interfaces. The autonomic computing enables binding the observation and manipulation tools with the

MAPE control loop that hides low-level deployment tasks behind a high-level management interface.

The plain software deployment build upon the model-based approach provides means to create self-representation of an application and environment. However, to achieve their proper observation, manipulation and autonomic management additional mechanisms are required:

- monitoring of an execution environment that uses the appropriate language to describe the environment and its resources. In the case of heterogeneous systems this is difficult to achieve, although some activities trying to resolve issues in this area are coordinated and standardized by DMTF forum defining the Common Information Model,
- monitoring of running applications that uses the appropriate language to describe an application and its requirements. It is essential for this language to be compatible with the resource description language, otherwise it may disable deployment planning. Moreover, it is desirable for monitoring to be transparent to applications to prevent monitoring aspects being entangled with the application business code,
- redeployment mechanisms that enable application reconfiguration. The availability of these mechanisms to high extent determines flexibility and strength of the adaptation process,
- adaptation algorithms controlling the process of deployment planning by realizing deployment strategies and taking into account user-defined goals.

The following sections discuss these requirements in more details.

3.2.2 Monitoring Facilities

A lot of work has already been done in the area of monitoring and many tools and solutions are ready to use. For this reason, in the course of research we focused on analysis and selection of a technology that would best fit an adaptive deployment in heterogeneous environments. As mentioned earlier, the choice we made was to use the standards proposed by DMTF i.e. CIM that together with WBEM offer a portable solution to monitor and manage distributed systems. CIM defines a conceptual information model for describing computing and business entities in a distributed environment. It attempts to unify and extend the existing instrumentation and management standards like Simple Network Management Protocol (SNMP), Desktop Management Interface (DMI) andCommon Management Information Protocol (CMIP). WBEM, in addition,

aims at unifying the management of enterprise computing environments using a set of standard Internet technologies like HTTP, XML and DTD [33].

The strong side of these standards and implementing tools is in their availability across different operating systems such as MS Windows, Linux and Solaris. In result, we can use the CIM model as a common vocabulary to describe target environment resources irrespective of their hardware and software platform. Using this language we can also express application requirements. The problem that remains unresolved is, however, monitoring of applications. As long as the CIM model is considered the application monitoring is limited to J2EE application only. Therefore, to address the problem of application monitoring additional extensions and development effort are required.

3.2.3 Reconfiguration Mechanisms

The actual set of reconfiguration mechanisms needed for adaptive deployment depends on the way how software redeployment is performed. Figure 3.10 presents the state diagram of the *adaptive deployment* process with highlighted four possible redeployment techniques. This process may be applied to the software application as a whole, to a group of application components or even to each component separately. If the state is changed for the whole application, this implies that all components change their state accordingly. However, the change in a state of a single component does not imply any changes in the overall application state.

In the diagram we highlighted activity paths that are part of application adaptation on different deployment stages.[12] We distinguish four levels of deployment adaptation:

Full redeployment is the most basic form of adaptation. To perform full redeployment a running application has to be deactivated and uninstalled. Then planning can occur that takes into account current execution conditions and an adaptation strategy. Once the new plan is ready, all application components are again installed and activated. This kind of adaptive deployment can be easily performed with either manual or automatic deployment tools. Unfortunately, the need to deactivate and uninstall the whole application is often too restrictive making this common redeployment technique of limited use for adaptive deployment.

[12]When compared to the activity diagrams presented earlier in Figs. 3.2 and 3.3, this diagram contains an additional `Passivate` activity (leading to the `Passivated` state). It is absent in the previous activity diagrams to make them more comprehensible and also because passivation is more important during redeployment than during the deployment process in general.

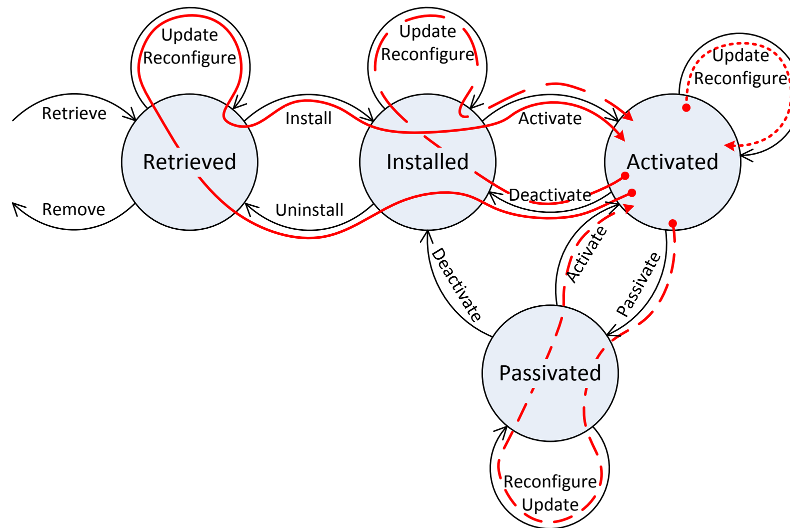

Figure 3.10: The state diagram of the adaptive deployment process with four adaptation techniques highlighted: *full redeployment* (continuous line), *deep redeployment* (long-dashed line), *shallow redeployment* (short-dashed line) and *runtime redeployment* (dotted line).

Deep redeployment is very similar to the full redeployment technique. The improvement stems from the fact that deep redeployment does not perform uninstallation step but only deactivates the application. Once all component instances are destroyed (with or without preserving their state) a new plan is prepared. For the components which location has changed installation takes place, then all components are activated. In this way some `Uninstall` actions are avoided but for the cost of higher resource consumption.
Similarly to the full redeployment, this technique may enable less invasive application reconfiguration if some tools for state preservation are used. Still, however, the need for deactivation poses noticeable interruptions in application execution.

Shallow redeployment is more conservative technique because it does not deactivate an application at all. To perform shallow redeployment the application is suspended preserving its current state. Then required changes in deployment plan are applied and the components that need to be relocated are installed and activated in their new locations.
For this technique, component state preservation is mandatory. Otherwise, activation would result in state inconsistency between relocated and not relocated components. Moreover, shallow redeployment requires a mechanism for component rebinding because all relocated components need to be bound again with their neighbours.

Runtime redeployment is the most seamless adaptation technique. It aims at performing adaptation without even suspending the application. Plan changes are applied in runtime what requires more sophisticated mechanisms than simple state preservation.
In [14, 15] we distinguish two complementary runtime redeployment mechanisms. First, *component migration* combines several lower-level tools like: component passivation, state preservation and links rebinding. All these to provide a mechanism that is able to move particular components from place to place suspending only the minimal part of the application. The main idea behind the second mechanism — *virtual redeployment* — is to make use of system virtualization layer in order to modify the parameters of the execution environment. This allows imposing changes that from the application point of view can be identical to changing components' actual location but do not require any component suspension.

The presented redeployment techniques need that the infrastructure provides some of the following low-level reconfiguration mechanisms:

- state preservation,
- passivation,
- link rebinding,
- runtime migration,
- virtual redeployment.

The key mechanism is the ability to store and load the application runtime state. This may be applied irrespective of the technique used and determines how seamless reconfiguration can be. Provided with a state preservation even full redeployment technique can create illusion of runtime reconfiguration. From the point of view of external clients, this view is somewhat blurred because when contacting deactivated application they receive an error. Nevertheless, from the application point of view, state preservation in full and deep redeployment is visible as runtime reconfiguration, provided that the time scale is not considered.

In order to avoid the problem of application availability and to improve external experience of the reconfiguration process, passivation mechanism can be used. The ability to suspend the application or selected components not only helps to alleviate communication problems but also decreases time needed for the reconfiguration. First, if a component is passivated, the communication infrastructure can hold all incoming requests and issue them

later when it becomes active. Second, it is also much more effective to switch a component state from passivated to activated than to instantiate and bind it after deactivation.

Link rebinding is the mechanisms that is required for shallow and run-time redeployment. When some components change their location while they neighbours are still instantiated, they need to be reconnected. Three rebinding techniques exists: deep update, chain of reference and use of home location agent. A proper combination of the passivation, state preservation and link rebinding makes basis for another important reconfiguration mechanism — runtime component migration. Later in this work we show how migration can effectively support runtime redeployment and also we discuss all the basic mechanisms in more details.

A different way to achieve runtime redeployment is to make use of a system virtualization layer by means of *virtual redeployment*. Unfortunately, virtual redeployment to be effective requires fine-grained isolation of application components. It is ideally when each component is running in a separate manageable container. Then, controlling resources of this container enables the virtual redeployment of the hosted component. Otherwise, the redeployment mechanism operates on a group of components what is not always desirable.

3.2.4 Adaptation Control Loop

Provided with monitoring facilities and reconfiguration mechanisms we now can follow the Autonomic Computing paradigm and focus on the remaining two steps of its MAPE control loop: analyse and planning. In relation to the software deployment the AC analyse step is actually deployment planning and the AC planning step is the way how adaptation manager implements the newly prepared deployment plan. The latter largely depends on the reconfiguration mechanisms available and is rather a technical problem. The former requires much more attention though.

The goal of the analyse step is to reason about monitoring observations that arrive constantly and prepare a new deployment plan. Unfortunately, deployment planning is a NP-hard problem, hence we cannot afford that every iteration of the MAPE control loop would run an exact planning algorithm. Instead, an approximate algorithm is required. The algorithm needs to be responsive enough to follow and properly react to the changes in environment and application. Moreover, due to iterative nature of the adaptation process it is desirable for the algorithm to improve its results iteratively, too. We propose a solution based on force-directed methods that exhibit many useful features in relation to adaptive deployment planning. Later in this work we

discuss a family of Force-Directed Algorithms and present our approach to the runtime deployment planning problem.

3.3 Summary

This chapter has presented the main elements that are part of our concept of adaptive deployment framework. The discussion has been grounded in the model-based approach to deployment which enables fully automatic planning and hence allows for adaptive application deployment. Model-based deployment assumes component design of software what very well fits with our adaptive deployment framework. Separation between a software component and its execution context facilitates achieving transparency of the framework. Much of the infrastructure can be embedded in the components' context leaving business code untouched.

We have pointed at the need for standard resource and requirement description language and propose to use the CIM model. It can be used in static description of environment resources and application requirements as well as in dynamic resource monitoring. Another important issue is to provide reconfiguration mechanisms that enable deployment adaptation. We have presented four levels of redeployment and showed how state preservation, component passivation, and other basic mechanisms are useful to achieve them. Finally, the issue of control loop for adaptive planning has been briefly introduced while a more in depth discussion is presented further in this work.

Apart from adaptation in deployment, we have also shortly presented two interesting research areas closely related to model-based deployment: planning in temporal and semantic dimensions, and influence of system virtualization on a deployment model. Although they are not the main focus of this work, both these aspects are important when deployment in heterogeneous distributed systems is considered. We believe that with growing interest in application deployment also these issues will be addressed by future deployment models and solutions.

Chapter 4

Adaptive Deployment Framework

In the previous chapters we outlined that adaptation is the only choice to perform effective software deployment in open distributed computer systems. Due to variability of execution environment resources and variability of application requirements the static approach to deployment is inadequate. To build an adaptive deployment framework for such systems, however, a number of issues needs to be considered. We discussed them in the previous chapter, whereas further in this work we describe a framework that implements selected aspects of the presented concept.

The framework implementation has been divided on the following three phases identified in the course of realization:

- building a plain deployment infrastructure that implements the model-based deployment,[1]
- devising planning algorithms and creating a deployment planner,
- building an adaptation infrastructure comprising of monitoring, reconfiguration and runtime planning mechanisms.

Each of these tasks is in itself a complex and challenging undertaking and it seems hardly possible to create a comprehensive framework that provides all of them in an exhaustive manner. Therefore, our implementation of the framework is a proof of concept showing that the proposed solution can be realized and successfully used for deployment adaptation. One of the important limitations of the framework is support for spatial deployment

[1] Unfortunately, OpenCCM — the component platform we used to implement the framework does not provide this infrastructure.

planning only. The problem of deployment planning in all three dimensions is, however, difficult and would require a separate extensive research.

Before discussing the details of the framework implementation we briefly present the technologies selected to realize the platform what had significant influence on its implementation.

4.1 The Model of Deployable Components

From the variety of available software component technologies for distributed systems (such as CCM, EJB, Fractal, GCM, SCA) we selected the CORBA Component Model (CCM) proposed by OMG in [103]. This technology, although not attained popularity, offers many strong and valuable features that are important for distributed component-based software systems in general and deployment frameworks in particular. CCM inherently realizes such design patterns as dependency injection,[2] late binding, two-phase initialization, event-driven programming, clear separation between a component and its execution context and many others. Actually, the CCM combined with PSM for CCM defined in the D&C specification is, in our opinion, one of the most advanced component models designed to support communication, software development and deployment in heterogeneous, distributed systems.

The basis for the model is a software component entity (Fig. 4.1) that defines its behaviour with provided and required ports. Ports enable connecting that can be thought of as a “bus” with multiple ports putting data on the bus, and multiple ports taking data off the bus [99]. Although currently ports are a common feature of many other component technologies, what distinguishes CCM is that they serve for three basic and disparate types of communication. Facets and receptacles are used for synchronous operational invocations, event publishers and consumers are used for asynchronous event communication. Lastly, stream sources and sinks are used for transferring continuous data streams. We believe that this particular characteristic of the CCM model makes it very attractive for both application designers and developers. Unfortunately, tight relation to complex and declining CORBA technology makes the model of less impact on software technology today.

A number of work has been devoted to the CCM technology [89, 125, 126] and deployment of CCM-based applications [6, 31, 74]. Therefore, we limit our discussion to point out only the most important characteristics that decided on our choice of CCM as the technology for realization of adaptation framework:

[2] Martin Fowler in [50] discusses inversion of control and its special case — dependency injection.

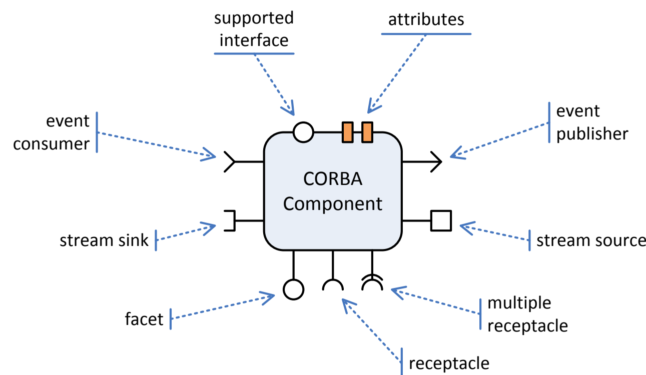

Figure 4.1: Illustration of the general concept of a component in the CCM model.

- CCM defines a general and "strong" components model for distributed systems what makes implementing the adaptive deployment framework an interesting and challenging task,
- the CCM model offers communication transparency and potential diversity of implementation technologies. This is important when realizing deployment in heterogeneous environments as most of them already support the CORBA standard,
- the D&C specification provides for the CCM model a transformation from the Platform Independent Model to the Platform Specific Model what facilitates implementation,
- there exist several open source platforms implementing the CCM model what allowed us to extend the model and incorporate the extensions into a selected platform.

The choice of the CCM technology for realization of our deployment framework means that further in this work whenever we use the term *component*, it refers to a CORBA component unless otherwise stated.

4.2 Overview of the Framework

The overall architecture of our adaptive deployment framework derives much from the models defined in the D&C specification. D&C proposes the structure of a plain deployment platform and based on this we added support for adaptive software deployment. As discussed in the previous chapter, the framework performs deployment at two abstraction levels: globally and

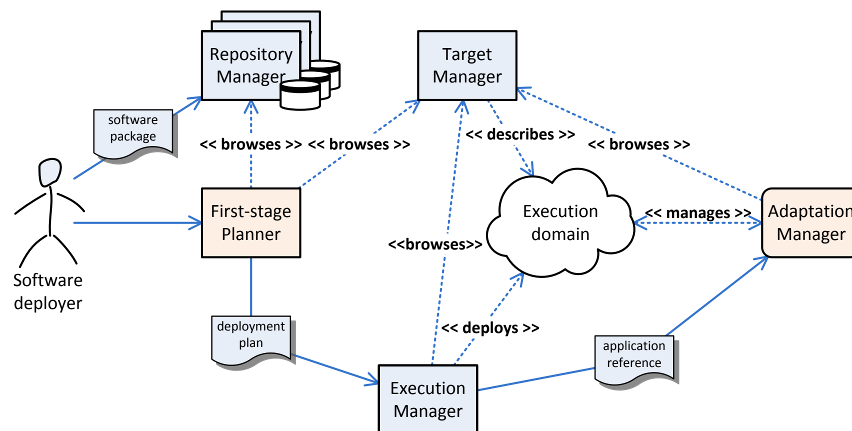

Figure 4.2: The global view on the architecture of our deployment framework. The highlighted elements represent extensions to the original D&C model.

locally. Global deployment tasks encompass planning, connecting and adaptation of component deployment over the whole execution environment, whereas local deployment includes artifacts retrieval, components instantiation, local binding and management functions.

The global view on the architecture is presented in Fig. 4.2. The highlighted elements represent extensions to the original D&C plain deployment infrastructure. The first-stage planner performs simple plan searching using information provided by the `TargetManager` and `RepositoryManagers`. The former describes all elements of the target execution environment, whereas the latter offer description of the software packages being deployed. The `AdaptationManager`, provided with a reference to the running application instance and domain information acquired from the `TargetManager`, performs appropriate deployment adaptation. Functionality of the remaining elements in the global view is defined by the D&C specification [100, Chap. 7]. `RepositoryManagers` maintain and manage component package data. The packages can be installed directly in a software repository or by reference that points to some external locations. Repository managers can also provide a list of all installed packages. The `TargetManager` is the central point that collects and provides all the information about the execution domain and tracks resource usage within the domain. Finally, the `ExecutionManager` is responsible for execution of a deployment plan. It delegates deployment actions down to the local level and binds together component instances running on different nodes. For resource management the `ExecutionManager` is also associated with the `TargetManager`.

Locally, the main role in software deployment plays the `NodeManager` element which receives from the `ExecutionManager` the part of the whole

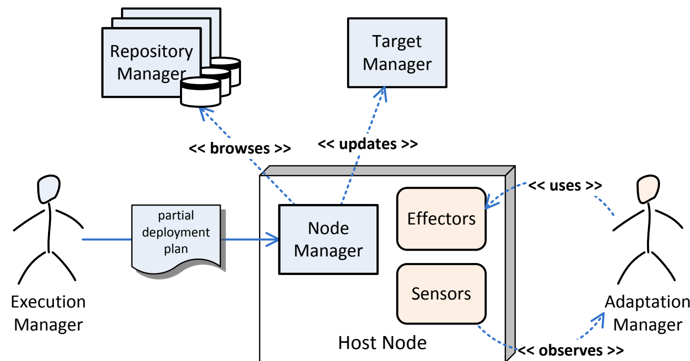

Figure 4.3: The local view on the architecture of our deployment framework. The highlighted elements represent extensions to the original D&C model.

deployment plan that has been assigned to the managed node (Fig. 4.3). The `NodeManager` is responsible for retrieving all the software artifacts relevant to the subset of components and for preparing the node to instantiate them. The preparation includes e.g. running and configuring component servers and containers. After preparation and instantiation of components, the manager configures them and performs their local binding. The `AdaptationManager`, presented also in this view, indicates that selected sensors and effectors of the adaptation infrastructure are present on every node in the execution environment. To avoid unnecessary overhead related to contacting the `TargetManager`, communication between sensors and effectors and the `AdaptationManager` is direct. We make use of the event distribution mechanism offered by CCM, which is very suitable for this purpose.

Further in this chapter we present more details of the implemented framework discussing the plain and adaptive deployment infrastructure.

4.3 Plain Deployment Infrastructure

In the course of implementation of the plain deployment infrastructure we realized most of the requirements imposed by the D&C specification and the “Deployment PSM for CCM” chapter [103, Chap. 14]. To minimize redundancy of information with the specification, we focus in this section mainly on these implementation aspects and decisions that were interesting, unspecified or related to our extensions to the model.

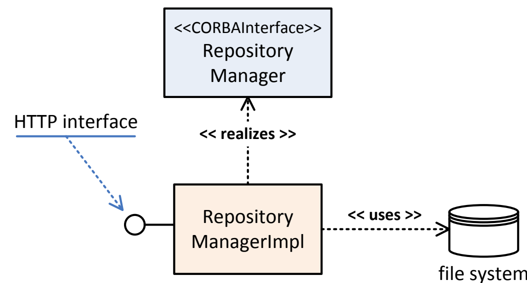

Figure 4.4: Implementation of the RepositoryManager element.

4.3.1 Repository Manager

The RepositoryManager is a simple element that manages component data. Our implementation provides two access interfaces: the RepositoryManager CORBA interface and additional HTTP interface (Fig. 4.4). The former, as defined in the D&C specification, enables management of the repository. The latter allows downloading component artifact files using URL references. This is convenient because URLs are human-readable and can be easily included in XML-based application description files. To store all files related to a software package, the manager uses a local file system.

4.3.2 Target Manager

The TargetManager is responsible for two tasks: providing information about the target domain and tracking resource usage within the domain. Our implementation of the manager supports only the former functionality. The domain data is read from XML files and then is accessible using the TargetManager CORBA interface defined in D&C. More advanced implementation of the manager would perform dynamic domain discovery. This is, however, a complex and rather technical task, hence we decided not to realize it.

We also did not implement tracking of resource usage. We took a different approach when realizing the deployment framework. To be useful, a resource tracking mechanism must have exclusive rights to allocate and deallocate resources what is in stark contrast to openness of the execution environment which we assumed. Therefore, we adopted the best-effort approach to resource management, which is much more effective than strict resource reservation and management.[3] The main idea behind our adaptation framework is to let all applications share the common execution environment. Then, our infrastructure supplied with data from environment observation can perform adaptation of a selected application.

[3]This problem was already discussed in Sect. 1.2.

4.3.3 First-stage Planner

The key element of a model-based deployment infrastructure is a planner, which process package information and description of an execution environment to produce a deployment plan. In our setting the planner is supposed to work in an open and dynamic distributed system. It means that changes in the execution environment and application may be caused by some uncontrollable, external factors. For this reason, an important characteristic of the planner is high responsiveness. The more immediate response from the planner is, the higher chances are that input information used for planning is consistent with the most recent state of the domain and application during plan execution.

As discussed earlier in this work, deployment planning is a NP-hard problem. Therefore, to tackle this complex task effectively we distinguished two types of deployment planners:

- an initial, first-stage planner, and
- an runtime, adaptation planner.

The main idea behind the first-stage planner is to provide an initial deployment plan with respect to mainly static information i.e. static resource properties and component requirements. Further, we assume that this initial plan will be improved in runtime taking into account dynamic and volatile information coming from the monitoring infrastructure. In this way we avoid the need for accurate information about resource consumption and pessimistic resource allocation. The adaptation planner is specifically designed to analyse dynamic system information. The main difference between these two planners lays in the level of correctness. While adaptation planner works on dynamic and often volatile data and is by definition less accurate, the first-stage planner strictly matches all static information from component requirements against available resources.

The Basic Approach to Planning

The basis for the initial, first-stage planner that we implemented is Best-First Search (BFS) algorithm with the first-fit heuristic. The main difference between the proposed algorithm and well known bin packing algorithms (first-fit, next-fit, etc.) stems from the limited number of nodes the deployment algorithm operates on. Searching for the solution, BFS may need sometimes to backtrack and change a previously selected node while a bin packing algorithm would simply open a new bin.

Table 4.1: Preferred BFS heuristics for initial deployment planning.

layout	larger domains	smaller domains
disperse	next-fit	worst-fit
compact	first-fit	best-fit

The first-fit function starts placing components beginning from the first node considered. If a component does not fit, a next node is visited until all nodes are scanned through (planning failure) or all components are assigned (planning success). Apart from first-fit, BFS can use other similar heuristics such as next-fit, best-fit or worst-fit. The next-fit differs from first-fit in that it starts placing components on consecutive nodes instead of starting from the first one selected. The other two heuristics require a bit more processing. For each component being placed they search a node with the most amount of free resources in case of the worst-fit, or the least amount of free resources in case of the best-fit function. Selection of a heuristic allows users to influence the initial component deployment to some extent. The next-fit and worst-fit tend to distribute components over nodes more evenly, whereas first-fit and best-fit compact them on a much lower number of nodes.

From the user point of view an interesting concern is also time complexity of the search algorithm. The next- and first-fit heuristics are less complex than worst- and best-fit by a factor of $O(n)$, where n is the number of execution nodes. This is because for each step the latter methods need to inspect all nodes searching for the worst/best component matching. Therefore, for large execution domains and large number of application components preferred are the next- and first-fit functions, whereas for smaller domains the difference in time complexity is not substantial and the worst- and best-fit heuristics should be used instead as they provide better results. Table 4.1 shows a brief summary of this comparison.

More Detailed Planning Requirements

Apart from these general aspects, there exists a number of additional issues, specific to D&C and CCM, which are also very important for deployment planning:

- a software package may contain a number of equivalent component implementations that may have significantly different requirements and capabilities,
- components are assembled together using connections between ports.

The connections may also have requirements against `Interconnect` and `Bridge` domain elements,

- both component and connection requirements may be of many different types and against many different resources,
- planning algorithms need to satisfy locality constraints.

Given the choice of component implementations available, the BFS algorithm may need to backtrack if a plan cannot be found in the first pass. Due to multiple distinct resources and multiple distinct capabilities it is, in general, a multi-objective problem to decide which implementation should be considered first and which left for later examination. To provide a simple solution for this we treat the set of available component implementations included in a `ComponentPackageDescription` as an ordered list that entails user preferences. The farther in the list an implementation is the less preferred it is considered.

Secondly, requirements of component connections are the aspect defined in the D&C models that makes it very valuable and distinguishes this specification form other deployment standards. Support for connection requirements, however, increases complexity of planning problem by a factor of $O(n)$, where n is a number of application components. This is because for each considered component all its connections with other components need to be verified.

The next issue, different resource types, again increases complexity of the planner algorithm. Usually, however, the number of requirements and resources is much smaller than the number of nodes and components, hence this factor does not significantly influence overall time complexity of the searching. It is also worth noting that different resource types can be used as another control parameter in deployment planning. Having an assembly of component instances, a heuristic can first consider the components earlier in the assembly instance list. However, provided with a priority of resource types (e.g. that memory size is more important than CPU utilization), it can pick the component with the highest/lowest resource demand. Taking both, the need for matching requirement against resources and prioritized selection of components from the assembly, increases complexity of the searching algorithm only by the factor of $O(n \log n)$, where n is the number of component requirements.

Locality constraints, the last of the aforementioned problems, may cause much more trouble when planning of deployment is considered. Imposing locality constraints on components can be modelled as the graph-colouring problem which is known to be NP-hard in general case [73]. Following we provide a mapping of the locality constraints problem on graph-colouring.[4]

[4]It is based on the mapping for job scheduling presented by Marx in [87].

> Let G be the conflict graph of the application components. Vertices of G correspond to components, whereas edges correspond to locality constraints. Two components are connected with an edge if they cannot be executed on the same execution node. The colours correspond to hosts in the execution domain. Resolution of the locality constraints problem is actually the problem of finding k-colouring of the conflict graph G, where k is the number of hosts.

Fortunately, when deployment planning is considered the number of hosts is usually much higher than the maximum degree of the conflict graph. Therefore, despite that k-colouring is NP-hard, in such cases it can be solved relatively easy e.g. using a greedy colouring algorithms [55].

Due to extensive development effort required to implement all of these features we decided to keep low complexity of the algorithm and to simplify the first-stage planner code. Our solution does not address the problem of locality constraints and connection requirements.

4.3.4 Deployment Plan Execution

The deployment approach proposed by OMG in the D&C specification realizes the *plain deployment* process.[5] As depicted in Fig. 4.5, the deployment steps are performed using three operations (`preparePlan`, `startLaunch` and `finishLaunch`) and a group of dedicated interfaces.

From a user point of view the main entry point to the deployment infrastructure is `ExecutionManager`. Provided with a deployment plan it begins the whole process of application deployment. First, it enables a user to install components on the nodes assigned in the plan, which means it realizes the `Prepare` task that is a part of the `Install` activity (path S1.1–S1.4). In return, the user receives a reference to a `DomainApplicationManager` object used further to start the activation process. By invoking `startLaunch` operation the activation is initiated i.e. the `Instantiate` deployment step is performed (path S2.1–S2.4). This results in a reference to a `DomainApplication` object. Finally, the user can perform the `Bind` step and finish the activation process using the `finishLaunch` operation (path S3.1–S3.2a). After this call the application is up and running.

Our implementation of the `ExecutionManager`, `NodeManager` and all other related interfaces follows the general recommendations included in D&C, hence we do not discuss it in more details. What is more interesting, however, is added support for application reconfiguration. As dynamic

[5] In contrast to *adaptive deployment*, as defined earlier in Sect. 2.1.

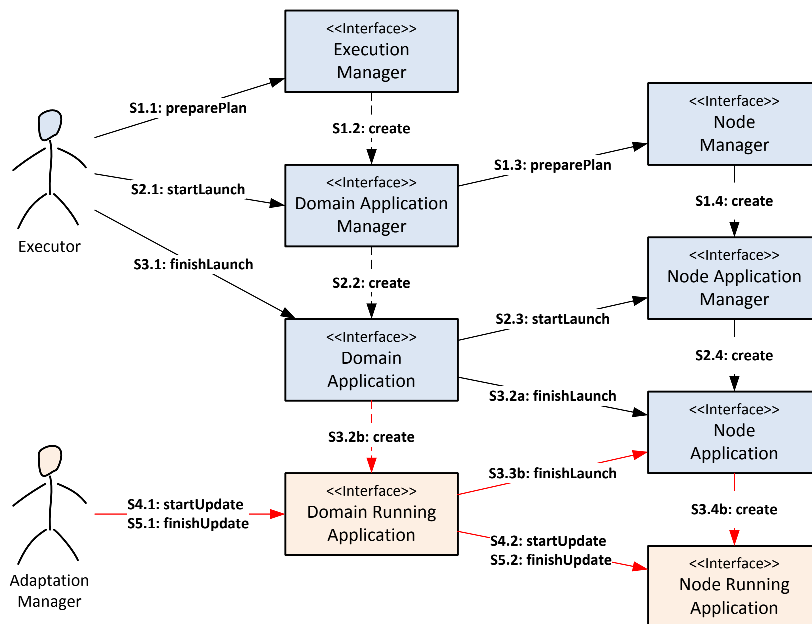

Figure 4.5: A collaboration diagram showing deployment plan execution. The highlighted entities represent our extensions to the original D&C specification that enable runtime deployment adaptation.

aspects of application deployment are beyond the scope of the current D&C specification,[6] to support adaptiveness in deployment some advancements to the model were inevitable. We extended the presented scenario such that invoking the `finishLaunch` operation returns a reference to a new model entity — a `DomainRunningApplication` object (path S3.1–S3.4b). This, in turn, allows a user and adaptation manager to perform deployment adaptation by means of the `startUpdate` and `finishUpdate` operations.

As shown in Fig. 4.6, the `Domain` and `NodeRunningApplication` are interfaces derived from the `RunningApplication` interface. They follow the approach adopted by the OMG specification and realizes two-phase initialization pattern. The `startUpdate` operation corresponds to `startLaunch` from the `ApplicationManager` interface and initiates the update by making deployment plan changes and possibly by creating/destroying component instances. The `finishUpdate` operation corresponds to `finishLaunch` from the `Application` interface and must be called to commit the update what may also involve completing components' configuration, interconnecting and activating them.

[6] This was officially confirmed in the resolution to the Issue #7746 available on the mailing list of the Deployment Revision Task Force at `http://www.omg.org/issues/deployment-rtf.html`

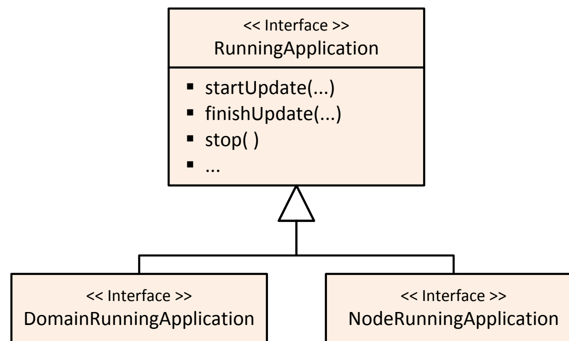

Figure 4.6: The interface enabling adaptation of application deployment.

An important detail worth mentioning here is the way how an application update may be performed. We distinguish two such methods: internal and external. If the update is requested to be done internally, it means that the implementation of the `RunningApplication` interface is responsible for realizing all the actions involved in the update. Depending on the implementation it may result in realizing any of the redeployment techniques presented in the previous chapter, namely *full*, *deep*, *shallow* or *runtime redeployment*. The other, external reconfiguration method, requests `RunningApplication` to make changes only in the deployment plan. All the actions required to perform the actual update are done externally to the deployment infrastructure. In case when the external update completes successfully, the `finishUpdate` operation allows for committing plan changes. It is the responsibility of an external reconfiguration mechanism to ensure consistency between the updated deployment plan and the real state of the application deployment.

In our adaptation framework we use both of these update methods. Internal updates are performed whenever we need to deploy the hosting infrastructure in a new location in the execution domain. It includes retrieving, installing and activating a component server, container and factory. The other, external method is used to redeploy a component instance between two locations in runtime. In this case we use a component migration mechanism that performs all needed reconfiguration actions externally to the deployment infrastructure. What internal update does in our implementation is simply the full component (re)deployment. Conversely, the external update follows the runtime redeployment technique by changing only the application deployment plan while more advanced reconfiguration steps are done by a dedicated external effector. Separation between these two kinds of deployment updates facilitates framework extensibility. While internal updates are fixed for a given deployment infrastructure, the external kind allows for testing and development of new methods of reconfiguration techniques without any changes in the deployment infrastructure.

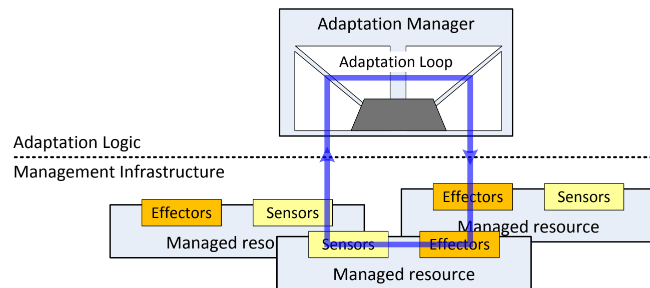

Figure 4.7: Two layers of the adaptive deployment framework.

4.4 Adaptive Deployment Infrastructure

When designing the adaptive framework for software deployment we followed the Autonomic Computing paradigm proposed by IBM. As shown in Fig. 4.7, the architecture of the infrastructure may be divided onto two layers: (1) the adaptation logic layer and (2) the management infrastructure layer. Between these layers there are provided well defined sensor and effector interfaces what facilitates replacing the adaptation logic as well as extending the infrastructure layer.

An important assumption we made when designing the adaptation infrastructure is that we consider deployment adaptation of a single application hosted in a shared distributed execution environment. This decision makes design and implementation of the adaptation logic easier yet does not limit flexibility of the framework. Following the principles of separation of concerns, Autonomic Computing proposes that a hierarchy of managers can be created. Then, a higher-level manager could coordinate lower-level managers each of which managing a single dedicated application. In result this hierarchical approach enables management of many user applications in a shared environment.

The architecture of the management infrastructure has also been split on two distinct parts. One is responsible for monitoring and management of the execution environment and may be shared by many adaptation managers, whereas the other is responsible for component monitoring and management and is dedicated to a particular application.[7] Figure 4.8 shows a more detailed model of our adaptive deployment framework. The manager is supplied with monitoring information delivered from the environment (E_S sensor channel) and application (A_S sensor channel). The collected data is used to make decision on how to reconfigure the application (A_E effector channel) and/or the environment (E_E effector channel). The model creates

[7]When system virtualization is considered, these two parts should further be divided onto more layers each of which responsible for a single virtualization layer.

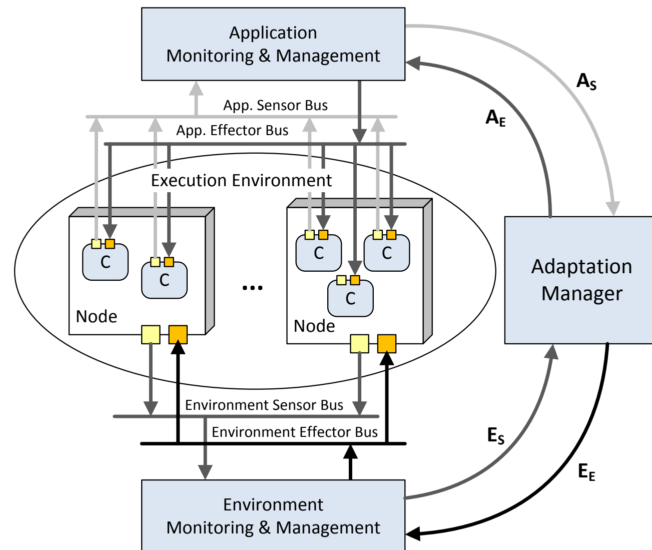

Figure 4.8: The model of adaptive deployment framework showing separation between application and environment layers. It creates four adaptation control loops: $A_S - A_E$, $A_S - E_E$, $E_S - E_E$ and $E_S - A_E$.

four potential adaptation control loops that `AdaptationManager` can utilize to control deployment. The existing prototype implementation of the framework makes use of two of them $A_S - A_E$ and $E_S - E_E$, however, in our previous work some experimentation was done to exploit $A_S - E_E$ and $E_S - E_E$ loops, too. We present this approach in the next section when discussing about the virtual redeployment effector.

4.4.1 The Management Layer — Sensors and Effectors

The key elements of the management layer are sensor and effector elements. We decided to build all the sensors and effectors in our framework as separate CORBA components that follow the same general design presented in Fig. 4.9. Three main factors influenced this decision:

- the component-based approach very well fits to the nature of sensors and effectors which present a separate, well defined functionality,
- apart from the single sensor/effector interface a CCM component can have multiple ports and attributes. This is very convenient in the case of sensors which offer two ways of accessing monitoring data: by polling the `resourceSensor` provider port (pull model) or by notification through the `resourceUpdate` event port (push model),

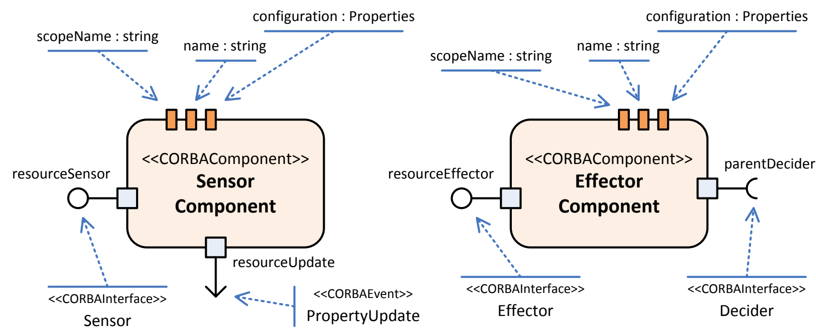

Figure 4.9: The general design of the sensor and effector components.

- using the available deployment infrastructure, sensor and effector components can be easily deployed and undeployed depending on the adaptation needs. This allows decreasing overheads related to unnecessary monitoring and management of the environment.

The `Sensor` and `Effector` interfaces are partially based on the *Properties* design pattern commonly used by e.g. CORBA Property Service [96] and Java Management eXtension (JMX) technology.[8] Apart from the basic ability to get and set property values, the interfaces provide reflective means to discover what properties or operations are offered by a managed element. What distinguishes the `Sensor` and `Effector` interfaces from the Properties pattern, however, is that they include operations to get and set extended properties. The extended properties are these which can be get or set providing an additional contextual information. This enables us to cover with one sensor a set of similar managed entities. For example, to monitor intensity of communication between many components it would be highly inefficient to attach a separate sensing element to each communication link. Conversely, using the extended properties one sensor may monitor many different component links and provide that information by means of an extended property. A client is then able to get the monitoring data by specifying the property name together with a link identification.

An important detail of the presented sensor and effector components is the `configuration` attribute. It allows accessing properties of the component itself and, in this way, enables controlling how the component works. A good illustration of a configuration property is the `UpdateInterval` property of sensors that determines the rate of generated resource update events.

In the course of the framework development we implemented the following sensors and effectors:

[8]Full IDL specification of these interfaces is included in Appendix A.

- CIM-based CPU and memory sensors,
- Container Portable Interceptor (COPI)-based link and communication sensors,
- instance sensor, and
- component migration effector.

It is worth noting, however, that the sensor and effector components are only a facade for internal mechanisms required to realize their functionality. Following we shortly characterize the component-based part of the sensors and effector, whereas their internal structure and embedding in the execution environment is discussed in the next chapter.

CIM-based Sensors

Common Information Model is an important model when monitoring and management of distributed systems is considered. It defines representation for most of the vital dynamic parameters of operating and computer systems. Therefore, CIM-based sensors can provide this information for our Adaptive Deployment Framework. To monitor an execution environment we used only a small part of the whole model proposed by DMTF and use only small number of CIM classes that describe computer's CPU, operating system, and file system. Table 4.2 shows the names of implemented sensor components together with the CIM classes they use and the selection of the monitored properties.

What is useful when working with the CIM model is that it defines properties together with units of measurement e.g. the OS `FreePhysicalMemory` property describes the number of kilobytes of physical memory currently unused and available. Often, such a definition is enough to obtain accurate measurement independent of the OS platform. An exception is however the CPU `LoadPercentage` property that describes processor load, averaged over the last minute, in percent. This value depends on processor type and OS scheduler algorithm and thus can hardly be compared between different CPUs and OSes.[9]

The implemented CIM-based sensors are gateways between WBEM and CCM technologies. This is in line with the Autonomic Computing approach which recommends that Autonomic Manager use only the well defined sensor and effector interfaces covering different monitoring and management technologies with a selected common standard technology.

[9]We noticed a considerable difference between CPU load reports on two PCs one with Linux and one with MS Windows operating systems.

Table 4.2: CIM-based sensors provided by the framework.

Sensor name	CIM class	Properties used
CPUSensor	CIM_Processor	LoadPercentage CurrentClockSpeed MaxClockSpeed
MemorySensor	CIM_OperatingSystem	FreePhysicalMemory FreeVirtualMemory TotalVirtualMemorySize TotalVisibleMemorySize

COPI-based Sensors

The purpose of these sensors is to measure basic properties of communication links between components. Link sensor assesses communication intensity of all ports connecting selected two components and provides information such as the number of operation calls in a time unit, the size of data transferred in a time unit, Round Trip Time (RTT). Communication sensor measures the general communication intensity of a component in relation to its maximum intensity over a selected period of time. This sensor informs about average number of operations and average amount of data sent and received by all ports of a component.

In order to monitor network communication, two common approaches are possible: *active* and *passive*. The active monitoring relies on injecting test packets into the network and then measuring Quality of Service obtained from the network. The passive approach, on the contrary, uses hardware and/or software tools to observe the traffic as it passes by [1, 25]. To estimate the communication link parameters we followed the less invasive, passive approach. Our implementation makes use of the native OpenCCM interception infrastructure and COPIs which we implemented in the course of this work.

Using these two mechanisms we can attach an interceptor to any of the component's ports and observe every communication attempt taking place on the port being intercepted.[10] When collecting raw monitoring data, the sensor performs its initial processing (i.e. estimation of the desired link parameters) and then makes this higher-level information available through its ports.

[10]The interception mechanism is not available for stream ports because the two specifications: Streams4CCM [99] and COPI [101] has not yet been properly integrated.

Instance Sensor

The instance sensor allows observing creation and destruction of component instances in runtime. It can be used to support deployment mechanisms such as the execution of temporal plans and the runtime component migration. Firstly, notifications from the sensor enable a plan executor to monitor appearance and disappearance of component instances and follow a DAG denoting temporal dependencies. By observation of component creation events, it is possible to impose `StartToStart` temporal collocations, whereas destruction events allow applying `FinishToStart` and `FinishToFinish` collocations. Secondly, ability to detect creation and destruction of component instances facilitates error detection during component migration. Provided with a confirmation if a new component's incarnation appeared at the target location and the old one disappeared from the source location, we can monitor correctness of a component move and throw an exception when necessary.

The CCM model provides two common methods for creating and destroying component instances. One, uses a standard component's factory defined by the model i.e. the `CCMHome` interface that is means to manage instances of a specified component type. The other, makes use of component's entry points and a deployment infrastructure. Originally, the second method appeared only in the D&C specification but later it was incorporated into the CCM model, too.

`InstanceSensor` provides a consistent method for observation when instances are created and destroyed irrespective of the responsible entity. It makes use of the factory listener mechanism included in the OpenCCM platform that allows attaching a listener at three different levels: home, container or component server. For example, attaching a factory listener to a component server enables our sensor to monitor every attempt of instance creation or destruction that takes place in any factory and any container running in this component server.

Migration Effector

The basis for the migration effector was the Component Migration Service (CMS) developed in our previous work [14]. However, for the purpose of the adaptive deployment framework instead of using the specific CMS interface we exposed component migration facility through the generic `Effector` interface. This enables connecting with the `AdaptationManager` directly.

`MigrationEffector` accepts only one operation which moves a component from its current location to a selected destination location. Usually, the movement follows the three stages presented in Fig. 4.10: (1) freezing the

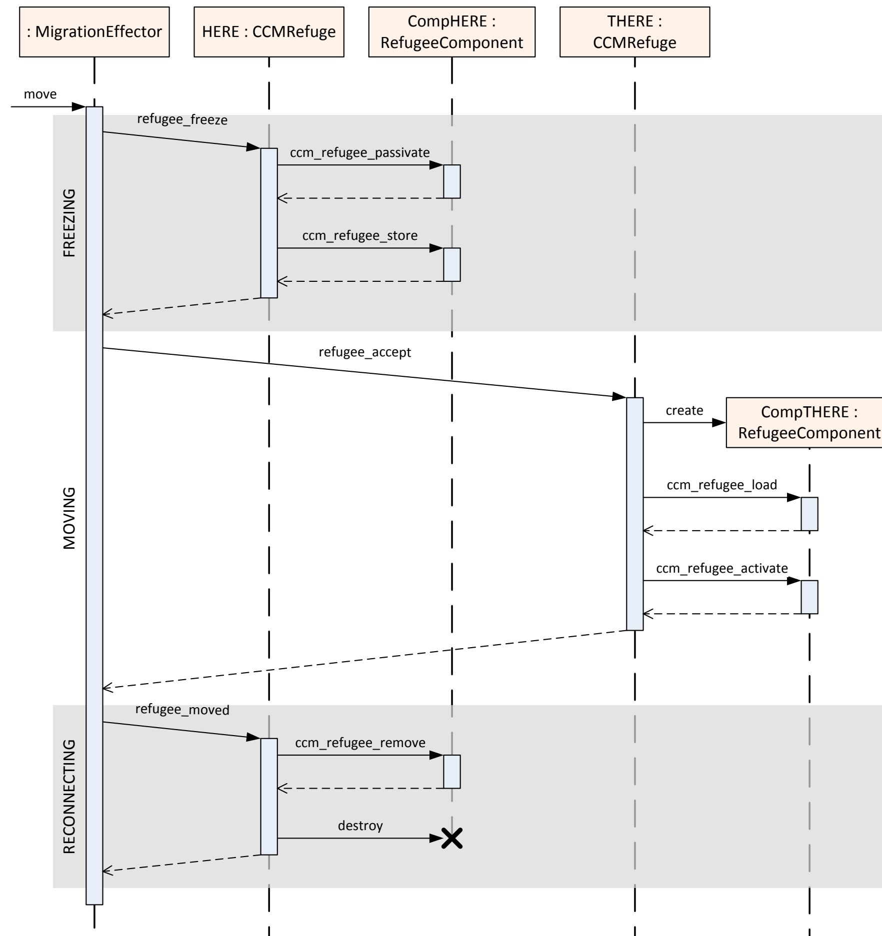

Figure 4.10: A sequence diagram of successful migration between HERE and THERE locations.

state of a component, (2) moving the component to a destination location, and (3) reconnecting the component.

As shown in the figure, to move a component the migration effector operates on component's factory (the `CCMRefuge` object) which is an extended version of the standard `CCMHome` interface. This makes an implicit requirement for the destination location to be prepared to host the component infrastructure. Before a migration can occur, the destination must run a `CCMRefuge` factory, container and component server. Moreover, all these infrastructure elements need to be configured according to the original deployment plan used to instantiate the migrating component. This clearly shows how component migration is dependent on the deployment infrastructure. In our solution the `AdaptationManager` component is in charge of coordination between the migration and deployment tasks.

Virtual Redeployment Effector

Runtime component migration is a reconfiguration mechanisms that operates at the application layer. A complementary mechanism, operating at the execution environment layer, is *virtual redeployment*. This technique is available when the deployment process is performed over a virtualized execution environment. Then, instead of changing components' location what happens during migration, it is possible to adjust properties of the environment. For example, increasing memory resources allocated to a node hosting a component (e.g. an OS container) may have the same result as moving the component to a node with larger amount of memory.

In our previous work [15] we showed that this kind of adaptation may be performed in an effective and transparent way. The transparency means that from the applications' point of view virtual redeployment is entirely invisible. Moreover, given a virtualization management layer provided with an operating system or virtual machines such as VMware or Xen, virtual redeployment is far less complicated to implement than the runtime component migration. To be done it does not need any modifications to the component execution environment but merely an access to a virtualization control layer.

The easiness of how virtual redeployment may be incorporated into an existing component platforms is a major advantage of this technique. The main drawback is, however, the required level of node granularity. Considering system virtualization, the finest manageable unit is a process.[11] Usually, a single process encapsulates a component server which runs several containers, and a container rarely hosts only a single component (Fig. 4.11). This makes 1-to-N relation between a hosting process and a running component what constraints the virtual redeployment technique to operate on groups of collocated components rather than on particular instances. The relation is even worse when lower virtualization layers are considered such as OS containers and operating system virtualization.

Consequently, the virtual redeployment technique is an attractive mechanisms that may support deployment adaptation but is not enough to achieve flexible application reconfiguration. Moreover, to make maximum use of the virtual redeployment, the adaptive deployment infrastructure would require support for virtualization. This is, however, out of scope of this work, hence we focus merely on runtime component migration as the more flexible reconfiguration approach.

[11]Some attempts to create a subprocess management units are Java Isolation together with Resource Consumption Management APIs [27, 28, 105] and .Net application domains. While the future of the isolates is unclear, the application domains focus mainly on isolation and do not offer any resource management functionality.

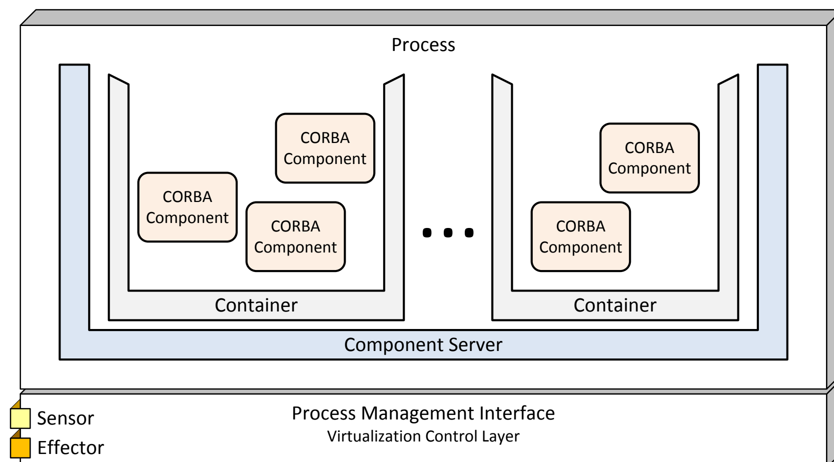

Figure 4.11: The relation between a CCM component, container, component server and a hosting process.

4.4.2 The Adaptation Layer

The key element of the adaptation layer in our deployment framework is the `AdaptationManager` component. It is realized according to the AC paradigm and implements the Autonomic Manager (AM) element. As mentioned earlier, one of the most important assumptions we adopted when designing the manager was that it is responsible for management and adaptation of exactly one user application. This decision does not limit flexibility of the framework but proposes to distinguish high-level Autonomic Managers that coordinate operation of multiple lower-level managers.

In this work we focus on designing the lower-level AM only. Figure 4.12 presents a general view on the `AdaptationManager` component. The `Decider` interface enables linking the manager with higher-level managers (through the `parentDecider` uses port) and lower-level managers and other AC elements (using the `decisionPoint` provides port). Our prototype implementation does not use these ports, though. Attribute `appInstance` allows setting the application instance that the manager is in charge of. Other ports are used to communicate with sensors and effectors.

As discussed earlier in this work, the main building block of the AM element in general and `AdaptationManager` in particular is the MAPE control loop. Using CCM technology and the proposed design of this component the first and last elements of the loop, namely *monitor* and *execute*, are naturally available within the CCM middleware layer. The manager can receive events and invoke required operations using plain CCM communication mechanisms. The real challenge when developing AM is, however, to implement the *analyse* and *plan* stages. In the case of application deployment the AC analyse step

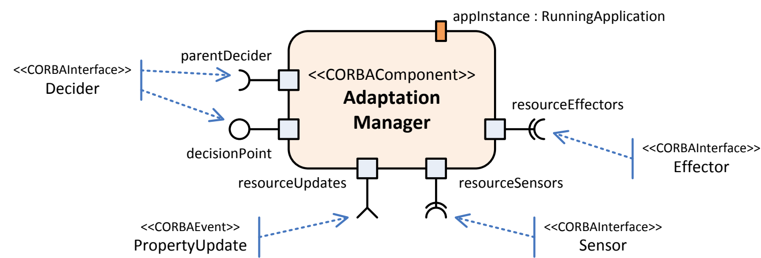

Figure 4.12: The general design of the `AdaptationManager` component.

refers to adaptive deployment planning which analyses the current state of a running system and searches for a better application deployment plan. The MAPE planning is the way how the elaborated deployment plan shall be effected in the target execution environment. Following is a more detailed description of how we implemented the MAPE control loop in our `AdaptationManager`.

Monitor and Analyse

Adaptation is based on monitoring of the application and its execution environment. To perform monitoring tasks, the `AdaptationManager` needs to run and connect appropriate sensors. The component-based design of the monitoring infrastructure brings one important benefit — it allows attaching and detaching the monitoring sensors on demand. The `AdaptationManager` makes use of this feature when it is made to manage a particular application. Then, using the deployment infrastructure, it does two actions. First, it runs a separate system-like application to monitor the execution environment. Second, it updates the application being managed to monitor its components. The ability to perform application update in runtime proved very useful when instrumenting the application. The manager starts and configures only these sensor components that are required in particular situation. This is clearly visible in the case of `LinkSensor`, which is attached to these components only that are susceptible for reconfiguration i.e. these which are mobile.[12] Observation of links between two immobile components is in our case futile because neither of the adjacent component can be moved.

Another monitoring facility used by the manager is `HomeSensor`. By providing information about creation and destruction of component instances, it is very helpful when migration mechanism is performed. The sensor enables additional verification if migration of a component was successfully

[12]We can easily detect mobility of components by checking if they implement the `CCMRefugee` interface. More about this is presented in the next chapter which discusses the component migration mechanism.

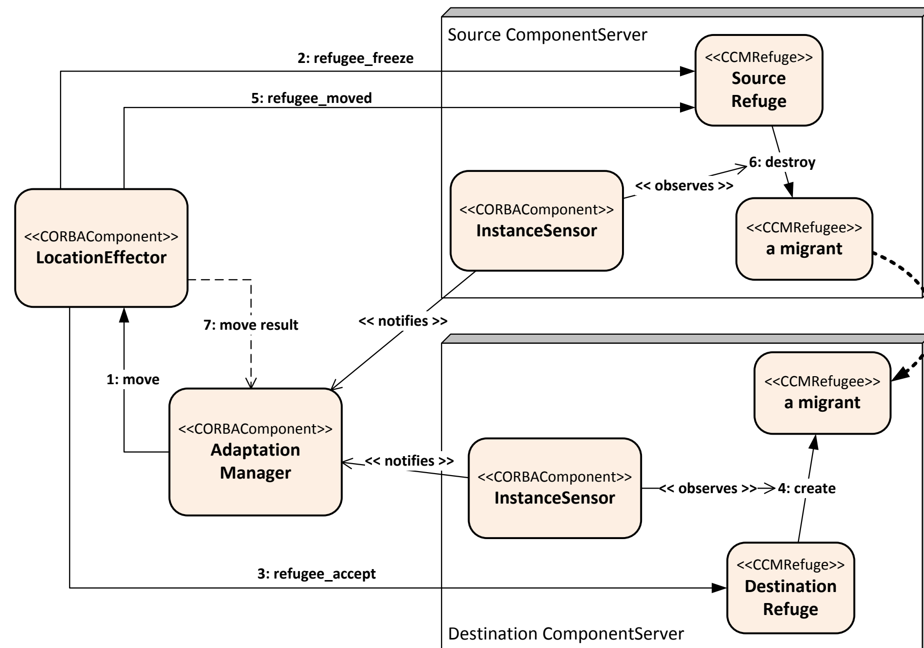

Figure 4.13: The collaboration diagram showing how migration process is observed by the `AdaptationManager`.

completed and in this way improves failure detection. Figure 4.13 presents a scenario when the `AdaptationManager` invokes `move` operation on the `LocationEffector` component. Apart from the result returned by the effector itself, the manager observes the migration being notified by appropriate `HomeSensors`.

Information from the application monitoring together with data coming from monitoring of the execution environment are collected by the `AdaptationManager` and then used in the analyse step. This step includes preparation of the deployment plan in runtime for which we use an adaptive planner. Runtime adaptive planning is, however, a complex task that requires more detailed discussion and thus later in this work we devoted a separate chapter to present it.

Plan and Execute

Provided with a plan update generated by the analyse step, the next MAPE steps include planning how to effect it using the deployment infrastructure and migration effector. As said in the previous section, the component migration mechanism allows moving a component between locations, provided that the destination location is prepared to host the component. The

MigrationEffector developed in the course of this work does not ensure coherence between deployment infrastructure and the current location of application components. This is the role of AdaptationManager to perform the appropriate state synchronization.

Figure 4.14 presents a sequence diagram showing interaction between the manager and deployment infrastructure. Reconfiguration is done in two steps. First the manager determines if the destination location runs the needed component infrastructure. Whenever there is no appropriate CCMRefuge able to host the migrating component type, AdaptationManager starts internal application update to deploy the factory component. In that case the startUpdate operation may not only run CCMRefuge factory but also is likely to start a new component server and container. They are started if the destination location has not hosted these elements already or any of the existing factory, container or component server has inappropriate configuration settings. Once everything is started properly, the finishUpdate operation is invoked to commit changes in the application deployment. Having a proper CCMRefuge running in the target location, the second step of the reconfiguration is performed. This time, however, the manager begins an external application update because we use external MigrationEffector to do the redeployment. Finally when the migration is successful the update is committed with finishUpdate.

4.5 Framework Usage Scenario

In this section we present steps required to use our ADF for managing deployment of a user application. To better illustrate the whole process we showed in Fig. 4.15 the flow of data between all ADF entities involved.

1. An administrator of an execution environment prepares its description according to the D&C *Target Data Model*. It is the best if resources of the environment elements are specified with a widely accepted language such as the CIM schema. The complete description is then passed to the TargetManager (step 1a).

2. Software deployer acquires a software application from a producer and packages it according to the D&C *Component Data Model*. Resource requirements of application components need to be expressed in consistency with the environment resource declaration e.g. using the CIM schema. Packaging may be avoided if the producer provides a D&C/CIM-compatible package already. Next, the application package is uploaded to a selected repository by means of its RepositoryManager interface (step 1b).

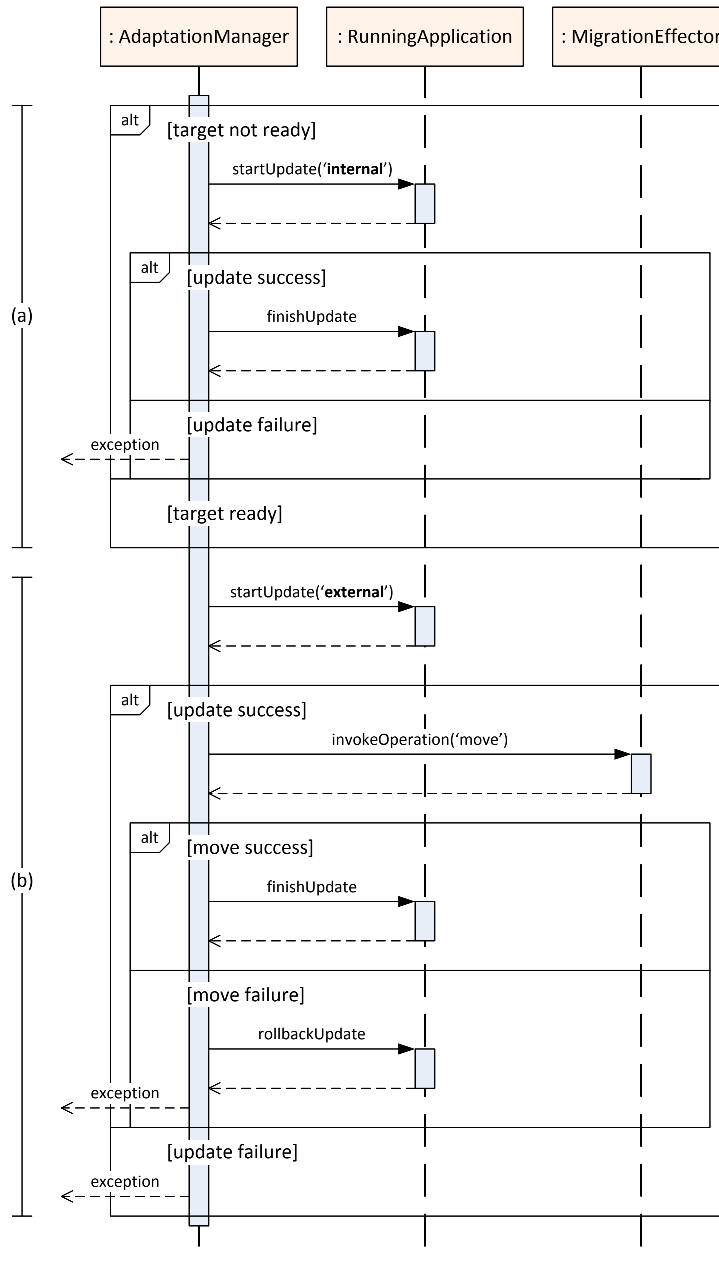

Figure 4.14: A sequence diagram showing two-step interaction between the `AdaptationManager` and the deployment infrastructure during a single reconfiguration attempt; (a) shows preparation of the target location to run components — internal update; (b) shows component migration — external update.

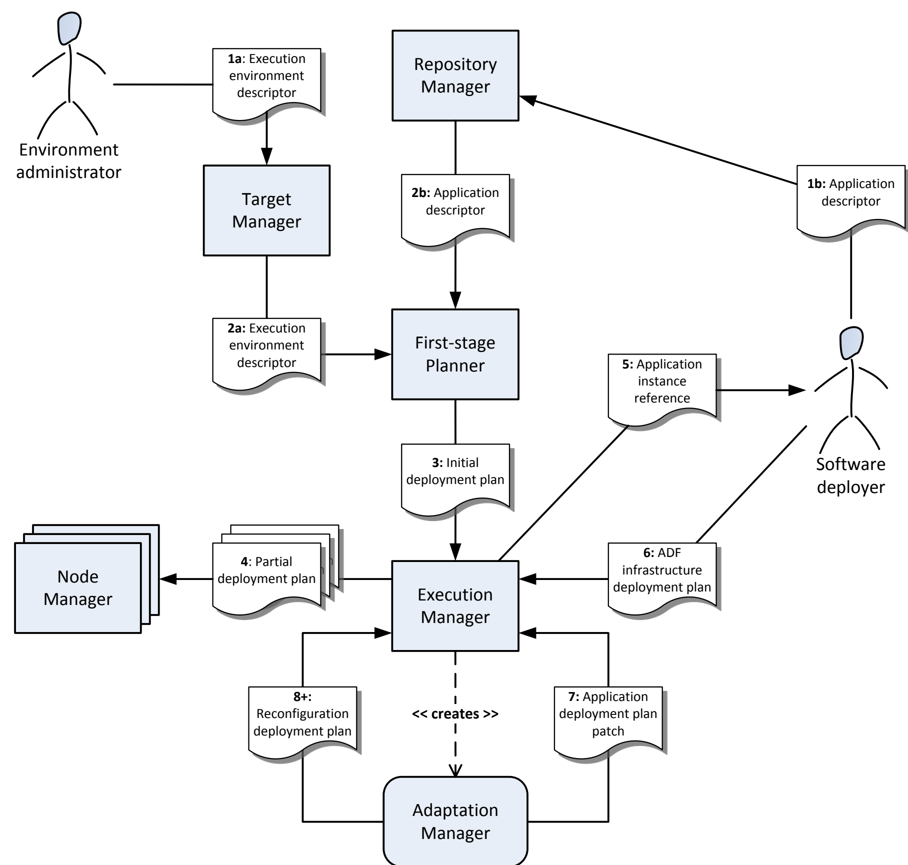

Figure 4.15: The flow diagram showing how ADF may be used to manage deployment of a software application.

3. Using our first-stage planner, software deployer can search for an initial deployment plan. The planner acquires environment and application data (steps 2a and 2b respectively) and produces an initial deployment plan.

4. The initial deployment plan, in the form of the `DeploymentPlan` object, is passed for execution to the `ExecutionManager` (step 3). This results in splitting the plan into parts and deploying these parts over all relevant `NodeManagers` (step 4). In return software deployer receives a reference to the application instance (step 5).

5. Next, software deployer prepares a deployment plan for the ADF infrastructure. We developed a simple ADF plan generator that helps in the plan preparation. The plan includes all sensors and effectors that are independent of an application (e.g. environment sensors) together with the `AdaptationManager` component. The provided application reference is used to configure the `AdaptationManager`. Having the infrastructure plan, software deployer can execute it (step 6).

6. During initialization, the `AdaptationManager` produces a patched deployment plan for the managed application. The patch is used in an internal update (step 7) which runs all application sensors. The set of required sensors mainly depends on the adaptation algorithm and is specific to the manager.

7. Once, the `AdaptationManager` finished the update it continuously monitors state of the system and performs appropriate application reconfiguration (step 8 and later).

4.6 Summary

In this chapter we have presented an overview of Adaptive Deployment Framework we realized in the course of this research. The basis for the framework is the D&C specification, therefore we have limited the discussion mainly to these implementation details that are not covered by the original document. The foundation for the design and implementation of framework elements is CORBA Component Model. We motivated this choice with several key features of the technology: CCM defines an interesting and advanced component model, it is suitable for heterogeneous environments and has already been integrated with the D&C deployment models.

In this chapter we have briefly overviewed the plain deployment infrastructure implemented according to D&C with some of the extensions proposed earlier in Chap. 3. To tackle the complex problem of deployment planning

we split this task between two planners: an initial — first-stage planner and a runtime adaptive planner. This enabled us to reduce complexity of the initial planning that can focus only on static properties of the system. Later, we have discussed more closely the adaptation infrastructure and presented a usage scenario that allowed us to illustrate how ADF entities collaborate to perform deployment adaptation.

The part of the framework responsible for deployment adaptation was designed and implemented according to the CCM model. By enclosing sensors, effectors and the adaptation manager within CCM components, we not only leverage built-in communication mechanisms but also:

- can easily follow the Autonomic Computing paradigm when designing the adaptation infrastructure,
- can clearly separate the adaptation logic from sensors and effectors. This, in turn, enables the adaptation logic to be easily exchanged for some other, improved version,
- make use of the plain deployment infrastructure to deploy the manager, sensor and effector components,
- are able to deploy sensors and effectors on demand to only these locations where they are really needed. This minimizes overhead incurred by the instrumentation.

It is important to mention that the sensor, effector and adaptation manager components implemented in the course of this work are only a facade for internal mechanisms working behind the scenes. In the next chapters we delve deeper into implementation details and discuss runtime component migration and component portable interceptors mechanisms followed by a runtime deployment planning algorithm.

Chapter 5

Monitoring and Management Infrastructure

Our Adaptive Deployment Framework, modelled according to the Autonomic Computing paradigm, separates an adaptation logic from a low-level monitoring and management infrastructure. The previous chapter presented a general overview of both these layers and the way how they interact with each other. In this chapter we discuss an important problem of interfacing the management layer with an execution environment. We only focus on the application management sublayer because monitoring and management of the execution environment is ensured by the CIM and WBEM infrastructure.

From the set of defined earlier redeployment techniques we concentrate in this work on *runtime redeployment*. To implement this technique we used runtime component migration as a basic reconfiguration mechanism and transparent communication interception for application monitoring. The main motivation for this choice was as follows:

- runtime redeployment is supposed to guarantee the most *agile* adaptive deployment system,[1]
- the responsiveness of the deployment adaptation enables using more sophisticated runtime planning algorithms, and finally
- it is a challenging endeavour to build all the mechanisms that enable the component migration and communication interception.

The issues discussed in this chapter are closely related to the underlying technologies we used for ADF implementation i.e. CORBA and CCM. We believe,

[1] Adaptation agility is a property of an adaptive system that determines speed and accuracy with which it detects and responds to changes in its execution context. The term was first coined by Noble et al. in [91].

however, that the main discussion points can be used for other distributed communication platforms as well.

5.1 Support for Runtime Component Migration

In this section we present the most important issues related to implementation of the runtime migration mechanism for CCM-based components. Additional details can be found in our previous work [13, 14].

The CORBA environment is well suited to resolving the problem of object migration in runtime. Built-in facilities such as Portable Object Adapter, servant managers, and `ForwardRequest` exception may well be used to support a migration mechanism for CORBA objects and CCM components. Before we present how to move a component from one place to another in runtime, it is important to analyse how, in general, movement of objects in a distributed environment is performed. An important assumption we made is that the objects which we consider are multithreaded. It means there exists 1-to-N relation between a mobile distributed object and execution threads that operate on its state. Although this is in contrary with the approach proposed by e.g. ProActive and it significantly increases complexity of object migration, multithreaded operation addresses a common use case that could potentially be applied to a variety of technologies such as CCM, GCM and Windows Communication Foundation (WCF). Moreover, it much better fits the increasing interest in multicore computer systems.

Following we present the stages which a running object has to go through when migrating from one location to another. :

1. **Suspending** the object is required to store its state consistently. The main issue here is in preserving the safety and integrity of the object and thus the whole system. Following suspension, the execution platform still has to deal with incoming, ongoing and outgoing requests, therefore suspension requires the object to achieve *quiescent state*.[2] From the object standpoint it means that after suspension and before activation it must not respond to any requests that can change its state. Otherwise, the stored state of the object would not match the actual state altered by the invocations and this would lead directly to loss of information. These issues are well discussed in [106].

[2]As defined in [72], an object is in quiescent state if: (1) it is not currently engaged in a call that it initiated, (2) it will not initiate new calls, (3) it is not currently engaged in servicing a call, and (4) no calls have been or will be initiated by other objects which require service from this object. As presented later, we relax the last condition and require that no calls initiated by others *will reach* this object.

2. **Storing** the state of the suspended object alone or together with code. Which action is to be performed depends on availability of the code at the destination. It is also crucial to answer the question what the state of the object is. If the object is connected with others, we must know whether they need to be copied as well or perhaps can be accessed remotely (shallow/deep copy problem). To make things even more complicated, storing state may also take into account heterogeneity of the environment and prepare a copy in an easily transferable format. Some of these issues are covered in [65, 68, 95, 97]. As our work is based on OpenCCM platform implemented in Java we may simplify this important aspect and focus more on other migration issues.

3. **Moving** the state between the source and target locations. This step is quite straightforward although migration requires the target location to be ready to accept incoming objects. For this reason, an appropriate hosting infrastructure must be prepared at the destination. Moreover, in case of problems with transferring the data, it should be possible to withdraw the whole process and return the system to the state just before the object suspension.

4. **Loading** the state of the object at the destination. This step requires the code of the moving object to be available at the destination. In case of heterogeneous environments, such as CORBA and CCM, this requirement is sometimes hard to fulfil — e.g. how to move a Java implementation of an object to an Object Request Broker (ORB) based on C++ language. Loading is much easier if we can assume platform homogeneity, such as offered by Java or .Net environments. Again, a proper deployment infrastructure can alleviate this problem by planning that directs the moving object to these locations only which can host it.

5. **Reconnecting** of the moved object in such a way that every other object/client communicating with the migrating object should not see any change in behaviour. Three possible techniques of referencing a moved object exists: (1) deep update, (2) chain of reference, or (3) use of home location agent. More details about this issue are presented later.

6. **Activating** the new incarnation of the object at the new location followed by destroying the original one at the previous location. This is the final step which ends the whole process of migration and results in the fully functional system.

All the presented steps are pivotal to implement migration mechanism for distributed objects and components. Later in this section we discuss some of the most important issues that we encountered when implementing runtime

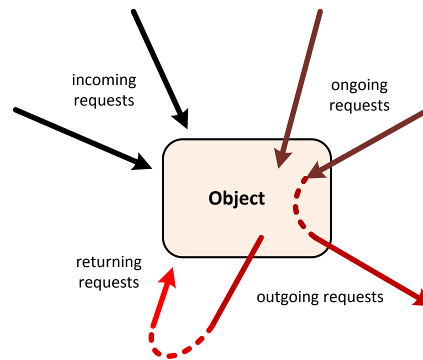

Figure 5.1: Four possible cases of when dealing with requests during object passivation.

component migration. What is worth mentioning here is that in the following discussion we often refer to CORBA objects instead of CORBA components. Firstly, this is because CCM does not in itself provide any mechanism that facilitates migration of components. Therefore the solutions we propose are based at the object level. Secondly, this is not a real issue as, to some extent, a CORBA component may be perceived as a collection of CORBA objects. Accordingly, whenever we refer to an object it means that this issue concerns both: objects and components (more specifically an instance of `CCMObject` and all server-side ports of the component instance). Otherwise, the problem is related to components only.

5.1.1 Suspension and Dealing with Requests

One of the major challenges required to respond to when resolving runtime migration problem is dealing with requests which an object is or should be involved in. In other words this is a problem of reaching quiescent state as defined in [72]. This problem arises when the object is going to be suspended to preserve consistency of its state but it is still entangled in some request processing. As illustrated in Fig. 5.1, four possible cases are important for an object to deals with:

a) *incoming requests* while it is suspended,

b) *ongoing requests* invoked just before suspension,

c) *outgoing requests* invoked by the object just before suspension, and

d) *returning requests* invoked on the object as a result of previous outgoing requests.

For the first problem we considered two solutions. One involves collecting all incoming requests until the component is reactivated again. In case of successful migration all the collected invocations are redirected to the new location using standard CORBA ForwardRequest exception. The problem with this approach is, however, that in case of abundance of incoming requests the collecting buffer is becoming quickly full and, ultimately, when it reaches the size limit any subsequent invocations need to be discarded. Moreover, this approach is prone to deadlocks in case of returning synchronous invocations that, similar all the others, are blocked waiting for reactivation. The other more straightforward approach is to discard all the incoming invocations immediately after the suspension succeeded and until the component is reactivated. For this purpose, we use the CORBA TRANSIENT exception that informs the caller about some temporary communication problems. In reaction to this exception the caller ORB usually automatically reissues a request again.[3] Although it shifts the problem of incoming requests to the client side, ORB can hide it from the client and ensure a transparent resolution.

Unfortunately, the case of ongoing requests is more troublesome when dealt at the application or middleware level. In a multithreaded environment[4] it is likely to happen that a component serves one or more client calls while suspension is requested. In general case, we cannot easily pre-empt the thread responsible for a request in service and, therefore, passivation cannot progress until all these call are finished. Otherwise, they could influence the state of the object what would result in loss of information during storing phase. To address this issue we make use of the standard ServantLocator manager. Two operations of the manager interface — preinvoke and postinvoke — are used to count the number of ongoing requests. The locator ensures that passivation does not progress until all ongoing operations are finished and in the same time, by discarding all other incoming requests, it avoids starvation of this process. Such a simple solution may, however, impose significant delays in suspending an object what developers should take into account when programming a mobile object. Once the passivation has started they ought to finish all ongoing request as soon as possible.

The third of the mentioned problems — outgoing requests when passivation of the object is requested — may cause even more delays. Outgoing requests may result from earlier ongoing invocations but also can be issued

[3]The exact reaction on the exception depends on the exception minor code and completion status. In this case, we use minor code 1 which means that request was discarded because of resource exhaustion or discarding state of the target POA. The CompletionStatus is set to COMPLETED_NO as the invocation never reaches the application code.

[4]Our implementation assumes the multithread threading model of a mobile component. It means that the container will not prevent multiple threads from entering the component simultaneously, which is much more interesting case comparing to the serialize threading model. The latter protects the component from any concurrent calls.

by the object itself e.g. by some internal execution thread. Again, it is important that the object is suspended when all outgoing invocations are finished already. Otherwise, the returning result could introduce some inconsistencies between the stored and current state of the object. There is no easy way, however, to determine the number of outgoing requests at the middleware level. In case the object communicates with external entities using only one selected communication technology such as CORBA, we could use the standard COPI interception mechanism. Conversely, if the object interacts with its environment by some other means too, it is hardly possible to provide an appropriate interception mechanism for all other communication technologies. We decided not to restrict developers in the matter how they communicate with external entities for the cost of their awareness of the object passivation. The benefit of this approach is that explicit passivation makes programmers to consider more carefully how the mobile object may interact with the external world. This is also important for returning requests.

The last issue, which tackles returning requests, is a combination of the outgoing and incoming request problems. It occurs when an outgoing synchronous operation call on some external entity causes eventually an invocation on the object itself. Restricting the way how an object may communicate with its environment we considered solving the problem by means of the Object Transaction Service (OTS) service [98]. An interesting property of OTS is an implicit propagation of a transaction context, which we could use to mark outgoing requests. In this way we could distinguish the returning requests that contain the transaction context from other purely external incoming invocations. A disadvantage of this approach is that not only it restricts the communication technology of the object but also it requires transferring of a transaction context by all external entities on the invocation path.[5] This is in case of heterogeneous distributed systems very hard to attain. Later, in this chapter we present a simple pattern which component developers can follow to tackle the problem or returning requests and passivation.

5.1.2 Factory Support for Reconnection

Another major challenge related to migration is reconnection between the migrated object and all other clients and entities that it interacts with. The importance of the reconnection stems from its direct impact on the *residual dependencies problem* which is one of the most fundamental issues related to mobility of a running code. The residual dependencies problem is the

[5] In the area of Distributed Transaction Processing (DTP) this is a well known problem that is addressed e.g. by XA standard [80]. We find this approach too perplexing for a migration service, though.

level of dependency of a migrating entity on the source location and it is the main factor that restrains broad use of migration mechanism. Poorly designed reconnection causes that successive movements of an object make it dependent on more and more systems what, in result, substantially reduces fault tolerance of the entity and the application as a whole. Therefore, it is important to carefully analyse how reconnection in distributed systems is done. In general, three possible techniques of resolving this issue exist: (1) deep update, (2) chain of reference, or (3) use of a home location agent.

From the deployment standpoint the first technique is the most obvious choice. If deployment infrastructure is able to connect all the components during application activation, it is also able to reconnect the moved components i.e. disconnect and connect again according to an updated plan. This approach presents, however, two undesirable effects. It is very expensive because it involves updating all other related components and, in fact, not a viable solution in distributed environments such as CORBA, since the clients may not yet exist when migration occurs.[6] Moreover, it is likely that the clients of the migrating component are not under management of our deployment infrastructure, hence they would need another way to refresh their references.

The second technique — chain of reference — is impractical due to significant inefficiencies in communication and because it aggravates the residual dependency problem as more and more systems are involved in transfer between clients and the object. For this reasons, we realized the reconnection following the last approach i.e. using a Home Location Agent (HLA) that seems to be the most appropriate for resolving the problem of referencing mobile entities in distributed systems.[7] Moreover, the solution with HLA very well fits the design of the CCM model. There is no need for any additional external location services, such as one proposed in [64] and [93, Sect. 11.5], as this role can be taken over by a component factory. A factory has all the code needed to create a component instance, therefore, it can be easily extended to be able to recreate an instance if provided with a stored component state and become HLA.

For every component the CCM model provides a standard factory interface `CCMHome`, which we simply extended to fulfil requirements of accepting mobile components — `CCMRefugees`. As shown in the Listing 5.1 the `CCMRefuge` interface has five operations supporting movement of components.

Listing 5.1: The IDL definition of the extended component factory interface

```
import ::Components::CCMHome;
import ::Components::CCMException;
```

[6] A very captivating explanation of this problem is presented by Henning in [58].

[7] The same approach for referencing portable hosts can be found in the MobileIP protocol [44].

```
import ::CosLifeCycle::Criteria;

interface CCMRefuge : CCMHome {
  Criteria refugee_freeze( in CCMRefugee refugee_here )
  raises (CCMException);

  void refugee_moved( in CCMRefugee refugee_there )
  raises (CCMException);

  void refugee_unfreeze( in CCMRefugee refugee_here )
  raises (CCMException);

  CCMRefugee refugee_accept( in Criteria refugee_state )
  raises (CCMException);

  void refugee_update( in CCMRefugee refugee_there )
  raises (CCMException);
};
```

The presented extensions to the factory interface are threefold:

- required at the source location — three operations `refugee_freeze`, `refugee_moved` and `refugee_unfreeze`. The aim of the first is to prepare a component and the infrastructure for movement. It suspends the component and returns its state in a `Criteria` sequence. The second operation is responsible for reconnection of the moved component. Its argument refers to the instance created at the destination. The last operation, `refugee_unfreeze`, is called in the case of movement failure when it is needed to reverse suspension of a component and return the system to the state just before migration attempt,
- required at the destination location — the `refugee_accept` operation is invoked to ask the target factory to accept a moving component. The operation returns a newly created incarnation of the component used further to reconnect the references. In case of problems, the operation throws an exception that is a signal to withdraw whole migration attempt and call `refugee_unfreeze` at the source.
- required at the HLA location — `refugee_update` is an internal operation that does a part of the reconnection task. It is used by the source factory to update the agent about the most recent component location.

Figures 5.2a-d present how the presented operations are used by `Migration-Effector` in the most common cases. First case depicts a successful migration from the location where the home agent `CCMRefuge` is running. Second scenario, presented in Fig. 5.2b, is a bit more complex because the home

agent needs to be updated with the most recent location of the component. The update is a part of the reconnection process and, therefore, is done by the source factory. The last two diagrams present the most common failures during migration. One is caused by the destination factory reporting an exception on the `refugee_accept` operation. The other, less common, may be induced by some problems with reconnection.

All these scenarios show that from the point of view of `MigrationEffector` (or any other client requesting migration) using the extended factories to move a component is very simple and in most cases limited to only three operation calls. This is, however, only an external facade which needs to be supported by lower-level mechanisms. The core of our solution makes use of the `ServantLocator` manager and the `ForwardRequest` exception. Mobile components are always run in containers that use servant locator to find appropriate servants for them and their server-side ports. This locating is done with help of the Active Object Map (AOM) table.[8] However, when a component has moved to a foreign location and a servant cannot be found in AOM, the locator looks to the additional *migratory table*. This table maintains association between component or port identification and the location of a component; either the most recent if this is the HLA container or pointing to the HLA if this is any other container where the component was running previously. Once the remote location of the component was found in the migratory table, the locator raises a `ForwardRequest` exception to direct the caller where to find the target.

This solution very well fits the existing CORBA approach to object mobility and guarantees effectiveness and high scalability of the reconnection mechanism. When a component is located in its home container (i.e. the one that includes HLA) there is no additional overhead in request processing. The locator behaves exactly the same as in the case of immobile components. Overhead appears only when a component is running outside of its home container. Then, a client contacting a previous location of a component receives a forward request to the component's HLA. Next, invoking operation on the HLA it receives a forward request again. This time, however, redirection leads to the actual location where a component is running.

5.1.3 Life Cycle of a Mobile Component

The aim of our solution for component migration was not to provide a purely transparent mechanism for programmers, which we think is hardly possible to attain at the middleware level, but rather to create a framework that helps them to deal with component mobility. One of the major benefits of making

[8]Actually, AOM is a table used internally by the POA object adapter but we use this concept in exactly the same way.

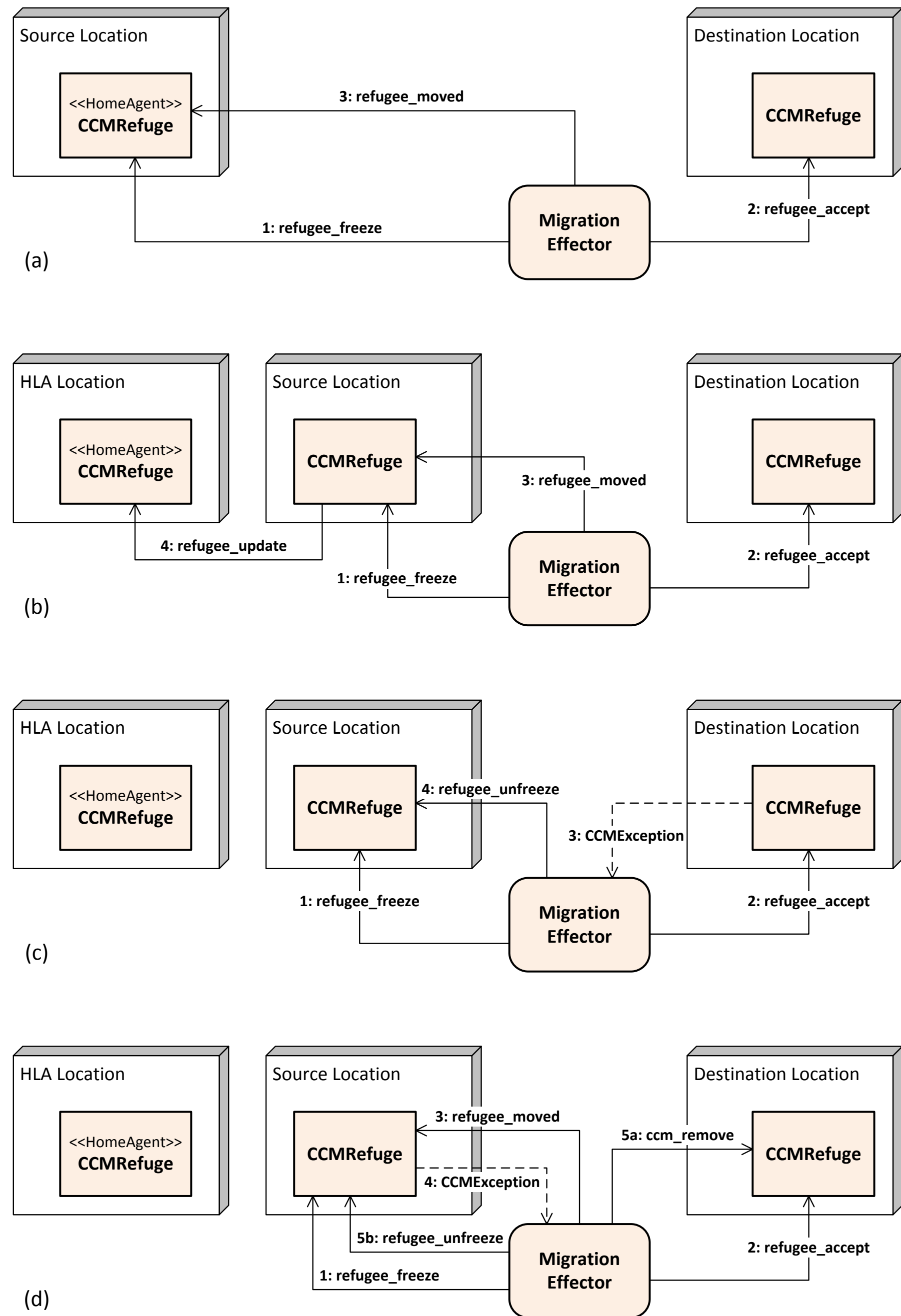

Figure 5.2: The most common scenarios of collaboration between migration effector and mobile component factories; (a) successful migration from the HLA factory, (b) successful migration from a non-HLA factory, (c) the most common case of unsuccessful migration caused by problems with accepting a refugee component, (d) the second most common case of unsuccessful migration caused by problems with reconnection.

component developers aware of migration is substantial freedom of using software and hardware platforms of their choice for implementation. For our prototype ADF framework, the basis for component development is the Java language and CORBA Component Model platform but our intention was not to limit access to local resources or native communication technologies. Instead of enclosing a component in merely the CORBA technology and IDL interfaces, we aim at opening its implementation for the outside world. This is in contrary to one of the properties of a component provided by Szyperski in [119] who states that:

> A software component is a unit of composition with contractually specified interfaces and *explicit context dependencies only*.

Although the explicit context dependencies facilitate component reuse, portability and mobility, we argue that imposing such constraints on components is impractical. For example, it would not allow a component programmer to create a new thread to do some background task unless the component's context provides suitable thread management API. It would also forbid creating a GUI interface unless the container explicitly provides a graphics API. In general, it would cause problems with accessing any local resources that are not covered by the appropriate context interfaces.

We decided to follow the opposite direction and let programmers use as much of technology and resources as they want and without any constraints. However, they need to be aware that if a component is mobile it may happen that during activation some resources may be not available. In the case a developer does not want to deal with situations when vital resources are unavailable, they may describe such specific requirements of a component in its deployment descriptor. In result, having the migration mechanism integrated with a deployment infrastructure, they can consciously limit component's mobility to only selected platforms. In the same time, programmers are freed from most of the burden of migration issues as it is resolved by the proposed middleware layer.

In order to make developers involved in the migration process smoothly, we extended components' life cycle. Again, the CCM model proved very convenient for this task. Listing 5.2 presents a definition of the `RefugeeComponent` callback interface that is a base for any mobile component implementation.

Listing 5.2: The IDL definition of a mobile component callback interface

```
import ::Components::CCMException;
import ::Components::EnterpriseComponent;
import ::CosLifeCycle::Criteria;

local interface RefugeeComponent : EnterpriseComponent
```

```
{
  void ccm_refugee_passivate( )
  raises (CCMException);

  void ccm_refugee_activate( )
  raises (CCMException);

  void ccm_refugee_store( out Criteria state )
  raises (CCMException);

  void ccm_refugee_load( in Criteria state )
  raises (CCMException);

  void ccm_refugee_remove( )
  raises (CCMException);
};
```

Operations in this interface are invoked by a component's container and indicate changes in life cycle of a mobile component. They should be used by a component developer to control: resource usage, progress in communication and internal state of the component. Following we describe the exact meaning and proposed use of each operation:

- `ccm_refugee_passivate` is called just before passivation of a component. A developer shall use this indicator to prepare the component for storing phase i.e. the component should interrupt any activities which may change its state after return from this operation. By throwing an exception, a developer may inform the migration infrastructure that the component is not yet prepared for the migration attempt. Later in this section we present more details on the most common scenarios related to the passivation problem,

- `ccm_refugee_store` is called to store the current state of a component after it was suspended. Although some languages such as Java and C# can serialize classes automatically by means of the reflection mechanism, in this work we adopted manual approach in order to preserve greater portability of CORBA environment,

- `ccm_refugee_load` is opposite to the store operation and shall be used by developers to restore the state of a component. All information included in the provided `state` argument should be enough to recreate the component instance to the form as close as possible to the one just before passivation. Once this operation is invoked, a developer is certain that the current instance is located on the destination host.

- `ccm_refugee_activate` is called in two cases. Firstly, after successful migration the operation is invoked on the newly created component

instance at the target location. Otherwise, if migration fails, it is called at the source location to indicate that the component returns to its operation as if migration attempt have not occurred at all. In general, `ccm_refugee_activate` indicates that an instance is going to be activated and gives developers chance to prepare the component to running state,

- `ccm_refugee_remove` is called on the component at the source location whenever the migration attempt is successful. The aim of this operation is to indicate that the component should release all resources acquired during its work at the source location. Once `ccm_refugee_remove` has returned, any other operation will not be called on this instance and a developer shall assume that it is destroyed.

All these callback operations are crucial for a component programmer to develop a well functioning mobile component. From this set, however, the most troublesome might be implementing the `ccm_refugee_passivate` operation when returning synchronous operations are expected.

5.1.4 Passivation During Synchronous Requests

In a multithreaded component environment special care is required while dealing with operations that can result in synchronous returning request and component passivation has been requested. We term these operations *composite* because they refer to some remote system what results in a callback call. The major problem in such a scenario stems from the fact that incoming requests do not convey any additional context information and cannot be easily distinguished from returning requests. Therefore, to properly deal with this situation some application-level knowledge is required.

The general contract is that when a component instance receives a notification of passivation it should stop all threads of execution such that during `ccm_refugee_store` it can safely store its state. Writing non-composite operations, developers may pay less attention to passivation. After returning from `ccm_refugee_passivate` the container will block any new incoming calls and wait until all ongoing requests are finished. However, when a composite operation is in progress it may happen that it made a synchronous call to some external system, is waiting for a returning callback call and a passivation has been requested. In this case, a developer has to consider two options: (1) they can simply return from `ccm_refugee_passivate` or (2) they can hold `ccm_refugee_passivate` until returning operation is processed. The consequences are presented in Fig. 5.3a–b.

The advantage of the first approach is its simplicity. A callback request coming from the external system, similarly to all other incoming request,

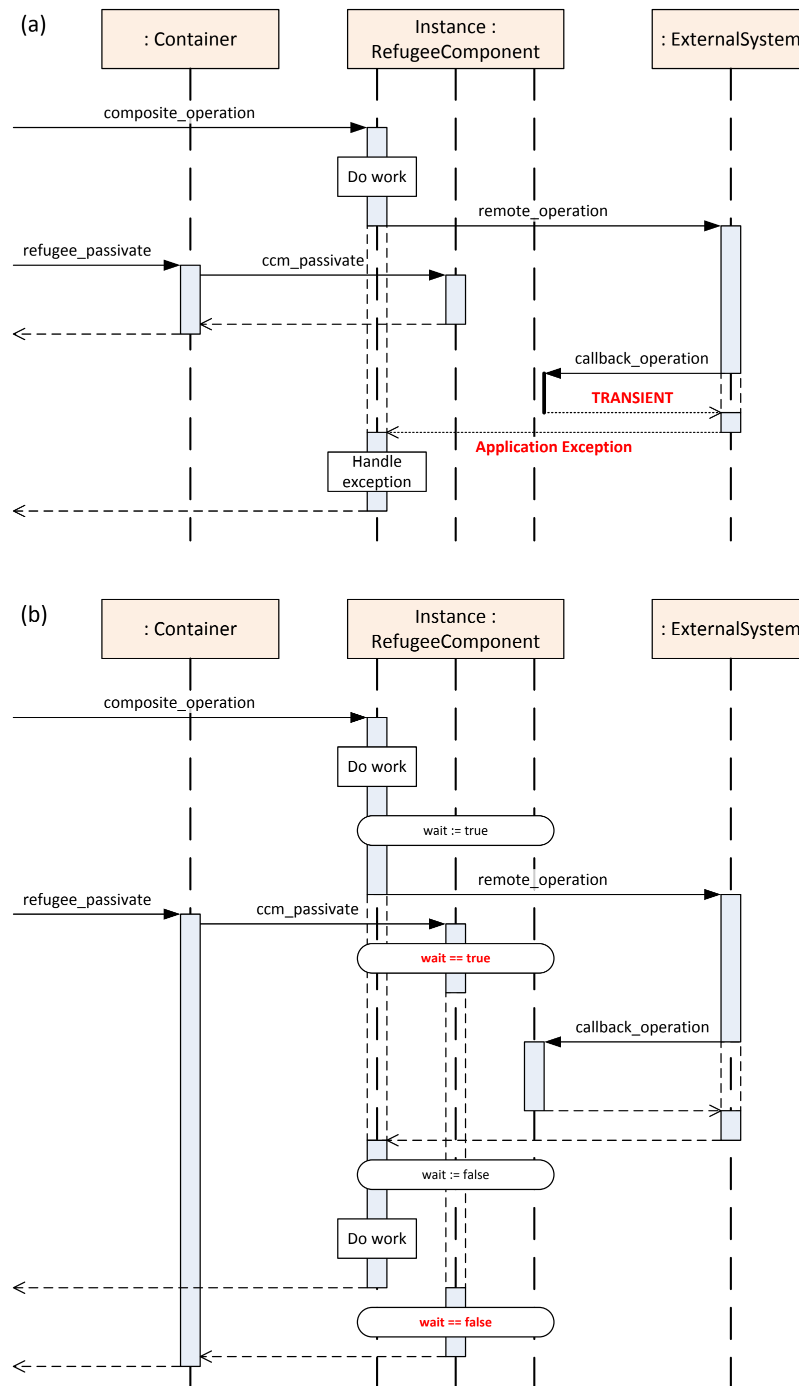

Figure 5.3: Two options of handling passivation requests when a composite operation is in progress; (a) a simple approach which allows for application exception, (b) holding passivation until the operation finishes.

would not reach the component executor but result in the `TRANSIENT` exception instead. If this exception results in an application exception returned from the remote operation call, this scenario is valid. It may happen, however, that the external system will try reissuing callback operation indefinitely what leads to a deadlock. In such a case a better approach may be to use a boolean flag to mark the remote operation call and hold the passivation until the remote call returns (Fig 5.3b). Unfortunately, disadvantage of this solution is that if passivation is blocked, all other incoming requests will be accepted as valid what could result in starvation of the passivation process. Figure 5.4 presents the solution we implemented using two boolean flags. It is based on the previous idea which blocks passivation until a remote operation is completed but, additionally, it uses a second boolean flag to mark passivation in progress. When accepting all other operations, a programmer can check if passivation is in progress and throw the `TRANSIENT` exception when needed. Developers can adopt this to address the passivation issue properly. It is worth noting, however, that the discussed problem is related to synchronous remote invocation only and does affect event-based and asynchronous calls.

5.1.5 Summary

The presented migration infrastructure provides programmers with a convenient mechanisms supporting mobility of components in runtime. The approach adopted does not provide fully transparent solution that seems to be unattainable for multithreaded middleware-based systems. Instead, we propose an extension to component's life cycle that makes developers aware of the migration process and, in the same way, does not impose substantial constraints on the range of resources and software technologies used in component's implementation.

We found that the component level and particularly the CCM model is very well suited for migration. Not only the components are of proper granularity but also they can be deployed separately which is very important for migration. Deployment infrastructure allows determining if a target location is able to accept a mobile component and enables preparing required execution infrastructure.

Our prototype implementation of the migration mechanism shows that the most important issues related to this problem were successfully resolved. There are, however, many directions that could be further developed to make the mechanism more convenient and comprehensive. One, especially interesting, is support for migration of components with stream ports. We were involved in work on a prototype implementation of the Streams for CCM extension [71] and believe that these two concepts can be successfully integrated.

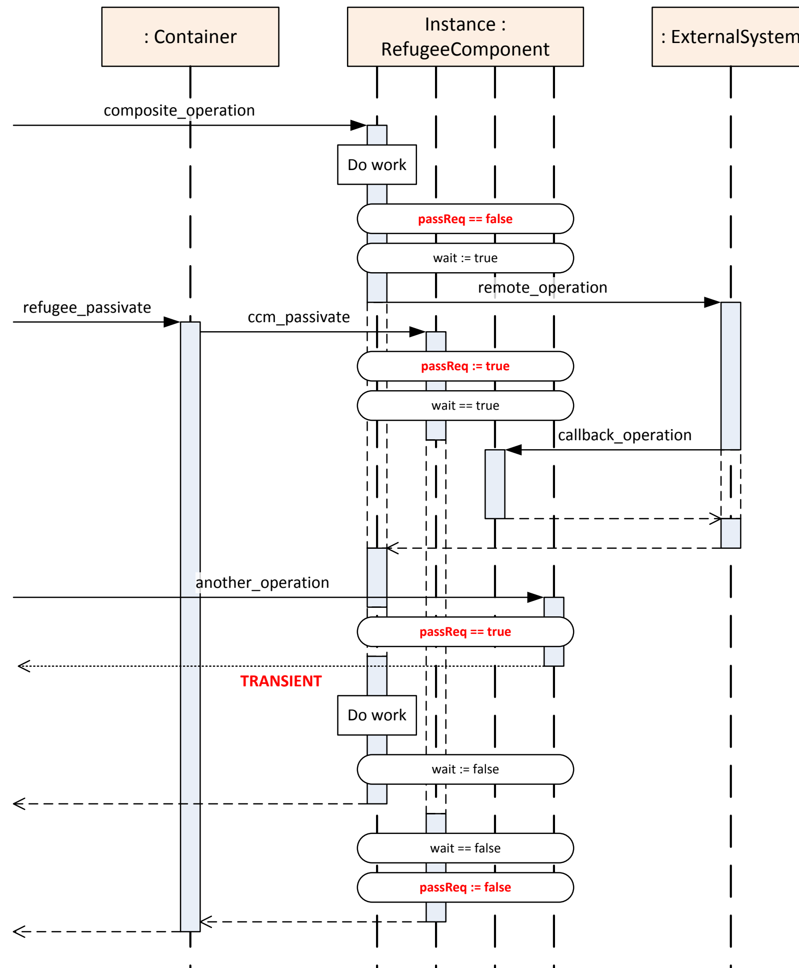

Figure 5.4: A solution to the passivation problem while a composite operation is in progress.

5.2 COPI-based Application Monitoring

The interceptor design pattern has found its use in many cases such as message logging, encryption, filtering and monitoring. We use interceptors in ADF mainly for transparent application monitoring. Under the umbrella of CORBA standards two interception mechanisms are defined: Portable Interceptors (PIs) [102, Sect. 16] related to plain CORBA objects and Container Portable Interceptors (COPIs) [101] addressing interception of components. Plain PIs were proposed much earlier and are available on many ORB platforms, however, they are inadequate to intercept component instances. The main reason for this is the problem with determining component instance identity esp. when a client-side component is considered (i.e. a component with receptacle, publisher or emitter ports). More recently, however, OMG defined the COPI interception mechanism that takes into account the CCM container architecture and the component setting. Unfortunately, this mechanism is rarely available in existing CCM platforms.

The main advantage of the COPI mechanism is in the ability to identify both communicating sides: a client- and a server-side component.[9] This enables many useful use cases such as enforcing security and QoS policies. Moreover, the mechanism may be used not only to monitor but also to change the intercepted requests. The COPI specification defines two levels of interceptors: *basic* and *extended* (Fig. 5.5). The basic COPI interceptors corresponds to the capabilities of the CORBA PI. They enable observation of the communication but does not allow influencing on a request. Conversely, extended interceptors provide an additional functionality to change a request and, therefore, the primary use of extended level interceptors is to modify component behaviour. The extended interceptors can for example return a result for an operation call and can prevent the further call processing from reaching the operation implementation. This may be important when integrating security into the component model [111]. Our intention, however, was merely to monitor the communication patterns between components, hence we implemented the basic level of COPI interceptors and then integrated it with the OpenCCM platform.

To implement functionality of the basic interceptors we made use of OpenCCM container plugins and plain portable interceptors. Figure 5.6 shows the detailed view on how request is transmitted from a client-side to a server-side component. Our `COPIController` container plugin is attached to client-side component ports and collects needed identification information

[9]Actually, the CCM model does not distinguish between client-side and server-side components. However, we use these terms to underline the fact that components can have two kinds of ports: client-side (receptacle, event publisher, source) and server-side (facet, event consumer, sink). COPI provides means to identify both communicating components irrespective where the interceptor has been attached.

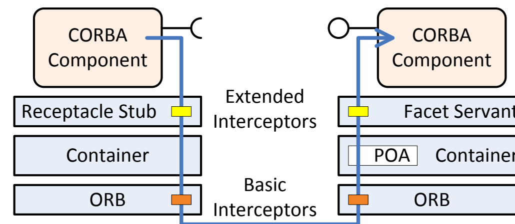

Figure 5.5: The general view on the basic and extended level of COPIs.

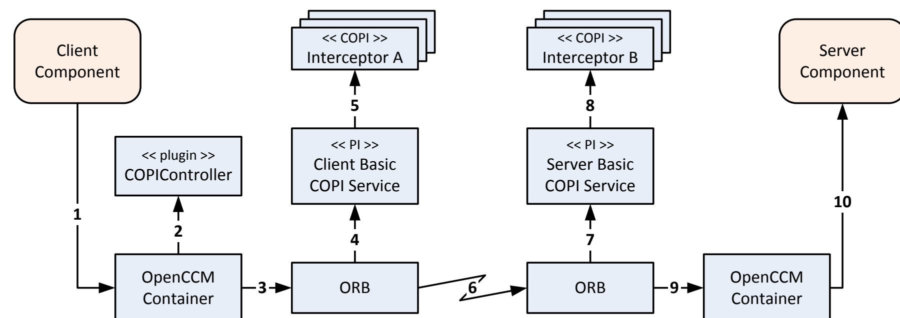

Figure 5.6: The detailed view on request transmission between client and server components when the COPI infrastructure is enabled.

which is stored in the `PICurrent` object related to the request. The basic COPI service is implemented as a plain PI that reads information stored in `PICurrent` and invokes all registered COPIs. After all container portable interceptors has been called the identification information is passed to the server-side component in the `ServiceContext` object associated with every client request. At the server side, ORB calls all registered portable interceptors and our basic COPI service. The service reads identification information from the request `ServiceContext` object and invokes all server-side COPI interceptors. Finally, the request is passed to the server component.

The major difficulty when implementing the COPI specification was to achieve proper identification of both communicating components. Currently, the handling of component instance identifiers is not supported by the CCM model. Therefore, we proposed a non-standard way to identify instances. The following section provides more details on this issue.

5.3 Component Instance Identification

Proper identification of component instances plays key role in many use cases. The *Quality of Service for CORBA Components* specification [101] uses identity of a component instance when associating non-functional properties

with them. We consider instance identification as very important for COPI, deployment and migration mechanisms. The deployment infrastructure needs identifying instances to bind, update and destroy them properly. The COPI interception must identify both communicating sides as soon as a request reaches a client-side stub. Lastly, the migration mechanism needs component identification to do correct request forwarding.

According to the CCM specification [103, Sect. 6.1.4 and 6.4.4] component instances are identified primarily by its component references, and secondarily by its set of facet references. The model provides the `same_component` operation that allows clients to determine reliably whether two references belong to the same component instance. Additionally, it offers the `get_component` operation that allows them to navigate from port to a component's reference. However, the definition of "same" component instance is ultimately up to the component implementer, in that they may provide a customized implementation of this operation. We find the proposed approach not efficient enough especially when the interception mechanism is considered. To verify if two references represent the same component with the `same_component` operation it requires at least one remote call.

Apart from instance identification proposed by CCM, the deployment infrastructure based on the D&C specification uses another two forms of identification. First, user-defined, is included in a software package descriptor represented using the *Component Data Model*. It combines `name` attributes from package and instance descriptions forming a "path" of the instance's origins in the model. The length of this path depends on the recursion level of application assembly components. Second form of identification is carried by a deployment plan created according to the *Execution Data Model*. It keeps association that enables navigating from instances included in a `DeploymentPlan` to appropriate instances included in a software package description. Neither of these identifiers, however, are related to a running instance of a CCM component which they describe.

To create relation between a running component instance and its description we proposed two extensions to the D&C models (Fig. 5.7a–b). The `DnCComponent` interface ought to be implemented by a CCM component to allow component users to: (1) determine the name of the component instance as described in the deployment plan and (2) retrieve a reference to the `RunningApplication` object which enables asking for the current deployment plan of the application. Using this plan and having the instance name we can e.g. determine deployment requirements of a selected component what is needed for component migration. Additionally, `RunningApplication` provides operations for searching component instances within a deployment plan. Together the proposed interfaces allow for easy navigation between running component instances and their deployment descriptions.

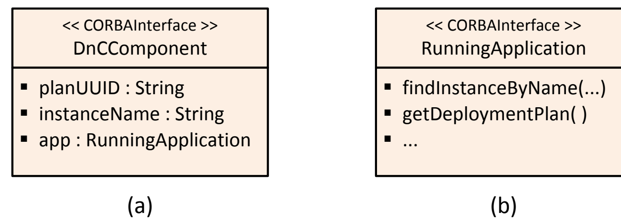

Figure 5.7: The extensions creating a link between a running component instance and a deployment plan; (a) the `DnCComponent` interface enables identification of a running application and component instance name; (b) the `RunningApplication` interface allows finding running component instances and the deployment plan related to the application.

Unfortunately, the name of a component instance included in a deployment plan cannot be used as a proper component identifier for the COPI and migration mechanisms. The deployment identifier is not globally unique as the same plan can be executed multiple times in a domain. Therefore, the `DnCComponent` interface and instance identification included in the D&C models are used mainly for deployment purposes, whereas COPI and runtime migration require an additional component identification. Container Portable Interceptors need instance identifiers by definition — to enable identification of communicating sides. The component migration mechanism requires instance identifiers to be a unique property of a component, which is invariant while the component is moving from a location to location. Otherwise, it would be impossible to follow a moving instance and redirect communication properly. In result two issues of instance identification need to be addressed: (1) uniqueness of the identity to ensure that two different component instances have different identifiers even if they origin from the same deployment plan executed more than once and (2) ability to determine instance's identity given its reference to be able to follow a migrating component.

In order to ensure uniqueness of the identifiers we simply use Universally Unique IDentifiers (UUIDs). They do not need any centralized authority to administer them and may be automatically generated preserving uniqueness with very high probability [76].

The problem of how to determine instance's identity is more difficult to solve. As mentioned earlier in this section, the `same_component` and `get_component` operations are means to compare component and facet identity. They are, however, not enough to determine its identification. To ensure effective way to retrieve component identifiers we decided to include the instance identifier in component and all server-side port references. More specifically, we included the component id in the `object_id` part of the CORBA reference that point to the component and server-side ports. For the

cost of broken opacity of the reference[10] some important benefits stem from this approach:

- any reference to a component instance is all what is needed to identify it unambiguously, hence we can avoid any remote calls for this purpose. Moreover, we avoid calling the `same_component` operation to compare references,
- by including the component identifier in all server-side ports of a component, we avoid `get_component` remote calls to navigate from a port to the component reference.
- a `ServantLocator` can easily decode a unique component's identifier what guarantees low space complexity of redirection. Even if a mobile component visits a certain location many times it will always be bound to the same id in the AOM table.

The presented solution is a simple yet effective way to identify components. For the fact that instance identification is an important quality of a component model we believe that future versions of the CCM model will incorporate it in a similar fashion.

5.4 Summary

In this chapter we have presented the low-level mechanisms that work behind the high-level sensor and effector interfaces. The main focus of this chapter was to provide more details on a very interesting problem of runtime component migration. We have shown most of the issues related to this reconfiguration mechanism such as the problem of component suspension, reaching quiescent state and reconnection. We have also presented how the CCM model may be effectively extended to provide measures to perform runtime migration. With simple extensions to the standard factory interface and component life cycle we offer programmers a convenient framework to develop mobile components. Although our approach does not provide fully transparent mobility it ensures a substantial freedom in how components may be implemented and does not impose constraints on resources and software technologies used.

[10]The CORBA standard assumes opacity of object and component references. This is, however, an area of discussion if this is a proper approach because many successful solutions such as Representational State Transfer (REST) and Internet Communication Engine (ICE) use explicit, human-readable values to refer to remote objects.

The other important mechanism presented is component interception. We have shortly characterized the COPI specification implemented in the course of this work, which is a fine tool for application-level monitoring.

Later in this chapter we have described the problem of component instance identification which is important for the deployment, interception and migration mechanisms. We have shown that instance identification for deployment purposes is separate from lower-level instance identification as required by COPI and runtime migration. Neither CCM nor D&C provide a viable solution to the latter mechanisms. We have proposed a solution that includes UUID-based identifier in every component and port reference. For the cost of broken reference opacity we ensure a simple yet effective instance identification.

Chapter 6

Evaluation of the Framework Building Blocks

This chapter presents the first part of the evaluation of our adaptive deployment framework. It focuses on the lower layer of the framework consisting of: plain deployment infrastructure, runtime component migration and application monitoring.

The principal goal of this chapter is to present effectiveness of the framework building blocks irrespective of a solution used for deployment adaptation. Independently of the kind and quality of an adaptation algorithm used, we show and discuss minimum costs incurred by an application that is being adapted. Another aspect presented in this chapter is conformance of our solution with the D&C specification which was the basis for our plain deployment infrastructure.

6.1 Configuration of the Testing Environment

The software and hardware configuration is listed in Tab. 6.1. In total, the environment included 11 machines hosting 4 different operating systems and was connected with the computer network as shown in Fig. 6.1. LAN 1 was

Table 6.1: The software and hardware configuration of the testing environment.

Computer ID	M_0	M_1	M_2	$M_3 - M_{10}$
CPU family	Intel Core 2 Duo	Intel Centrino Duo	Pentium 4	UltraSPARC-IIe
CPU clock speed	2.66 GHz	2.6 GHz	2 GHz	650 MHz
Memory size	3 GB	3 GB	1 GB	1 GB
Operating system	Windows Vista	Windows XP	Linux 2.6	Solaris 9

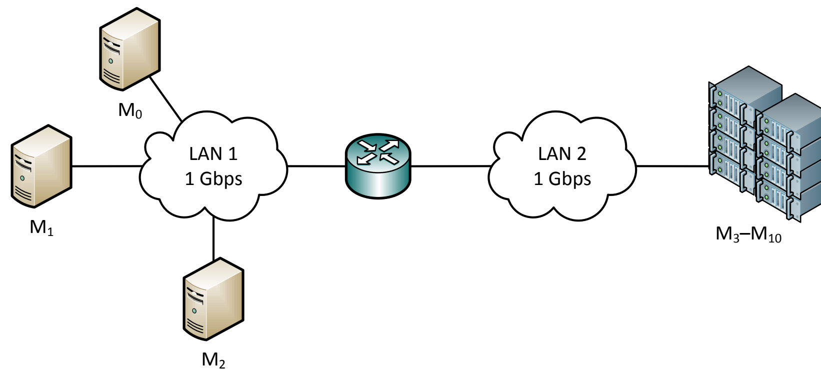

Figure 6.1: The topology of the testing environment.

the Distributed Systems Research Group's local Gigabit Ethernet network, whereas LAN 2 was a dedicated Gigabit Ethernet network connecting the cluster of 8 machines $M_3 - M_{10}$.

6.2 Testing Applications

To test different aspects of the adaptive deployment platform and its building blocks we designed and implemented two testing applications. Our intention was to simulate three common cases: (1) processing-intensive applications, (2) communication-intensive applications and (3) "mixed" applications (partially processing-, partially communication-intensive systems). This allowed us to better assess the influence of tested mechanisms on a distributed system.

6.2.1 Traffic Generator

This application represents a class of communication-intensive applications. The architecture of Traffic Generator is presented in Fig. 6.2. The key component of this application is `Runner` which is declared mobile (the «`Refugee`» stereotype) and can be moved between different `RunnerHome` factories (the «`Refuge`» stereotype). As shown in the figure, `Runner` accepts two types of traffic produced by the `Generator` component: operation calls and events. Received traffic is passed immediately to the `Receiver`. All three components send to the `Logger` information indicating submission/reception of the traffic messages what allows us to monitor any loss or errors in communication. The `Generator` component can be configured to produce traffic of different intensity such as best-effort, random distribution and specified frequency. This enables simulating a broad range of communication-intensive applications.

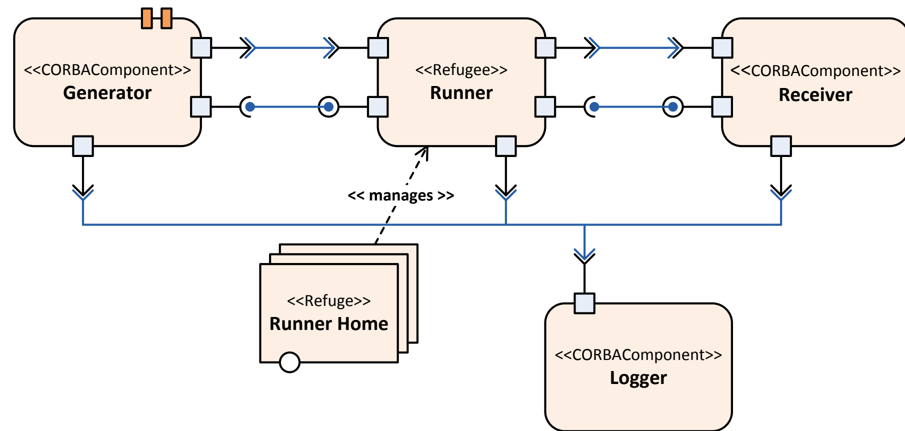

Figure 6.2: The architecture of the Traffic Generator application.

6.2.2 Asymmetric Ray Tracing

This application was implemented to simulate more sophisticated distributed systems. It renders 3D scenes using the ray tracing technique, however, to introduce more asymmetry in the way how the rendering work is distributed throughout the application we designed and developed an additional layer of control. Depending on its configuration, Asymmetric Ray Tracing (ART) can represent a class of processing-intensive, communication-intensive or "mixed" (partially processing-, partially communication-intensive) applications. The ray tracing technique is an embarrassingly parallel problem[1] and, therefore, can be easily implemented as a distributed system. Parallel ray tracing is usually solved using the master-worker architecture where master distributes between workers parts of the image (chunks) to render and then collects the results. This basic form of the ray tracing application exhibits a lot of symmetry, though. Workers are offered parts of the same scene and render chunks of the same size. Even if image chunks have different complexity, on average workers will process comparable number of the chunks and will use comparable amount of CPU and network resources. Therefore, to simulate more asymmetric processing and communication patterns we added to the system the `Controller` component. It controls many `Managers` each of which controls a set of `Worker` components. `Manager` stores its own configuration parameters such as: a number of rendering workers, scene to be rendered, size of the scene, size of an image chunk. This design allowed

[1]In [51, Sect. 7.1] G.C. Fox et al. defined an embarrassingly parallel class of problems as these which spatial structure allows a simple parallelization as no (temporal) synchronization is involved. These problems characterize with the modest node-to-node communication requirements what makes them particularly suitable for a distributed computing implementation on a network of workstations.

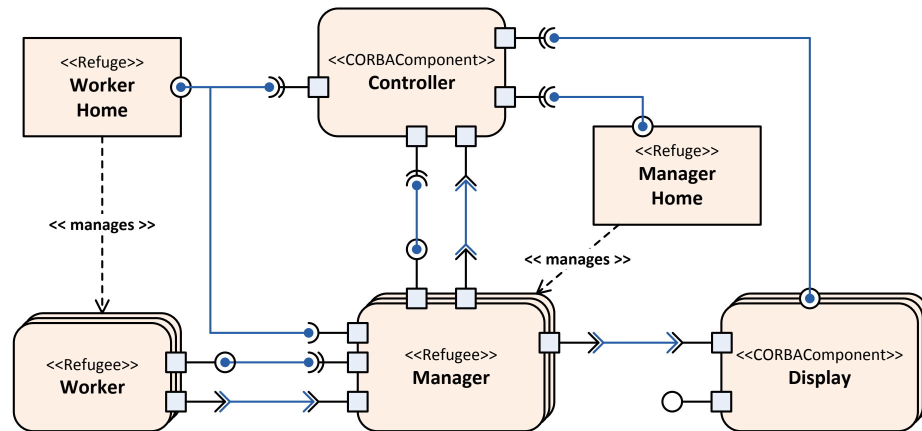

Figure 6.3: The architecture of the Asymmetric Ray Tracing application.

us to simulate applications that have processing-intensive nature (a complex scene, large chunk size), communication-intensive characteristic (a simple scene, small image chunks, many workers), or any combination of these. Moreover, controlling the number of scenes to be rendered by a `Manager`, we can influence how long selected parts of the application are executed.

Figure 6.3 presents the architecture of the ART system. The `Worker` and `Manager` components are declared mobile, the others are immobile. The `Controller` component uses the `Manager`'s provides port to set configuration parameters and receives events about progress of the rendering. `Display` is a GUI component that receives image chunks from managers and presents them to a user. `Controller` uses `Display` interface to dynamically bind `Manager` instances with `Displays`. Additionally, it uses `ManagerHome` and `WorkerHome` factories to create a number of `Manager` and `Worker` components as defined by a user.

6.3 Evaluation of the Plain Deployment Infrastructure

The aim of this section is twofold. First, to show conformance of our plain deployment infrastructure with the D&C specification which was the foundation for our ADF framework presented in this work. Second, to demonstrate effectiveness of the infrastructure when performing basic application deployment tasks. Apart from only conformance issues, the first part also verifies usability of the D&C specification for deployment in open heterogeneous and distributed systems. We clarify why some parts of the specification were not implemented in our framework.

The second part — performance evaluation — is important because the plain deployment infrastructure is used not only for the initial application deployment but also whenever its reconfiguration is requested. Although the deployment activities are performed in background and do not directly influence application they may affect the adaptation process resulting in degradation of application performance. Later in this chapter we show relation between the plain deployment infrastructure and the reconfiguration mechanism.

6.3.1 Conformance to the D&C Specification

Developing an implementation fully conformant to the D&C specification requires a lot of effort. The intention of this research was not to implement the complete specification but rather verify what elements are needed to enable adaptiveness in the deployment process. For this reason we have not developed all the defined functionality, nonetheless, the implemented functionality is as much compliant with D&C as possible.

The D&C specification defines segmentation of the deployment model in the two following dimensions:

1. *Data Models vs. Management Models* — that distinguishes between a model of descriptive information and the model of runtime entities that process that information,
2. *Component Software vs. Target vs. Execution* — that separates the deployment model on three segments representing: (1) an application software package, (2) a target execution environment running the software, and (3) an execution infrastructure used to deploy and start software in a target environment.

These dimensions define six different *pages* of the model [100, Sect. 7.1.3]. Our adaptive framework makes use of three pages referring to the Data Models dimensions exactly as they are defined in the specification. Conformance to the other three pages is presented in the remainder of this section.

Target Management Model

The key functionality defined in this model is accessible through the `TargetManager` interface. Our infrastructure implements two of the five operations defined i.e.: `getResources` and `updateDomain`. The manager reads a static description of a target execution domain from an XML file (see Appx. B)

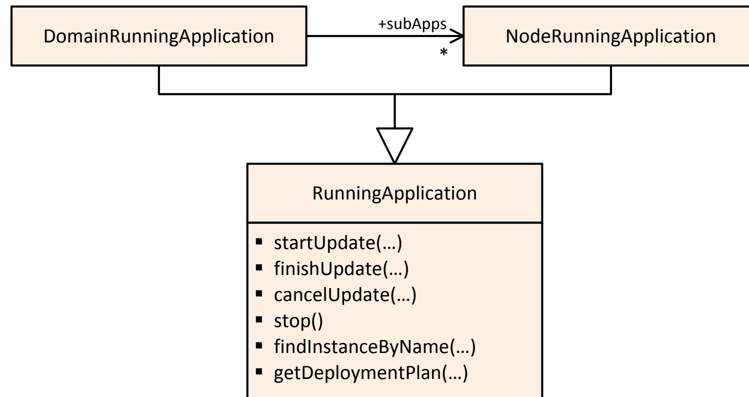

Figure 6.4: Extensions of the *Execution Management Model* related to runtime application reconfiguration.

and makes it available through `getResources`. The `updateDomain` operation allows for updating this static information with the current monitoring data.

The other three operations of the `TargetManager` interface are related to resource reservation management. We have not implemented this functionality of the manager because we accepted the assumption that our adaptive deployment framework works in the best-effort manner. This approach is more appropriate for adaptation in open distributed environments as it avoids the need of specifying intricate resource reservation functions.

Execution Management Model

This model is the fundamental and the most complex part of the D&C deployment specification. We implemented all its relevant elements[2] together with our extensions that enable runtime reconfiguration. The infrastructure can be accessed either using the standard D&C interfaces or through our proprietary interfaces. The model was extended with three new entities: `RunningApplication`, `DomainRunningApplication` and `NodeRunningApplication` (Fig. 6.4). The structure of this extension reflects the existing structure of an application in D&C which separates the model between a global (domain) and local (node) infrastructure. A running application can be updated in runtime according to a deployment plan provided to the `startUpdate` operation. Update has been split on two stages what follows two-phase initialization scheme of D&C-based applications.[3]

[2]The distributed `Logger` facility is the only missing D&C entity in our implementation. It is not properly mapped by *PSM for CCM* transformation and, therefore, its exact form remains unclear. Instead, we use local logging facility offered by Java API.

[3]The full definition of the changes to the D&C specification is included in Appendix A.

Again, the elements of the execution management model that we did not implemented are related to resource management. Therefore, our DomainApplicationManager is not connected to the TargetManager and does not reserve resources when performing deployment. Moreover, we did not realize the getDynamicResource operation included in the NodeManager interface that allows for retrieving values of dynamic satisfier properties associated with a managed node. Our framework does not use this functionality because information about dynamic properties are collected directly from monitoring sensors. In this way we avoid unnecessary delays and can better control the how monitoring of the execution environment is performed.

Component Management Model

The D&C Component Management Model defines only a single RepositoryManager interface. The primary goal of the manager is to support an application packager, software provider and end-user in maintaining collection of software packages. Our ADF framework does not focus on application development and publishing, therefore, the implementation we developed is limited to only these operations that are necessary for the first-stage planner to prepare an initial deployment plan. It includes package installation and searching.

Additionally, our repository manager provides means to retrieve component artifact files. For this purpose the specification suggests using URL references that can be easily included in XML descriptor files, hence the manager contains a simple HTTP server that enables such referencing.

Overall Conformance Statistics

Table 6.2 presents quantitative view on the conformance of our implementation with the D&C specification. In the table we included all three management models and presented the number of implemented entities and operations comparing to all entities and operations defined in the models.

As may be seen, the most important part for the specification — *Execution Management Model* — is implemented in more than 90%. The unimplemented functionality is related to resource reservation which is in contrary to the best-effort resource management approach we accepted. The proposed solution offers three new entities concerning runtime application reconfiguration. The extensions in the *Component Management Model* denote the provided HTTP interface.

Table 6.2: The number and percentage of the entities and operations defined in D&C that are implemented by our deployment infrastructure.

Execution Model	No. of entities	No. of operations
Component Management Model	1/1 (100%)	5/8 (63%)
+extensions	0	+1
Target Management Model	1/2 (50%)	2/7 (29%)
Execution Management Model	8/9 (89%)	13/14 (93%)
+extensions	+3	+5
Total	**10/12 (83%)**	**20/29 (69%)**
+extensions	**+3**	**+6**

6.3.2 Performance of the Deployment Infrastructure

In this section we focus on performance aspects of the implemented plain deployment infrastructure. As already mentioned, effectiveness of the infrastructure has only indirect influence on application, nevertheless it may impact on *agility of the adaptation process*.[4] In this work we use the component migration mechanism as a basis for application adaptation and plain deployment is a prerequisite for the migration. Before component migration can take place the plain deployment infrastructure is used to prepare an execution environment according to a new deployment plan. Two issues are important in this case: (1) influence of the preparation step on application performance, and (2) the time required to perform the preparation. To host a mobile component a target execution node needs to run component's factory and this in turn requires a proper container and component server to be running, too. If any of these entities need to be deployed on a target node depends on whether the node runs the hosting infrastructure already and whether its configuration is consistent with the required by the mobile component. In the worst case all of the entities are deployed.

The exact influence of the reconfiguration preparation step on application performance depends on the number and location of nodes being involved. When the preparation affects unoccupied nodes it causes only minor disruption for the network. More significant impact is made, however, when there is a need to deploy the hosting infrastructure on nodes which already execute some application components. We observed such a case and measured its influence on performance of the ART application. In this test we ran all the application components on a single node and requested a number of different updates. Table 6.3 shows the measurements for different settings.

[4]As mentioned earlier in Chap. 5 on page 107, adaptation agility is a property of an adaptive system that determines speed and accuracy with which it detects and responds to changes in its execution context.

Table 6.3: Time (in seconds) required to complete a single run of the ART application with and without deployment updates.

Target node	No update	10 deployment updates		
		factory	container	comp. server
M_0	66.1(±2.2%)	66.0(±1.2%)	64.0(±2.7%)	72.8(±1.9%)
M_2	252.8(±4.4%)	265.5(±1.2%)	256.6(±2.3%)	297.8(±1.1%)

As expected, the most disruptive case was when preparation included instantiation of a component server, container and factory (the last column). Deployment of a new component server involves creating a new JVM process in the operating system and this is the most expensive part of the whole preparation. For cases when only a container and/or factory was deployed, degradation of performance per one deployment update was negligible (below 1‰). Somewhat surprising was the fact that running a container and factory (fourth column) was faster than creating a new factory within an existing container (third column). After a closer analysis it appeared that during the factory-only update the Java VM performed additional garbage collection which did not occur when the factory and container update was performed. Another unexpected behaviour we noticed when comparing results on machine M_0. Running the container and factory update rendered better results than leaving the application untouched. We tested this case many times and the only explanation we could find was that the OS scheduler on Windows Vista machine M_0 promoted the testing process which was in the foreground at the time of the test.[5]

Overall, even if the most significant impact on application performance is caused by running fresh component server instances the need for separate servers on a node already hosting application components is relatively rare. In most cases it is enough to create a new factory or, if a component has some specific configuration requirements, to create a separate container within the existing component server. Consequently, influence of the preparation step on application performance can be considered as only minor.

The second problem related to the reconfiguration preparation is its completion time. We measured time required to perform a deployment update of the ART application. Figure 6.5 shows the target environment which consisted of three hosts: M_{10} and two others named A and B. The exact assignment of the execution nodes to the presented target hosts A and B is shown in Tab. 6.4. Initially, only the `Controller` component was deployed. Later, during the tests, we updated this deployment with three

[5]This is in accordance with *Inside the Windows Vista Kernel: Part 1* by M. Russinovich `http://technet.microsoft.com/en-us/magazine/2007.02.vistakernel.aspx`

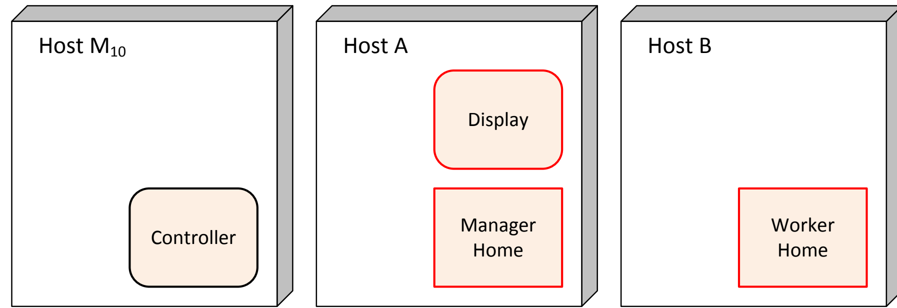

Figure 6.5: Target environment used for testing time required to prepare reconfiguration of the ART application. Host M_{10} was initially deployed with the `Controller` component. Hosts A and B were updated with three other components.

Table 6.4: Time (in seconds) required to update deployment of the ART application depending on the target execution nodes.

Target host A	Target host B	`startUpdate`	`finishUpdate`	Total time
M_0	M_0	2.659(±1.5%)	0.020(±13.0%)	2.679(±1.6%)
M_2	M_2	4.299(±1.2%)	0.035(±4.6%)	4.334(±1.3%)
M_0	M_2	5.795(±1.3%)	0.032(±11.9%)	5.827(±1.3%)
M_0	M_3	11.718(±1.3%)	0.024(±26.5%)	11.742(±1.3%)
M_2	M_3	13.400(±0.7%)	0.037(±6.6%)	13.437(±0.7%)
M_3	M_3	13.165(±1.1%)	0.109(±2.8%)	13.274(±1.0%)
M_3	M_0	14.850(±1.6%)	0.138(±2.1%)	14.988(±1.6%)
M_3	M_2	15.678(±1.0%)	0.133(±4.0%)	15.811(±1.0%)
M_3	M_4	21.741(±0.7%)	0.142(±7.6%)	21.883(±0.7%)

other components as shown in the figure. Table 6.4 includes measurements of deployment time for different target hosts A and B. The process comprised of two steps presented in the table: `startUpdate` and `finishUpdate`. It is clearly visible that the vast amount of time (about 97%) was consumed by the `startUpdate` operation. In this test it included full preparation (i.e. instantiation of component server processes, containers and homes) because the target nodes A and B were not hosting any of the application components yet.

As shown later, the time required to prepare the execution infrastructure is more than three orders of magnitude longer than the time required to perform migration. Unfortunately, the longer the time is, the lower responsiveness of the reconfiguration mechanism is and the less agile adaptation can be. Figure 6.6a presents a case when preparation and reconfiguration actions are

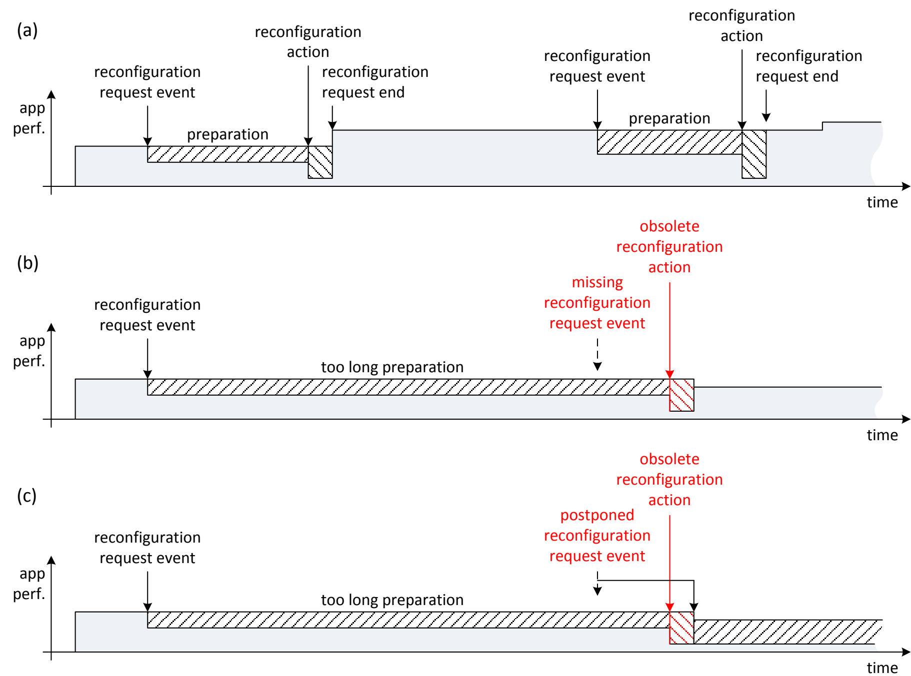

Figure 6.6: Influence of the reconfiguration preparation step on adaptation agility.

fast enough to be accomplished before the next application reconfiguration attempt. When the deployment infrastructure receives a reconfiguration request event, it starts preparation of the infrastructure according to the provided plan. Only after this step the adaptation infrastructure can perform the actual reconfiguration of the application (i.e. migration of components). Depending on the quality of the adaptation algorithm used, it should result in some performance gains. However, in the case when preparation is longer than the interval between decisions coming from the adaptation algorithm, some undesirable effects will occur. First, depicted in Fig. 6.6b, is the problem of missing reconfiguration attempts. It causes an obsolete reconfiguration action to be done despite the fact that the adaptation algorithm requests a new, possibly contradictory, action to be performed. This may lead to deterioration in application performance. Second, *reconfiguration storms* is shown in Fig. 6.6c. By postponing reconfiguration requests it causes the system to be constantly reconfigured. When designing the adaptation algorithm we took both these issues into account.

6.3.3 Possible Extensions

Two important aspects are not at all addressed by our deployment infrastructure: security and transaction-awareness. Both are crucial for produc-

tion use of the platform. Security in deployment is required to ensure basic credibility of the platform and to protect users from spreading viruses and other malicious software. Transaction-awareness provides measures to cope with failures during distributed deployment. Developing a secure and transaction-aware deployment infrastructure is, however, a large and complex undertaking and was deliberately omitted from this work.

6.4 Performance of the Migration Mechanism

In this section we assess performance of the migration mechanism considering three aspects. First is the time required to carry out migration of a component. Second is the influence of this mechanism on application performance. Third is the comparison of performance between the original implementation of the CCM platform (OpenCCM) and the platform extended with support for runtime component migration.

6.4.1 Effectiveness of the Migration Mechanism

In this test we used five identical Sun Blade machines and the Traffic Generator application. The deployment of the testing application is presented in Fig. 6.7. In this test the `Runner` component was being moved by the `MigrationService` component 2000 times starting from the host M_3 (the home location) through M_4, M_5, etc. To focus merely on migration aspects in this test and avoid disruptions caused by infrastructure preparation, on all hosts the `RunnerHome` factory was deployed a priori. Moving the `Runner` component between identical hosts we could observe how performance of component migration depended on location of adjacent application components.

Table 6.5 shows the gathered results with separation on subsequent migration steps. As expected, the quickest response (67.5 ms) was whenever the home location (host M_3) was directly involved in `Runner` migration. This is mainly due to faster reconnect step which involves the `refugee_moved` and `refugee_update` operations.[6] The former is called on the source `Refuge`, whereas the latter on the home `Refuge`. In the case of $M_3 \rightarrow M_4$ jump, the M_3 host was both the source and home `Refuge`, hence `refugee_update` invocation was avoided. However, in case of the $M_7 \rightarrow M_3$ jump reconnection was faster due to collocation of the home `Refuge` and the migration service component that controlled `Runner` migration. This collocation was also the reason why the accept step (always called on the target factory) was the fastest for the $M_7 \rightarrow M_3$ jump. Remaining locations exhibited almost

[6]This was discussed earlier in Sect. 5.1.2 on page 112.

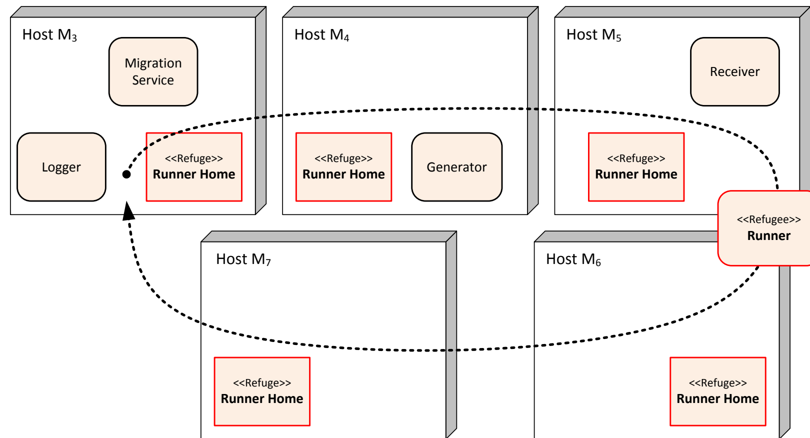

Figure 6.7: The deployment of the Traffic Generator application among five Sun Blades servers. The dotted line shows the migration path of the `Runner` component.

Table 6.5: Time (in milliseconds) required to perform subsequent migration steps when moving a component between the indicated locations.

Move	Freeze	Accept	Reconnect	Total time
$M_3 \rightarrow M_4$	14.4(±45%)	35.6(±26%)	17.5(±31%)	67.5(±21%)
$M_4 \rightarrow M_5$	14.0(±52%)	36.6(±27%)	29.7(±29%)	80.4(±22%)
$M_5 \rightarrow M_6$	13.3(±48%)	40.0(±28%)	30.0(±30%)	79.8(±22%)
$M_6 \rightarrow M_7$	14.9(±51%)	35.6(±29%)	29.8(±35%)	80.3(±24%)
$M_7 \rightarrow M_3$	13.8(±54%)	31.0(±27%)	23.6(±30%)	68.4(±23%)

identical performance (about 80.0 ms) that was nearly 20% worse than the fastest $M_3 \rightarrow M_4$ jump.

Overall, the test shows that migration of a simple stateless component requires about 70 ms. This time factor is important when considering deployment adaptation because it causes interruption in the component functioning what, in turn, may influence the whole application. Depending on the application architecture interruption of a single component can have very different ramifications on application performance. In this example the `Runner` component is the central point between `Generator` and `Receiver` and, therefore, each migration of this component resulted in interruption to the whole application.

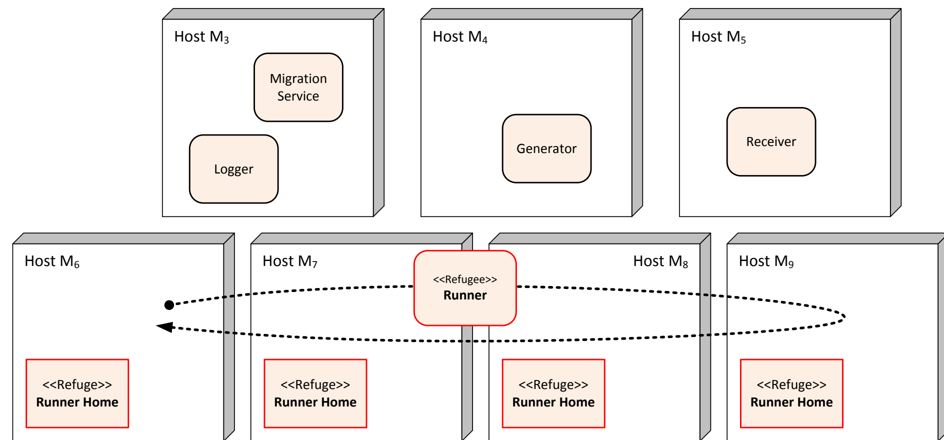

Figure 6.8: Deployment of components when testing influence of migration on call performance.

6.4.2 Influence of Migration on Communication Performance

The goal of this test was to verify how migration influences call performance of a mobile component. `Generator` sent to the `Runner` component as many events as it was able to process and we observed how location of `Runner` affected message processing. The test was conducted using four Blade machines as component refuges and three other Blades to control the migration and generate traffic. The exact deployment of the testing application is depicted in Fig. 6.8. Separation between the `RunnerHome` factories and the rest of application components was intentional to create the same conditions for all `Runner` jumps. As shown in the figure, `Runner` was being moved in a cycle: $M_6 \to M_7 \to M_8 \to M_9 \to M_6$ starting from home location M_6.

The results presented in Fig. 6.9 show the average number of event transfers per second and their RTT. The test was repeated in a cycle and in the figure we highlighted one such cycle with four periods when component was executing on machines: M_6, M_7, M_8 and M_9 respectively. As shown in the figure, `Runner` executing in its home location performs nearly twice as fast as being located in a non-home location. This was because the `Generator` component used the original object reference of `Runner` and when it was executing at the home location no additional redirection was required.[7] It is also visible that consecutive movements of the `Runner` component did not introduce any degradation in the communication performance. This clearly indicates that there is no chain of reference built while a component is moving and proves correctness of our implementation.

[7] Usually, this behaviour would not be observed as ORB caches the most recent redirection to a specified object reference. Unfortunately, the ORB implementation we used in the tests (OpenORB) caused problems when redirection caching was turned on.

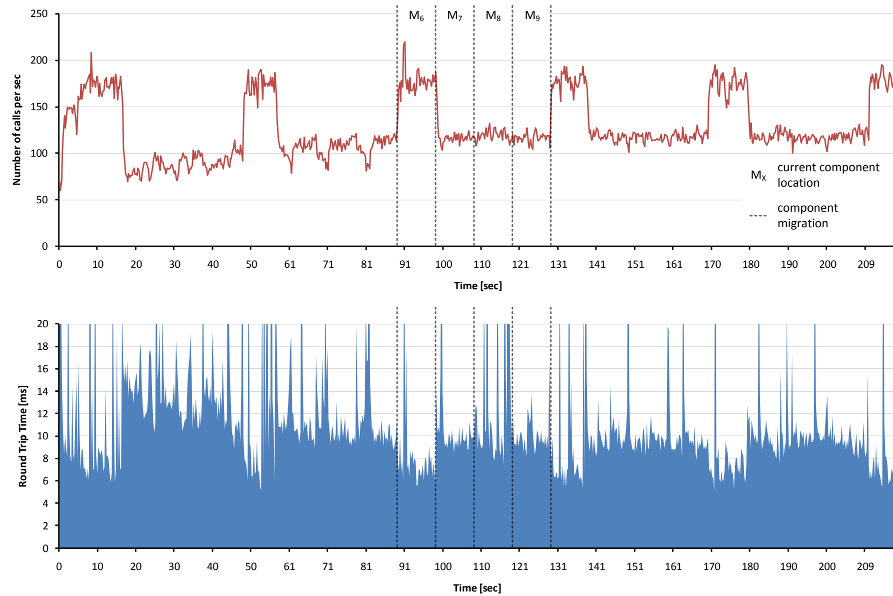

Figure 6.9: Influence of migration on the number of operation invocations per second and invocations' RTT.

6.4.3 Influence of Migration on Processing Performance

In order to measure influence of component migration on application performance we used the Asymmetric Ray Tracing application. We simulated communication-intensive and CPU-intensive applications by controlling the size of an image chunk being processed by a Worker.[8] In this test there was exactly one Manager and one Worker running. To assess migration costs we used a single controller machine and a cluster of four identical Sun Blade servers. Figure 6.10 presents deployment of the ART application in this execution environment. By moving Worker between the Blades we did not expect any gains in processing performance because of the same hardware and software configuration. We could rather observe decrease in performance incurred by the application due to component migration.

Table 6.6 shows the measurements in three cases. First, when Worker was running in its home location (host M_3) and no moves were done. Second, when Worker was running in a non-home location and no moves were done. Third, when the Worker component was moving between all Blade hosts in a cycle, starting from the home location. In this case component migration was performed 20 times, once per 8 seconds. When moving, Worker was running 25% of its execution time in the home Refuge and 75% in non-home Refuges. After the last move and until the end of each test it was

[8]The larger the chunk size is, the more processing-intensive the application becomes.

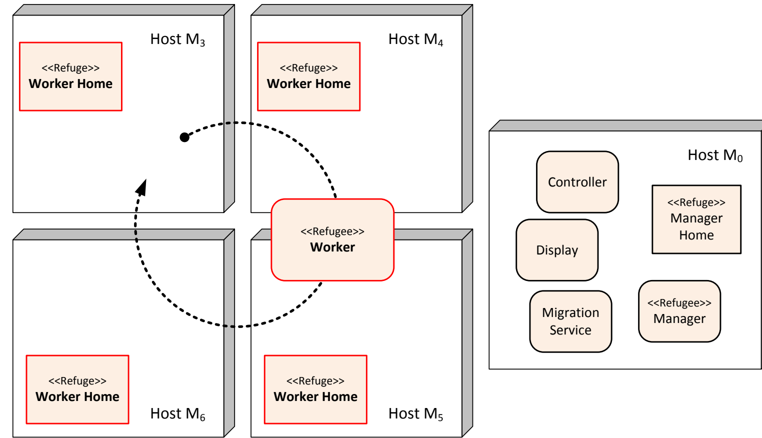

Figure 6.10: Deployment of components during the test that verified influence of component migration on its processing performance.

Table 6.6: Time (in seconds) required to perform a ray tracing task depending on chunk size, `Worker` location and its mobility.

Chunk size	at home no moves	at non-home no moves	20 moves 1/8 Hz
4×4	516.4(±1.2%)	612.7(±2.9%)	557.4(±5.4%)
8×8	267.8(±1.4%)	301.1(±0.7%)	292.1(±3.6%)
16×16	181.3(±1.0%)	183.1(±1.0%)	199.4(±0.6%)
32×32	154.3(±0.7%)	153.4(±0.4%)	171.2(±1.1%)
64×64	143.1(±0.8%)	139.3(±1.4%)	160.4(±1.9%)
128×128	137.2(±0.6%)	132.9(±1.9%)	211.0(±7.7%)

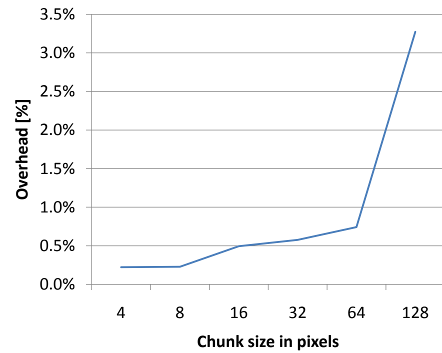

Figure 6.11: Performance overheads incurred by the ART application related to a single `Worker` move.

again running in the home location. Given this and the time of the first and last move we calculated the absolute deceleration and average decrease in performance of the ART application per single `Worker` move. Results are presented in Fig. 6.11. In most cases the overheads were very low (below 1%), however, for large image chunks, i.e. CPU-intensive components, we observed significant increase in processing time and a sudden surge of the overheads.

When moving, the `Worker` component does not store any parts of a rendered image chunk and after activation in a new location it starts rendering the recently rendered chunk from the beginning. The main drawback of this simple implementation can be noticed when time required to calculate a single image chunk increases. If it is comparable to the interval between two

subsequent moves, the application will incur high overhead (for chunk size 128x128 and interval 8 seconds). For larger chunks of shorter intervals the application may even be blocked. We deal with this problem by rejecting too frequent migration attempts — migration of the `Worker` component will be rejected until it produces at least one image chunk.

During migration a component is passivated and requested to store its state. As we implemented the weak migration approach,[9] it is rarely possible to store the component's state exactly as it is when the passivation event occurs. Instead, if the component is involved in some operations when the passivation is requested, it may follow one of the three solutions. First, if it is acceptable, it may interrupt processing of the operations immediately and loose some state. Second, it may postpone passivation event until all the operations are finished or a restore point is reached.[10] Third, it may reject migration request. Figure 6.12 depicts these three cases comparing them with the strong migration approach. For strong migration (Fig. 6.12a), execution of a component can be resumed exactly from the state of passivation (m_n) and, therefore, the only overhead is the time required to perform the actual component migration. Conversely, for weak migration the three aforementioned cases are possible. Figure 6.12b depicts the case when passivation is conducted immediately on migration request (m_1, m_2, and m_3) but execution is resumed from the most recent restore point (s, b and c). In this case overhead is twofold: related to migration itself and related to rerunning component from the restore to passivation point ($s \to m_1$, $b \to m_2$ and $c \to m_3$). Second and potentially more effective solution is presented in Fig. 6.12c. In this case migration is postponed until the next restore point is reached ($m_1 \to a$, $m_2 \to c$ and $m_3 \to d$) and the application incurs overheads of component migration only. The delay of the move operation is sustained by the entity performing component migration. The third case, which is followed by our `Worker` component, is presented in Fig. 6.12d. It is similar to the case (a) but migration requests are rejected until a component reaches at least one restore point. This avoids component starvation if migration is too frequent because the component can guarantee that it will progress towards at least one restore point before it allows any movement.

In the case of `Worker` that rendered large image chunks (128x128) the surge in overhead presented earlier was connected with the state loss. The time needed to accomplish rendering with migration was nearly twice as

[9] *Weak* component migration is the approach that moves entity's code and state. It is limited when compared to *strong* migration that, additionally, transfers current program counter. Strong migration allows for resuming processing exactly in the stage it was interrupted, whereas weak migration requires explicit passivation. At the middleware level, however, access to the program counter is limited and, therefore, implementing the strong migration approach seems to be hardly possible. More about strong and weak migration is explained by A. Fuggeta et al. in [53].

[10] We discussed some of these issues in Sect. 5.1.4 on page 119.

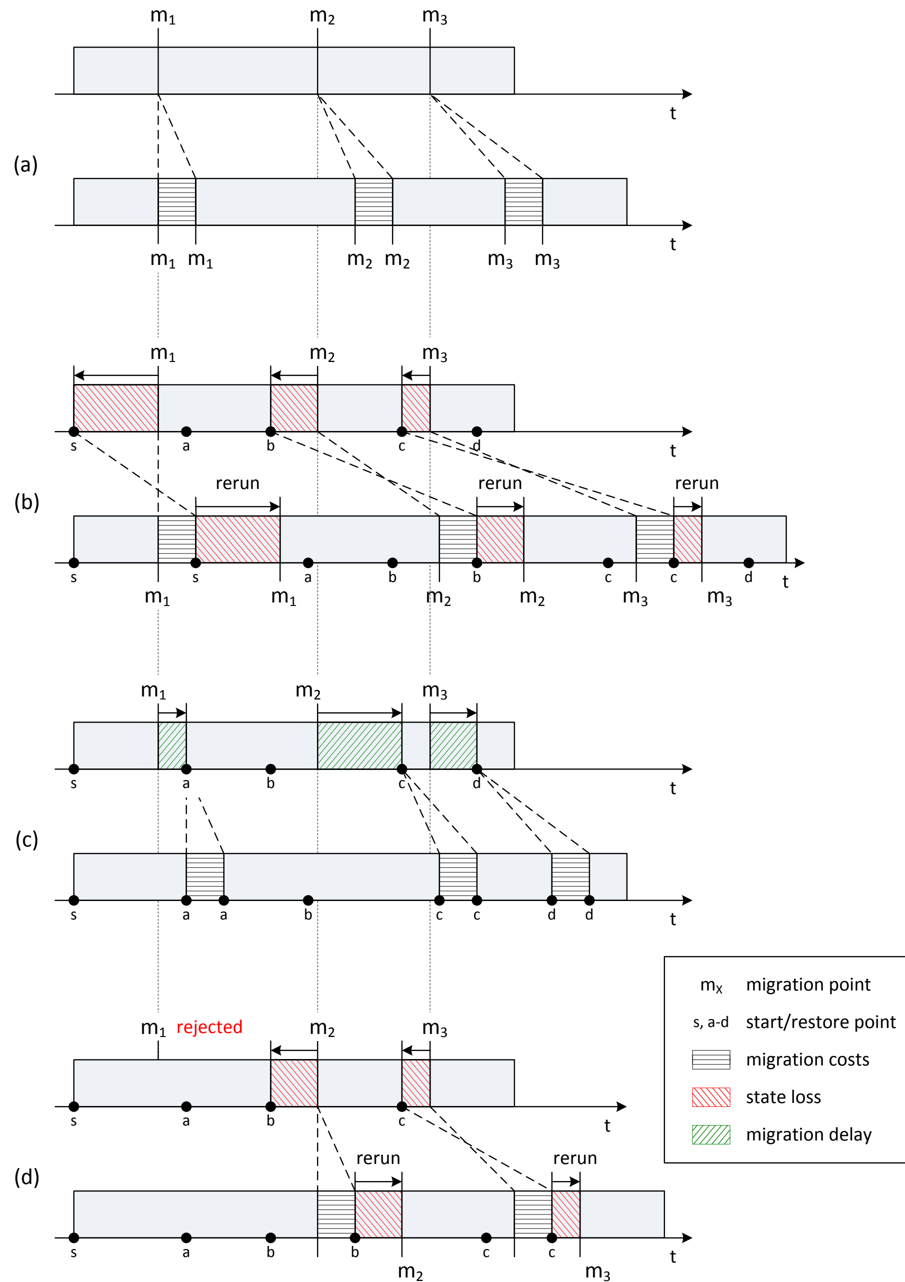

Figure 6.12: Influence of migration on application processing time. The upper axes show original execution time line. The lower present execution with migration; (a) strong migration does not incur any loss of state; (b) state loss during weak migration caused by reverting to the recent restore point; (c) avoiding state loss during weak migration by its postponing until a restore point is reached; (d) avoiding processing starvation during weak migration by rejecting too frequent migration requests.

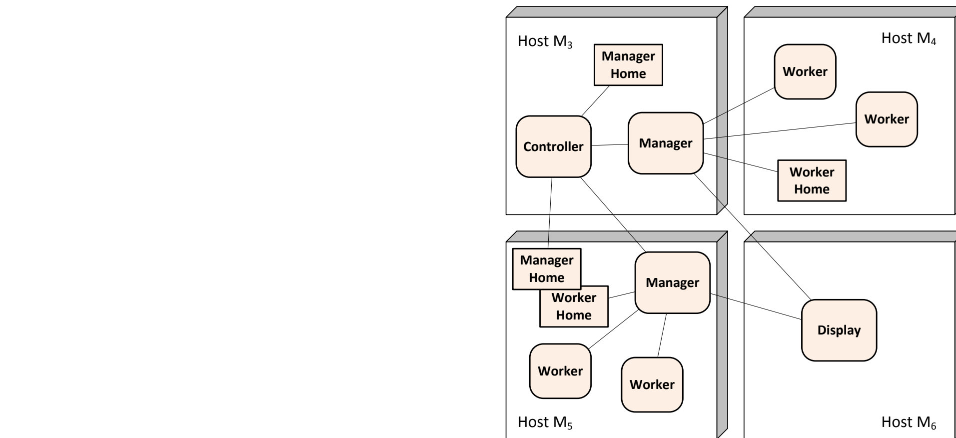

Figure 6.13: Deployment of the Asymmetric Ray Tracing application to measure overheads of the our migration-aware CCM platform.

long as in the case with no component migration. It means that the time needed to render a single image chunk must have been longer than 8 seconds and moves, if not rejected, were related to loss of almost the whole rendered chunk. Developers should take these problems into account when programming support for migration of their components because poorly designed passivation and state checkpointing can yield disappointing results.

6.4.4 Overhead of the Migration Infrastructure

The purpose of this test was to measure overhead related to running our mobile-aware infrastructure when compared to the original OpenCCM implementation. To assess the overhead we used the Asymmetric Ray Tracing application in the deployment settings presented in Fig. 6.13. This time the application was executed twice: (1) using the original OpenCCM platform, (2) using the modified version of this platform supporting mobile components. In this test no component migration was performed but we simply compared application performance for three application settings. First simulated comm.-intensive applications (rendering a number of simple scenes using small image chunk size), second simulated CPU-intensive systems (rendering complex scenes with large image chunks). The last was a mixed configuration where parts of the application were more comm.-intensive and parts more CPU-intensive.

Table 6.7 presents execution times for different application settings. As may be seen, our extension of the CCM infrastructure does not introduce any performance overheads. Conversely, it exhibits small performance gains which are result of improvements in thread management.

Table 6.7: Execution time (in seconds) of the ART application for the original and mobility-aware OpenCCM platforms.

Application settings	Original OpenCCM	Mobility-aware OpenCCM	Overhead
comm.-intensive	472.1(±0.3%)	463.0(±0.3%)	−1.9%
CPU-intensive	304.3(±0.1%)	291.9(±1.0%)	−4.1%
mixed	515.8(±0.8%)	516.7(±1.8%)	0.2%

6.5 Overhead of Monitoring Infrastructure

Monitoring infrastructure is an important part of any adaptive framework. It supplies the adaptation engine with crucial, runtime data that convey information about the current state of an adapted system. Unfortunately, monitoring almost always introduce performance degradation to the monitored system and hence it is important to measure and minimize its influence.

One of the advantages of our framework is the ability to install and uninstall monitoring sensors on demand in runtime. This allows decreasing the overheads when some monitoring information is not available for a certain type of applications or not required for an adaptation algorithm used. Note, however, that at the middleware layer monitoring sensors often require proper instrumentation of the execution environment. While our framework allows uninstalling sensors, it cannot remove the instrumentation which also may have negative influence on application performance. For example, to assess intensity of communication between components we implemented the `LinkSensor` component that requires the COPI infrastructure to be running. This infrastructure needs to be enabled in a container and even if we uninstall the `LinkSensor`, it will still be running and introduce some overheads for call performance.

In the following tests we measured influence of both: the instrumentation layer and sensors. For CIM-based sensors we tested environment with and without the WBEM infrastructure running and with and without `CPUSensor` and `MemorySensor` installed. For `CommSensor` and `LinkSensor` we tested environment with and without the COPI infrastructure running and with-/without sensors installed. More limited were tests of `HomeSensor` because it uses a proprietary OpenCCM home listener infrastructure that cannot be easily uninstalled. Therefore, in the case of `HomeSensor` tests, we only compare application performance with and without the sensor installed.

To assess CIM- and COPI-based sensors we used the Asymmetric Ray Tracing application. It was deployed as shown previously in Fig. 6.13 and was applied the same three configuration settings that simulate communication-

Table 6.8: Execution time (in seconds) of Asymmetric Ray Tracing depending if the WBEM infrastructure and CIM-based sensors were enabled.

Application settings	WBEM disabled	WBEM enabled				
		no sensors	CPU-Sensor	Memory-Sensor	CPU- and MemorySensor	Null-Sensor
CPU-intensive	290.0(±0.8%)	290.6(±0.6%)	349.6(±0.8%)	349.9(±1.5%)	415.0(±0.5%)	293.3(±1.6%)
comm.-int.	429.7(±0.6%)	430.1(±1.0%)	521.3(±0.7%)	521.3(±1.3%)	611.0(±0.3%)	438.2(±0.4%)
mixed	514.7(±0.8%)	516.8(±1.2%)	616.7(±0.5%)	617.1(±1.5%)	735.9(±0.6%)	526.8(±0.8%)

Table 6.9: Execution time (in seconds) of Asymmetric Ray Tracing depending if the COPI infrastructure and COPI-based sensors were enabled.

Application settings	COPI disabled	COPI enabled			
		no sensors	Link-Sensor	Comm-Sensor	Link- and CommSensor
CPU intensive	290.0(±0.8%)	290.9(±0.2%)	293.6(±0.6%)	299.6(±0.7%)	308.5(±1.0%)
comm. intensive	429.7(±0.6%)	463.6(±0.6%)	476.3(±0.7%)	471.4(±0.4%)	480.9(±0.4%)
mixed	514.7(±0.8%)	529.7(±1.0%)	527.0(±0.9%)	543.4(±1.9%)	543.0(±1.1%)

intensive, processing-intensive and mixed (partially comm.-, partially CPU-intensive) applications. Table 6.8 shows the results collected for CIM-based sensors. Running the WBEM infrastructure had only a minor influence on application performance: in case of the CPU-intensive application setting there was no noticeable overhead and for the other two settings a decrease of 1–2% was measured. However, installing and running sensors introduced substantial penalty for the monitored application which was executing about 20% longer if one sensor was installed. Installing two sensors on a node caused over 40% increase in execution time irrespective of the application settings. Surprisingly, such a high overhead was not result of the sensor operation but rather activation of the WBEM infrastructure. When a sensor was not installed in the system the infrastructure remained dormant. However, installation of the sensor activateed a WBEM agent which consumeed substantial amount of resources.[11] To verify this we implemented the `NullSensor` component that differs from the `CPUSensor` only in that it does not call the WBEM infrastructure but reports random data instead. Results show that overhead caused by the `NullSensor` implementation is as low as 1-2% irrespective of the application settings.

Much different are measurements gathered for COPI-based sensors. Table 6.9 shows that installation of the COPI infrastructure has only minor influence on application performance unless it is a communication-intensive application. In the tested settings, for the CPU-intensive and mixed settings the COPI infrastructure caused as low as 1–3% decrease in performance,

[11]We performed the tests mainly on low-power Sun Blade Servers where running the WBEM infrastructure consumed a lot of memory and CPU resources.

Table 6.10: Total migration time (in milliseconds) of the Runner component with HomeSensor disabled and enabled.

Application settings	HomeSensor disabled	HomeSensor enabled	Overhead
1000 jumps, best-effort	63.7(±20.0%)	68.7(±19.7%)	7.8%

whereas for the communication-intensive setting overhead was about 7%. Similarly, installation of COPI-based sensors had only minor influence on non communication-intensive applications (1–4% overhead) which increased to about 9% for the communication-intensive settings. Substantial penalty for the latter settings was result of running the COPI infrastructure that intercepted every invocation between components. The higher the rate of communication was the more frequently interception occured and higher overhead was.

In order to measure overheads caused by the HomeSensor we used the Traffic Generator application with migration infrastructure enabled. Moving a component between containers involves creating and destroying its consecutive incarnations, therefore, it is a good testing scenario for the HomeSensor. In this test we moved the Runner component 1000 times between four machines as fast as possible. As shown in Table 6.10, overhead for a single component movement (one create and one destroy operation) was about 8%. It is worth noting, however, that overall overhead for application performance depends on how often application creates and destroys component instances.

6.6 Summary and Conclusions

This chapter presented evaluation of the three building blocks of our adaptive deployment framework: the plain deployment infrastructure, monitoring and component migration mechanisms. The deployment infrastructure implements the most important parts of the D&C specification leaving unimplemented elements related to resource reservation and management. We adopted the best-effort approach to the resource management rather than strict resource management proposed by D&C. Later, we showed that although effectiveness of the plain deployment does not influence application performance directly it may affect agility of the adaptation process. This, in turn, may cause indirect performance degradation and some undesirable effects such as reconfiguration storms. To avoid them an adaptation mechanism needs to take effectiveness of the plain deployment infrastructure as an important input factor for system reconfiguration.

When testing the runtime component migration mechanism, we measured the absolute time required to perform a single move of a simple component. We also presented results that indicate correctness of our implementation. Moving a component many times between different machines did not affect its performance. Further, we showed that component migration may have only minor influence on application performance esp. when a component does not involve long processing. For CPU-intensive components the key issue is to effectively manage their state on passivation. Unfortunately, at middleware layer we cannot afford for the strong migration mechanism and, using the weak approach, components are susceptible for state loss. We discussed three possible solutions for this problem: reverting to the recent restore point, postponing a migration request and rejecting a migration request. Which of these is the most appropriate for a particular component and application is difficult to answer and, therefore, we leave this decision to programmers. Our framework provides, however, all required tools to easily follow any of these solutions.

Lastly, we focused on monitoring infrastructure and presented influence of environment and application monitoring on application performance. The major overhead we measured was for CIM-based sensors that used WBEM infrastructure to collect the data. However, the source of the problem was in high requirements of the Solaris' WBEM infrastructure rather than in sensor implementation itself. The costs related to running COPI-based application monitoring was relatively low unless the application exhibited comm.-intensive characteristic. Finally, we measured overhead of `HomeSensor` that enables monitoring of creation and destruction of component instances. Although results show a substantial overhead, its actual influence on application performance largely depends on how often the application creates and destroys component instances. Overall, to minimize monitoring costs incurred by an application the presented infrastructure offers on-demand installation and uninstallation of sensors. This may be used to attach sensors by an adaptation mechanism in runtime.

Chapter 7

Adaptive Deployment with Force-Directed Algorithms

In the previous chapters we presented and evaluated most of the elements of the adaptive deployment framework developed in the course of this research. The last missing part in the description of the framework is the adaptation logic that coordinates the way how sensors and effectors cooperate together. The main task of this key element is to collect information from sensors, analyze it and produce a deployment plan that improves overall application performance. In this chapter we describe an approach for adaptive deployment planning that closes the control loop of the proposed ADF framework.

We based our solution on the fact that in model-based deployment both an application and execution environment are represented as graphs. Vertices of these graphs symbolize components and execution nodes, whereas edges represent component links and network interconnections respectively. Moreover, by adding an edge between each component vertex and its hosting node vertex, we can represent a deployed application as a graph which is a combination of the two mentioned graphs. This draws an idea to deal with the deployment planning problem using graph layout algorithms.

From a diversity of graph layout algorithms we focus in this work on Force-Directed Algorithms (FDAs) as some of the most flexible methods to calculate layouts of simple undirected graphs [121]. Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries and, for planar graphs, are often cross-free. Typical applications for FDAs are graph layout and derived problems such as layout in VLSI [20, 39], sensor networks [43] and visualization in data mining [32, 46]. However, due to their simplicity and flexibility they can be easily adapted and extended to fulfill other layout criteria. Further in this chapter we show how FDA can be used to drive adaptive deployment planning.

7.1 Overview of FDA algorithms

Force-directed algorithms are the methods of choice for drawing general graphs. Interest in these methods stems from their aesthetically pleasing drawings, applicability to general graphs and conceptual simplicity. They view a graph as a system of bodies with forces acting between them. Adjacent nodes connected with an edge have an attractive force and nodes without an edge between them repel each other.

In general, FDAs use an energy (or cost) function E that assigns to each embedding $\rho : G \in R^n$ of a graph G in some Euclidean space R^n a non-negative number $E(\rho)$. The algorithms are based on the premise that minima of reasonably chosen energy functions produce desirable graph layouts. The main differences between FDAs are in the choice of energy function and the methods for its minimization [54]. One of the common approaches is to simulate a physical N-body system in which there are repulsive forces (e.g. an electrostatic force between charged particles) between all nodes and attractive forces (e.g. a spring force) between nodes which are adjacent [52, 109]. Then, the goal of the algorithm is to minimize kinetic energy of this system (Listing 7.1).

Listing 7.1: Pseudocode of a basic Force-Directed Algorithm

```
Force_Directed_Algorithm (G : graph)
begin
  set up vertices in random locations
  repeat
    energy := 0
    for each v in V(G)
    begin
      vertex.force := 0

      for each u in V(G) - {v}
        v.force += repulsive_force( v, u )

      for each e in E(v, G)
        v.force += attractive_force( v, e )

      v.velocity += v.force / v.mass  // time interval == 1
      v.velocity *= damping
      v.location += v.velocity        // time interval == 1
      energy += v.mass * v.velocity * v.velocity
    end
  until energy < small_constant
end
```

A characteristic feature of force-directed algorithms is their iterative nature. For simple, static graphs the algorithm loop is finished when system energy

drops below a given small constant. For *dynamic graphs*[1] this loop can be infinite enabling adaptation to the changes in the modeled system. This particular feature is very useful when modeling dynamic systems with constantly changing attributes such as applications running in open distributed execution environments.

The main disadvantages of the presented basic FDA is computational complexity and finding poor local minima. Complexity of the algorithm stems from the fact that each iteration of the algorithm computes $O(|E|)$ attractive forces and $O(|V^2|)$ repulsive forces. There exist, however, several solutions that allow computing graph layout with $O(n\log(n))$ time cost [43, 57, 110] enabling real-time graph visualization consisting of tens or even hundreds thousand of vertices. Similarly, the problem of poor local minima becomes an obstacle with the increasing number of vertices. For the purpose of component deployment, however, neither the time complexity nor the local minima problem are deterrent. This is because software applications and execution environments usually do not comprise of more than several hundreds of components and nodes. For large scale deployments both these issues need further research that is out of scope of this thesis.

7.2 Force-Directed Deployment Planning

The Force-Directed Deployment Planning (FDDP) is a proposed method of deployment planning that makes use of a force-directed algorithm to support adaptive application deployment. As said earlier, a deployment plan of a component-based application can be represented as a graph consisting of two subgraphs: one denoting an execution environment, the other an application being deployed. The actual deployment is represented by edges between verticies of these two subgraphs indicating the hosted–hosting relationship between a component and its execution node. FDDP aims to lay out the environment and application subgraphs and then to map them to each other in a way which yields the best application performance. For example, if an application tends to communicate in a hub-and-spoke manner with a single coordination component and many border components, it will perform best when hosting environment has star-like topology, too. Then the coordination component can be located on the central node and all other vertices on border nodes. The proposed FDDP planning uses the force-directed method to accomplish all three tasks: to lay out the application subgraph, to lay out the execution environment subgraph and to map these subgraphs to each other.

[1]By dynamic we mean that different attributes of a graph may vary in time such as the number of vertices and edges or the forces between vertices.

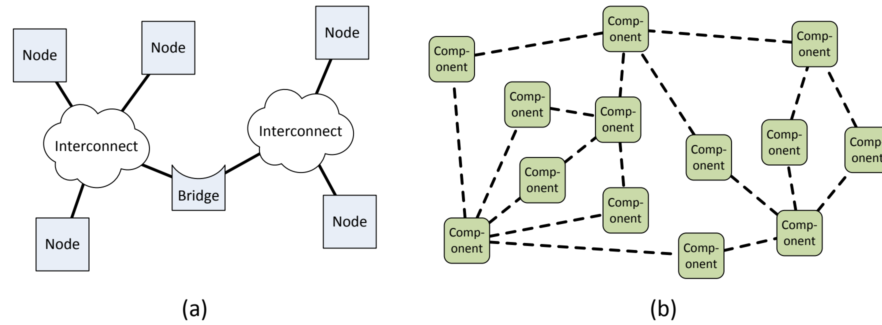

Figure 7.1: Examples of graphs that represent (a) an execution environment and (b) a component-based application in FDDP.

7.2.1 Graph Representation of the Deployment Problem

Before applying a force-directed algorithm to the deployment planning problem we need to model application and execution environment as graphs. This could be done in many different ways but we use models defined in the D&C specification as a basis.

To represent an execution environment we use the *Target Data Model* from D&C. It defines three main elements: `Node`, `Interconnect` and `Bridge`. `Node` represents any element in the environment able to host application components — typically a computer machine.[2] `Interconnect` usually models a computer network that connects closely related host nodes. It is often used to represent local area networks but may also describe overlay networks, point-to-point links or any other connections used as means of communication between components. Lastly, `Bridge` is an abstract network element that joins two or more `Interconnects`. Depending on what an adjacent `Interconnects` are, a `Bridge` may represent a network switch, router, gateway, proxy or any other element located on a border between `Interconnects`. For the purpose of FDDP and applications designed according to the CCM model we assumed the most natural mapping where: `Nodes` represent host computers, `Interconnects` model computer networks and `Bridges` describe network routers. All these elements are vertices in the execution environment graph, while edges always connect `Interconnects` with the other two elements (Fig. 7.1a). Every edge in the graph is assigned a *target length* value that is inversely proportional to the `Interconnects`' throughput. FDDP tries to lay out the graph to minimize the difference between the actual distance of adjacent vertices and the target length of the connecting edge.

The graph representing an application consists of only one kind of vertices. We use `InstanceDeploymentDescriptions` from the *Execution Data Model*

[2]Other examples could be: an OS on a virtualized host or a component server.

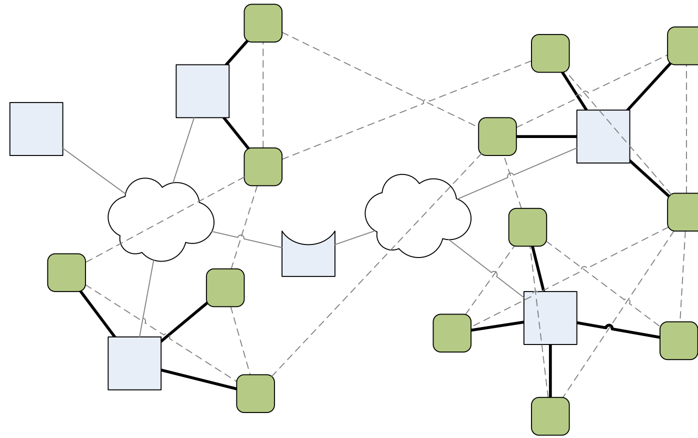

Figure 7.2: An example of a graph representing application deployment in FDDP. Thick black lines denote [component–host node] links; gray solid lines depict interconnect links; gray dashed lines represent [component–component] links.

as graph vertices and connect them with edges only if they communicate (Fig. 7.1b). Instead of using `PlanConnectionDescription` element of a deployment plan we detect communication of components in runtime by exploiting application monitoring. This enables us to assign each edge a value of communication intensity and take it into account when arranging graph vertices.

Lastly, to model the actual deployment of the component-based application in the execution environment we use the `node` attribute of `InstanceDeploymentDescription`. It defines one-to-many mapping between component and node vertices that may be represented as an edge between vertices. Figure 7.2 presents an example of such a mapping.

7.2.2 Forces in FDDP

The basic idea behind any FDA-based graph drawing algorithm is to use forces that will control movement of graph vertices. The algorithms differ between each other in the number and kind of the forces used. When designing the FDDP algorithm we splitted the whole deployment graph lay out problem into three simpler subtasks: (1) to arrange the execution environment subgraph, (2) to arrange the application subgraph, (3) to match these two subgraphs to each other.

In order to lay out the environment graph we used two sets of forces (Fig. 7.3). R_{nn} was a constant repulsive force acting between all node vertices, whereas attractive force A_{ni} was acting between node vertices and

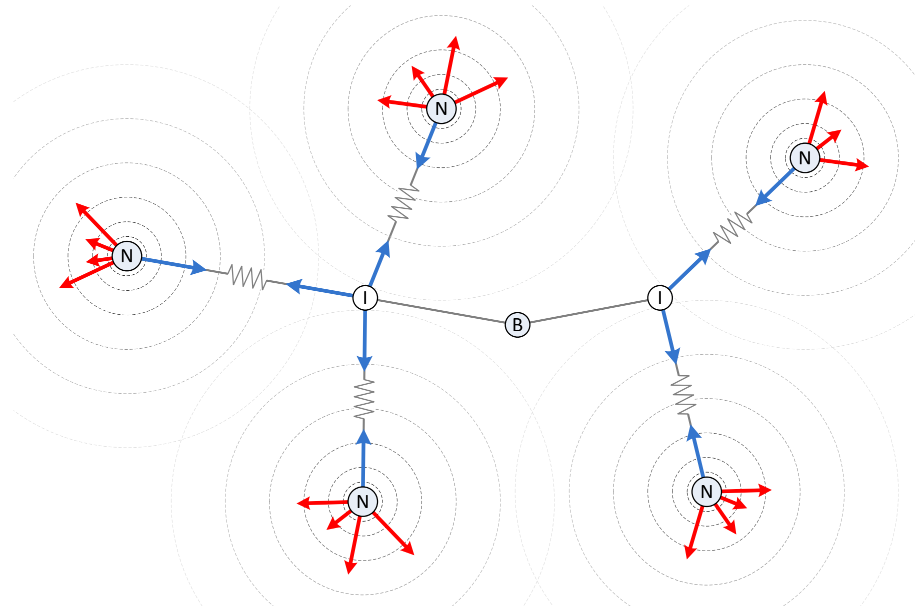

Figure 7.3: An illustration of repulsive R_{nn} (red arrows) and attractive A_{ni} (blue arrows) forces between vertices of a graph representing an execution environment in FDDP.

adjacent `Interconnects`. The properties of the link influenced strength of the A_{ni} force which was proportional to the quality of the link.[3] The higher quality of an interconnect the closer adjacent node vertices were located. Similarly, to arrange an application graph we used two sets of forces. Again, all component nodes repeled each other with a constant force R_{cc}, while an attractive force A_{cc} was acting between directly communicating components only. The strength of this force was proportional to communication intensity. This was desirable because we wanted to minimize the nework distance between the most intensively communicating components. By bringing the components closer to each other the quality of network links was higher and we could avoid unwanted network delays and errors. Lastly, for matching of the application and environment subgraphs we used two kinds of attractive forces. A_{ch} was a constant force acting between a component and its current host node. It represented costs related to component migration. The other, A_{cn}, was acting between a component and all execution nodes and was proportional to the amount of resources available on each node. In result, these two forces made powerful nodes attract components stronger, whereas nodes with fully allocated resources attract only the components they currently hosted.

[3]As link quality we considered network throughput but other metrics could also be included such as latency, reliability, load and error rate.

We did not introduce any repulsive forces between components and nodes subgraphs. After experimentation it appeared that more stable is the solution where the application and environment graphs are separated by a constant distance. Both graphs were laid out on 2-dimensional planes and were separated along the third dimension. It resembles behaviour of charged particles in a capacitor where the particles can move around the capacitor plates but cannot cross a dielectric. The attractive forces influence the direction of their movement until the equilibrium is reached. Concluding, below is the summary of the forces interacting in the FDDP model:

- R_{nn} (a repulsive node–node force) — a constant force between all node vertices as if they are charged particles; responsible for nodes layout,
- A_{ni} (an attractive node–interconnect force) — a force proportional to quality of a link; responsible for grouping together nodes connected with high quality newtork links,
- R_{cc} (a repulsive component–component force) — a constant force between all component vertices as if they are charged particles; responsible for components layout,
- A_{cc} (an attractive component–component force) — a force proportional to communication intensity between components; responsible for grouping together communicating components,
- A_{ch} (an attractive component–host node force) — a constant force between a component and its hosting node; expresses costs of moving a component between nodes,
- A_{cn} (an attractive component–node force) — a force proportional to the power of a node that acts on all application components; responsible for distribution of components among execution nodes.

After some experimentation we chose the following functions to model behaviour of this force-directed system. First, to lay out graphs representing an application and execution environment we used:

$$A_{ni}(d) \sim A_{cc}(d) \sim l \cdot d \qquad \text{and} \qquad R_{nn}(d) \sim R_{cc}(d) \sim -\frac{k \cdot c_1 \cdot c_2}{d^2}$$

where d is a distance between two vertices, l is a model parameter associated with the link between the vertices (similar to resilience of a spring), k is a global constant, and c_1, c_2 are model parameters associated with repulsing vertices (they resemble charge of particles). To model the attracting forces acting between these two subgraphs we used:

$$A_{ch}(d) \sim l \cdot d \qquad \text{and} \qquad A_{cn}(d) \sim \frac{G \cdot m_1 \cdot m_2}{d^2}$$

where l and d are defined as above, G is a global constant, and m_1, m_2 are model parameters associated with attracting vertices (they resemble mass of particles).

Given the presented selection of forces, we needed to map them onto some observable values that could be collected by monitoring sensors.

7.2.3 Mapping of Observables on Model Parameters

When designing FDDP we divided the deployment planning problem onto two separate tasks: matching static and matching dynamic attributes of component requirements against resources. FDDP focuses merely on dynamic attributes of the system, while matching static properties is done by the plain deployment infrastructure. This is because static attributes need to be precisely validated, whereas approximate and iterative nature of force-directed methods can better follow dynamic observables. Based on this assumption we proposed mapping between model parameters and system observables for all six forces presented earlier.

First, to layout the execution environment graph we used two forces: A_{ni} and R_{nn}. Our intention was to lay out its vertices in such a way that directly connected nodes with high quality links were close to each other, whereas nodes indirectly connected or these with low quality network links were located farther away. To achieve this we calculated the l parameter of the A_{ni} force as:

$$l = \frac{A}{\text{link throughput}}$$

where A was a constant defined during experimentation and 'link throughput' was a selected quality metric of a network interconnect. The c_1 and c_2 parameters of repulsive force R_{nn} were constants set experimentally. The higher their values were set the farther away vertices were located. Consequently, distribution of execution node vertices in the model space depended on one dynamic observable — the link throughput — the better interconnect's throughput was the closer vertices were positioned.

Similarly, to lay out an application graph we used two forces: A_{cc} and R_{cc}. Again, we wanted to place application components in such a way that frequently communicating components were closer to each other, while these which did not communicate directly could be farther away. To achieve this we calculated l parameter of the A_{cc} force as:

$$l = \frac{B}{\text{communication intensity}}$$

where B was a constant and 'communication intensity' was a metric that described how intensively two components communicate between each other. We defined this parameter as a ratio between the current and the maximum number of operation invocations per second.The parameters of the R_{cc} force were constants set experimentally, hence distribution of component node vertices in the FDDP model space was influenced merely by communication intensity dynamic observable. In result, the more intense communication was the closer components were located what well corresponded with the layout of the execution environment graph. By tunning strength of the R_{nn} and R_{cc} forces we could acheive appropriate initial matching between application component and execution node graphs.

The last step was to find a mapping for the A_{ch} and A_{cn} forces acting between these two subgraphs. We followed the idea that proper matching would result in an improved application performance, therefore, we aimed to favorize nodes with more resources available. The A_{cn} force was responsible for attracting components to potential host nodes. We set the m_1 parameter of this force as:

$$m_1 = C \cdot \text{resource availability}$$

where C was a constant and resource availability took into account free CPU and memory resources. Value of the m_2 parameter was a constant set experimentally. Consequently, more powerful nodes were the source of a highly attractive force, whereas fully saturated execution nodes exhibited only minimum attractive force. Finally, to represent costs related to component migration we introduced the A_{ch} force and set its l parameter as a constant.

7.2.4 Experimenting with the FDDP Model

The presented mapping between system observables and model parameters is by no means exhaustive and many other solutions could be proposed. Therefore, we perceive FDDP not as a black-box algorithm but rather as an adaptation tool that should be tuned for specific application and execution environment. FDDP offers users many configurable model parameters that can be adjusted to achieve desired behaviour. We experimented with several applications trying to determine model parameters that result in some useful high-level adaptation policies. Although we could not find one universal parameters setting that would fit any kind of applications, we observed

interesting properties of the FDDP model that can be used to control the deployment adaptation process. Thus, to achieve:

- even distribution of components over all execution nodes — decrease the A_{cc} force and increase the A_{cn} force,
- compact distribution of components — increase the A_{cc} force and decrease the A_{cn} force,
- higher system reliability — use settings similar to the compact distribution but take node reliability as an additional metric when calculating the A_{cn} force,
- lower migration overhead — increase the A_{ch} force.

These general guidelines show that despite the approximate nature of the force-directed algorithms it is possible to achieve high-level adaptation strategies. Nevertheless, more extensive tests should be conducted to find finer dependencies and more precise relations.

7.3 Evaluation of the Adaptive Deployment Framework

In order to evaluate our Adaptive Deployment Framework in general and the FDDP algorithm in particular we focus in this section on four aspects. First, we discuss a user interface of the `AdaptationManager` component that enables user interaction with the framework. Second, we present overhead incurred by an application when the ADF framework was active. Third, we show influence of our framework on application performance. Last, we describe behaviour of the application being adapted in case of an external disturbance. All this allows for qualitative analysis of the ADF framework.

7.3.1 Using the ADF Framework

Earlier in this work (see Sect. 4.5 on page 102), we presented general scenario that can be followed to adapt deployment of an application. From a deployer point of view the main interface to the framework is provided by GUI of the `AdaptationManager` component (Fig. 7.4). It supports three main activities in runtime:

- observation of the current application deployment in the environment as well as behaviour of the FDDP algorithm,

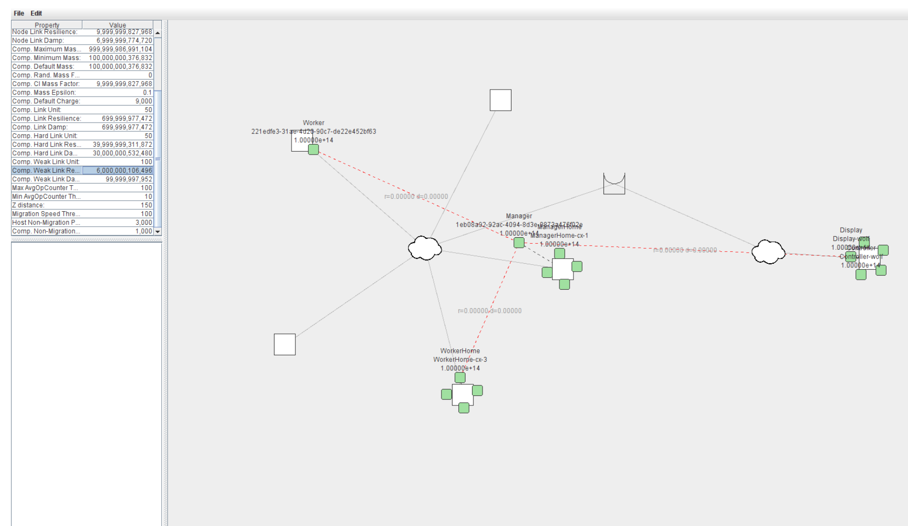

Figure 7.4: The GUI of the prototype adaptation manager component. Green boxes illustrate application components, white boxes represent host nodes, clouds depict interconnects.

- interaction with the migration effector by the drag-and-drop mechanism that allows users to move components between host nodes,
- setting FDDP model parameters to influence operation of FDDP.

As discussed earlier, the force-directed methods produce aesthetically pleasing layouts, therefore, we use this feature to visualize the execution environment, software application and its current deployment. Both the application and environment graphs are presented on a single 2-dimensional plane and are bound between each other with links from each component node to its current host node. Links between components and nodes are either solid lines indicating non-mobile components or dashed lines representing mobile components. The manager discovers which components are mobile by checking if they support interface `CCMRefugee`. Additionally, GUI links components between each other to present the current communication intensity. This allows observing application communication patterns in runtime.

For mobile components, the GUI of the manager enables users to move the components between hosts with a mouse. When a component vertex approaches a host vertex to an arbitrary distance (either dragged by a user or moved by FDDP), the `AdaptationManager` signals the deployment infrastructure to perform runtime reconfiguration. In the first phase it is verified if a particular component can be moved to the indicated host node. Using plan validation, component requirements are matched against node resources. If this matching is successful i.e. the node is able to host the component, further

redeployment steps are performed and finally migration of the component occurs. In the case when matching is unsuccessful, the component is added to the host's *black list*, which means that further migration attempts of the component to this particular host node will be abandoned and the attractive force between these two nodes is zeroed.

The graphical interface of the manager component also enables observation and influence on the operation of FDDP. A user can set a number of model parameters such as k and c factors of the R_{nn} force, global damping and many others and observe how they influence deployment adaptation. This allowed us to determine the set of the FDDP adaptation strategies discussed in the previous section.

7.3.2 Costs of Runtime Adaptation

In the previous chapter we presented costs related to running lower level infrastructure that enables runtime adaptation. In this section we focus on overall costs incurred by an application when adaptation is performed. This includes costs related to running the monitoring infrastructure as well as the adaptation manager. To estimate the overheads we measured execution time of the test application deployed on four Sun Blade servers and one controller node — host M_0. Running the test with the adaptation infrastructure in operation we did not expect any gains from adaptation because the application comprised of only one `Worker` component and all four hosts M_3–M_6 had the same hardware and software configuration. Instead, we could observe negative influence of the ADF on the application execution time.

Figure 7.5 depicts deployment of the testing application (the components in black) and adaptation infrastructure (the components in red). The dashed line marks components that were dynamically deployed as a result of `Worker` migration. They comprise of `WorkerHome`, which is required to support the mobile `Worker` component, and three sensor components needed for application monitoring. These sensors were initially deployed only on machines that run the application (host M_0, M_3 and M_4). Later, when `Worker` was moving between all Blade hosts, the `AdaptationManager` deployed application sensors on all remaining machines. Conversely, `CPUSensor` was deployed on all hosts in the execution environment to supply the `AdaptationManager` with the current state of CPU utilization of all nodes.

Results for different configuration and deployment settings are collected in Table 7.1. As expected, the best performance was achieved when no adaptation infrastructure was running. In this case the system comprised of components drawn in solid black lines in the figure and `Worker` executing in host M_3. For 10×10 and 32×32 chunk size, running the ADF infrastructure

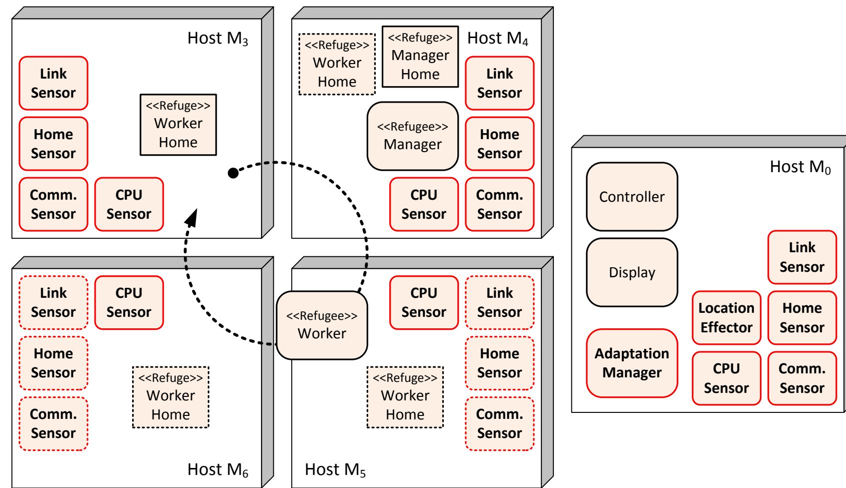

Figure 7.5: Deployment of the ART application together with ADF to verify costs related to its operation.

Table 7.1: Execution time (in seconds) of the ART application with the ADF infrastructure disabled and enabled showing the overheads introduced by the framework.

		With ADF enabled		
Chunk size	No ADF at home	At home no moves	At non-home 1 move	10 moves
10×10	117.0(±1.3%)	151.9(±1.2%)	184.2(±0.1%)	196.5(±1.9%)
32×32	74.0(±2.0%)	92.9(±1.5%)	107.9(±0.9%)	117.3(±0.6%)

caused 30% and 26% increase of execution time, respectively (column 3). However, when `Worker` was moved to node M_4 which was its non-home location (column 4), the observed increase of execution time was 57% and 46%, respectively.[4] Finally, we measured application performance when the `Worker` component was dragged by a user 10 times between hosts M_3–M_6. The observed overhead of 68% and 59% in performance includes the costs related to deployment of all infrastructure and application components needed (depicted in dashed line in the figure) as well as costs of migration and redirection to non-home locations.

It is worth noting that the presented test was designed to show the worst case adaptation scenario. Moving the `Worker` component was related to interruption of the whole application because it was the only computing element in the application. As discussed in the previous chapter, migration

[4]As discussed earlier in Sect. 6.4.2 on page 142, this was the result of redirection which was not cached by the selected ORB implementation.

Table 7.2: Selected static and initial deployments together with the measured execution time of the ART application.

Deployment name	Manager A 4 Workers	Manager B 5 Workers	Manager C 4 Workers	Execution time [s]
Static 0	M_0	M_2	M_3	465.7(±1.0%)
Static 1	M_0	M_2	M_4	459.3(±1.1%)
Static 2	M_2	M_0	M_4	250.1(±0.5%)
Static 3	M_3	M_0	M_2	981.6(±1.2%)
Static 4	M_0	M_3	M_2	1083.0(±1.0%)
Adaptive 0	M_0	M_2	M_3	254.8(±12.5%)
Adaptive 3	M_3	M_0	M_2	289.9(±16.2%)

of the `Worker` was also connected with state loss. Moreover, when moving the `Worker` component to a previously not used node (nodes M_5 and M_6), the `AdaptationManager` had to deploy most of the adaptation infrastructure. Usually, however, when application comprises more mobile components, some of which are of the same type, the incurred overheads will be significantly lower.

7.3.3 Application Performance

In this section we compare application performance for selected manual deployments with the results achieved when the adaptation infrastructure was enabled. We evaluate overall effectiveness of application adaptation and in particular efficacy of the FDDP algorithm. For this test we used six machines and the Asymmetric Ray Tracing application. The application included three `Manager` components that realized mixed, communication- and processing-intensive ray tracing scenarios in parallel. Figure 7.6 depicts a selected application deployment, whereas all tested deployments are summarized in Tab. 7.2. The table presents location of the `Manager` and `Worker` component but does not include the `Controller` and `Display` because for all tests they were placed on host M_0. The deployments 'Adaptive 0' and 'Adaptive 3' denote the initial application deployments 'Static 0' and 'Static 3' when adaptation infrastructure was enabled. These were changed in runtime as adaptation was performed.

The deployment table includes only a small subset of all possible static deployments. However, we tried to select potentially the best settings taking into account performance of the host machines and complexity of the ray tracing tasks. Therefore, vast majority of the static deployments would exhibit worse results than these presented here.

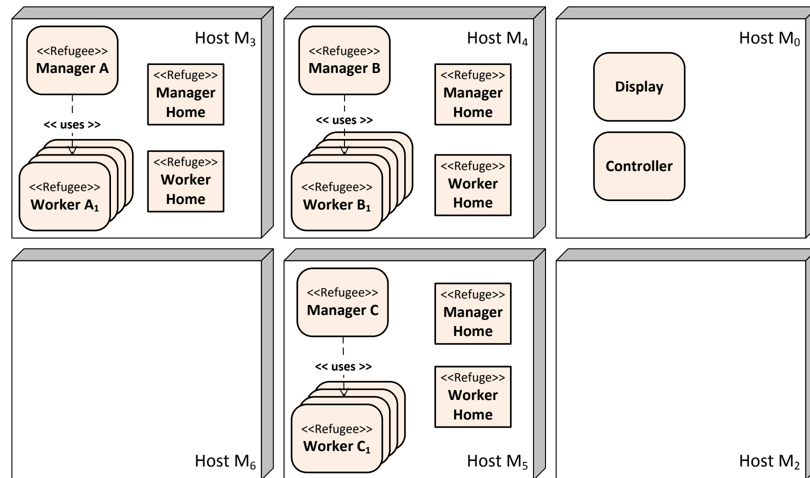

Figure 7.6: A selected deployment of the ART application used for testing effectiveness of our ADF framework.

Surprisingly, the results show that despite of relatively high costs related to running ADF, an application can greatly benefit from deployment adaptation. When the adaptation infrastructure was running, execution time was comparable to the best static deployment we found (Fig. 7.7). We can see three main reasons why these results were so satisfactory. Firstly, in the case of static deployment overall time of application execution was equal to the slowest assignment: [machine ⇐ task to render] e.g. deployment 'Static 4' is the slowest presented because slow machine M_3 was assigned the hardest task B to process. When adaptation was enabled free machines were able to host some workers and do parts of the job on behalf of the others. However, this was only possible due to inherently parallel nature of the ray tracing problem, which can be easily split into smaller parts. Thus, the fastest M_0 host was quickly available after it completed the preassigned task and then it attracted some foreign `Workers`. Moreover, when adaptation was enabled not only the machines initially planned to do the rendering were involved in processing but also all other free hosts in the domain. In result the application was automatically distributed over the whole execution domain.

Additionally, the presented results confirm what was expected — the initial application deployment is important for achieving the best results. Comparing deployment 'Adaptive 0' with 'Adaptive 3' we observed nearly 14% increase in execution time. It was the cost of reconfiguration from ill-suited initial component distribution to a more effective detected by ADF.

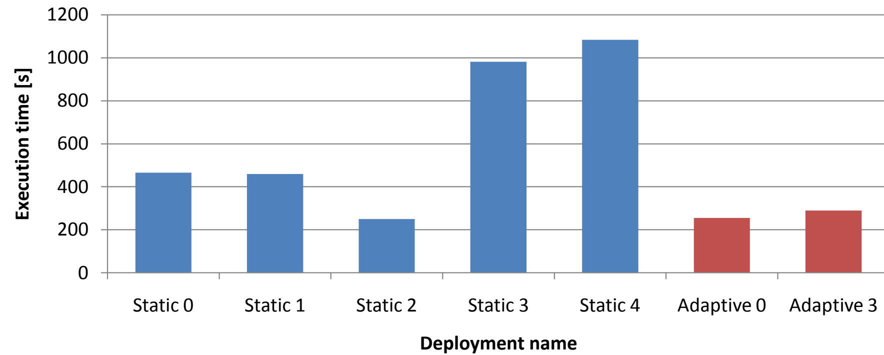

Figure 7.7: Execution time for different deployments of Asymmetric Ray Tracing.

7.3.4 Adaptation to an External Disturbance

The previous section showed that, for some class of applications, adaptation can yield very good results which can hardly be achieved with only manual deployment. Our adaptive framework, however, not only allows increasing performance but also can react to an external distrubance and reconfigure application in a way that avoids allocating overloaded nodes. In this testing scenario we were running two instances of the Asymmetric Ray Tracing application: one, deployed statically, played the role of disturbance, while *the control instance* was adapted by our ADF framework. Figure 7.8 shows deployment of the disturbing instance and the processing load spread over the execution nodes. Nodes M_0, M_2 and M_3 were hosting the most CPU-intensive `Worker` components. Nodes M_4 and M_5 were running low demanding `Managers` which merely generate input for workers and relay the results to the `Display` component. Lastly, node M_6 hosted the `Controller` and `Display` components that required more processing power because `Display` collected and presented image chunks from all the managers.

During the test we noticed desirable behaviour of the ADF framework. When the disturbing instance was active we deployed the control instance and observed its reconfiguration. Figures 7.9 and 7.10 present screenshots of the ADF console window. We augmented the figures to ease node identification and to mark the external workload they were assigned. The initial deployment of the control instance placed all mobile components on node M_2, while `Controller` and `Display` were located on node M_0. The figures show migration of `Workers` towards the least loaded nodes M_7, M_4 and M_5. Most components moved to the node M_7 which was free from any external disturbance, whereas nodes M_4 and M_5 hosted less number of components.

In order to evaluate how this adaptation process influence application performance we compared execution time of the control instance when it was managed by ADF with times collected for selected static deployments of

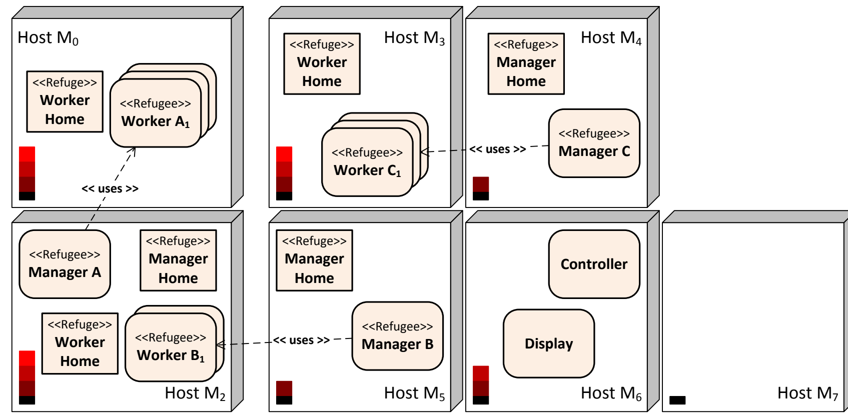

Figure 7.8: Deployment of the application that played role of an external disturbance. Color bars show the processing load carried by a host machine.

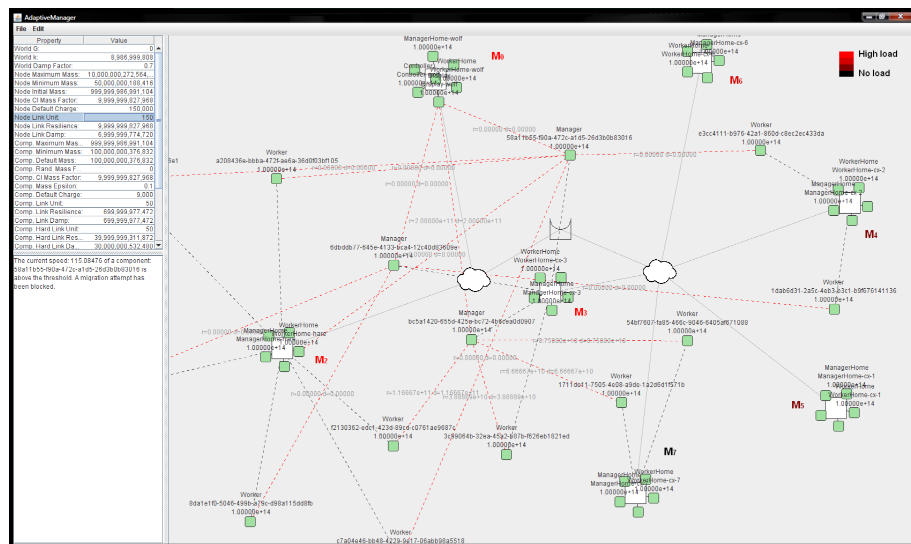

Figure 7.9: Distribution of application components in the first phase of application adaptation; red labels were added to this screenshot to allow correct identification of the execution nodes and to show their load from an external disturbance.

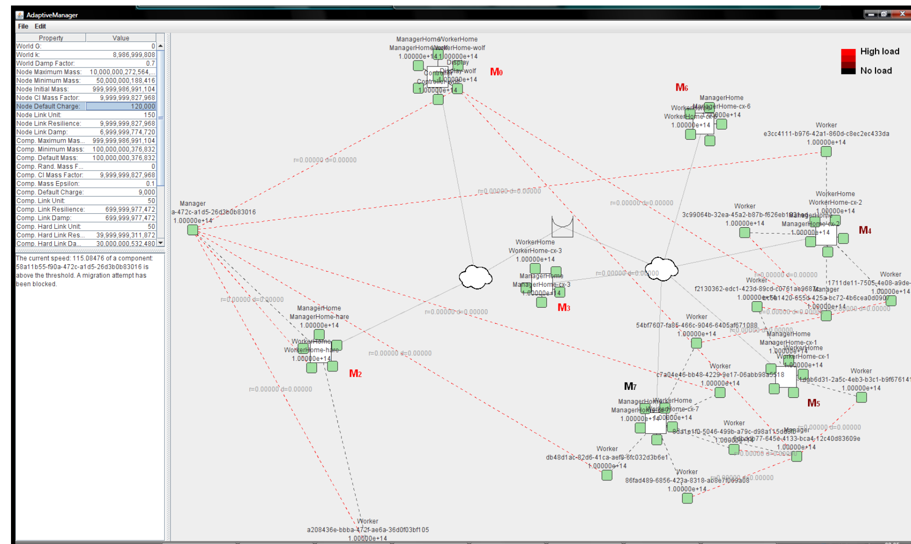

Figure 7.10: Distribution of application components that avoids an external disturbance; red labels were added to this screenshot to allow correct identification of the execution nodes and to show their load from the external disturbance.

the control application. In all these tests the distrubance instance was active. Table 7.3 presents the results in relation to static and adaptive deployments. In all cases the `Controller` and `Display` component were deployed on the machine M_0. As shown in the table, despite the high costs of adaptation ADF proved to be able to increase overall application performance in the presence of an external disturbance. This time, performance of the application managed by ADF was much better than the best static deployment — we noticed over 20% of decrease in execution time comparing to deployment 'Static 1'. This scenario shows that in particular settings adaptation of deployment can be an effective tool for improving application performance.

To better understand why the results were so positive it is important to notice that disturbance was deployed on the fastest nodes in the environment i.e. M_0 and M_2. Therefore, any static deployment that involved these nodes made the control instance performing significantly worse. Furthermore, in the middle of application execution the disturbing instance released the second the fastest machine M_2 what again had major impact on performance. All statically deployed components were bound to their hosts while the mobile `Workers` were attracted by a free powerful machine. The change in the disturbance was required to distinguish this testing scenario from the case presented in the previous section. Overall, the results show that ADF is able to adapt component-based system to an external disturbance and that adaptation can improve application performance.

Table 7.3: Execution time (in seconds) for selected static deployments of the ART application in comparison to the application managed by ADF.

Deployment	`Managers` locations	`Workers` locations	Execution time
Static 0	{A, B, C} $\Rightarrow M_2$	{A, B, C} $\Rightarrow M_0$	519.8(±0.7%)
Static 1	A $\Rightarrow M_5$, B $\Rightarrow M_4$ C $\Rightarrow M_3$	{A, B} $\Rightarrow M_2$ C $\Rightarrow M_6$	448.8(±0.9%)
Static 2	{A, C} $\Rightarrow M_5$ B $\Rightarrow M_4$	{A, B, C} $\Rightarrow M_2$	590.7(±0.6%)
Static 3	A $\Rightarrow M_2$, B $\Rightarrow M_4$ C $\Rightarrow M_5$	{A, B} $\Rightarrow M_2$ C $\Rightarrow M_6$	454.0(±0.6%)
Adaptive 4	{A, B, C} $\Rightarrow M_2$	{A, B, C} $\Rightarrow M_2$	353.1(±9.1%)

7.4 Limitations of the FDDP Planner

When adapting deployment of applications the FDDP algorithm tries to solve three different subproblems: (1) arranging execution environment graph, (2) arranging application graph and (2) matching these two graphs between each other. The first two problems are relatively easy tasks for force-directed methods. We achieved a very pleasant looking network and application graph layouts that were sensible for changes in communication patterns and resource availability. FDDP could easily follow these changes in runtime. The main difficulties we found when experimenting with the algorithm were related to the latter — the matching problem. The key issues that made the force-directed approach problematic were:

- difficult mapping between model parameters and real system observables — we proposed an example mapping but it was not easy to find an appropriate forces for a particular observable. For example is was not obivous how quality of a network link or intensity of communication should influence host and component vertices. The former we modeled as link length, whereas the latter as link resilience. However, many other mappings could be proposed and tested,
- unclear relation between model parameters of the environment graph and application graph — for example it was not obvious how to set length of edges representing network connections in relation to the length of edges representing links between components. Setting network edges too long made components be bound to their host nodes, while setting them too short caused too many migration requests,

- problems with scalability and limited use of model parameters — we did not find any other way for tuning model parameters than by experimentation. Improperly set parameters result in instability of FDDP for certain application conditions. Once found, the correct parameter values cannot be directly applied for systems of different size. The positive fact is that as long as parameters are valid for a certain number of components and execution nodes they can be applied to different applications and do not depend on application architecture,
- unclear user-level control — although we presented some rules how to achieve a number of high-level adaptation policies it is not easy to find such rules for any kind of user-level control policy. Therefore, to apply other adaptation policies more experimentation would be required.

7.5 Summary and Conclusions

In this chapter we presented and evaluated the key elements of our adaptive deployment framework: the `AdaptationManager` component and the FDDP planning algorithm which is a novel approach to software deployment adaptation.

We briefly presented the graphical user interface of the manager and discussed how it enables interaction with the running application and FDDP. Later we measured costs of runtime adaptation and showed that our prototype implementation incurs up to 60% overhead on application execution time. Despite of high costs, however, we achieved very promising results in respect to application performance and deployment adaptability. Execution time for an application managed by our framework was comparable to the best static deployment we found. Moreover, in emergence of an external disturbance the framework was able to avoid overloaded execution nodes and move application components towards less utilized hosts. This resulted in over 20% decrease of execution time.

We found that key points to achieve such good results were: component-based design (esp. according to the CCM model), effective reconfiguration mechanism (i.e. runtime component migration) and suitable deployment planning algorithm. Firstly, the component-based architecture forces programmers to split a system into well defined parts what enables better parallelization and reconfiguration making adaptation easier to conduct. Secondly, the component migration mechanism combined with a plain deployment infrastructure proved to be effective means to achieve deployment adaptation. Relatively low overhead of migration makes it suitable for use in runtime deployment reconfiguration. Lastly, the idea of modelling a distributed system as a set of interacting physical bodies very well fits its

dynamism. Although the proposed force-directed deployment planning is not the main contribution of this work, it well illustrates the potential of the constructed platform in performing deployment adaptation. Flexibility of the force-directed approach allows adapting it to different user criteria. We proposed some general rules for mapping user-level adaptation strategies onto model parameters. Unfortunately, the flexibility of FDAs is also their major disadvantage. We could not find a way different than experimentation to determine a proper mapping of observables onto model forces and setting correct values for their parameters. The abundance of possible mappings and parameter settings makes the problem of using FDDP for deployment planning the area of further research. Nonetheless, we find this approach to be promising.

Chapter 8

Conclusions and Possible Research Directions

In this work we have presented an adaptive deployment framework for distributed software systems running in heterogeneous environments. The framework is dedicated for component-based applications built on top of a middleware layer. This locates our solution between adaptive platforms based on system virtualization and solutions exploiting object-oriented reconfiguration techniques. We find this approach right. On the one hand component-based software design makes adaptation easier to conduct while on the other hand it can be more effective than fine-grained object-based solutions. Moreover, the component-based application modelling promotes separation between component's business code and its execution environment what facilitates achieving reconfiguration transparency. It also very well corresponds to the model-based deployment methods that enable automation of deployment planning, thereby clearing the way for deployment adaptation.

The basis for an adaptive deployment framework is a plain deployment infrastructure. Building a complete deployment infrastructure for component-based distributed applications and heterogeneous environments is earnest endeavour though. Therefore, in this work we focused only on these aspects that are crucial for deployment adaptation. One of them is the problem of deployment planning. We have shown that this complex step may benefit from adaptation by splitting it onto two simpler tasks: initial and runtime planning. The initial deployment planning can focus merely on static attributes of the software and execution environment what greatly simplifies description of their requirements and resources. The generated initial plan can then be improved in runtime when more accurate data on the operation of the whole system are available.

To conduct deployment adaptation we used spatial reconfiguration of software using the runtime component migration mechanism. The design and implementation of this mechanism is one of the major contributions of this work. It proved to be fast and lightweight enough as a deployment reconfiguration tool. However, providing runtime migration in the CCM environment that supports asynchronous and multithreaded operation was a difficult task. This environment poses many fundamental issues that need to be addressed by a migration mechanism such as portable state preservation, residual dependencies and reaching quiescent state. As they can hardly be hidden from developers without enforcing significant constraints on component implementation, we proposed a solution that involves developers in component migration. By means of extensions to component life cycle, we allow for a lot of freedom in the way how components can be implemented and also we make developers aware of the issues inherent to the migration process. This, we believe, can help building components that make most of the runtime mobility.

In heterogeneous environments a component migration mechanism requires a deployment infrastructure to operate properly. The deployment infrastructure can validate reconfiguration requests and prepare the execution environment for changes. Therefore, runtime component migration and software deployment are complementary mechanisms which combined together create a synergy that can improve management and performance of distributed applications. We showed that despite its costs, support from our ADF framework can very positively influence on application performance. The conducted experiments have demonstrated that for a certain class of systems, which we simulated by the Asymmetric Ray Tracing application, adaptive deployment yields very good results in terms of execution time. We achieved results comparable to the best static deployment found and in the case of external disturbance our ADF reduced execution time by 20%. Such satisfactory results were achieved even though our prototype implementation imposes relatively high overheads.

Our adaptive deployment framework was built using the component-based approach and the Autonomic Computing paradigm. This facilitates its further extensions in respect of new planning algorithms as well as new sensor and effector components. There are, however, many other potential extensions that involve more radical changes. A very interesting one is support for virtualization. A combination of the model-based approach for distributed systems proposed by the D&C specification with vertical software deployment defined in the Solution Deployment Descriptor specification could result in a very robust deployment infrastructure. Such an infrastructure would make deployment of application easier because together with the application itself it could deploy its execution environment (e.g. a JEE application, JEE application server and JRE environment). This, however, would be only a step

on the way to adaptive distributed deployment in virtualized heterogeneous environments.

Another interesting area of further research embrace the force-directed methods in deployment. In this work we discussed design and implementation of FDDP — a novel approach to runtime component deployment. Our experiments showed that FDDP can effectively perform deployment adaptation, however, there is still a number of issues that require closer investigation. The main one is scalability of the approach. We tested FDDP in a LAN connecting up to 11 hosts and simple applications consisting of 3–25 components. Even in such limited settings we observed the need for manual tuning of model parameters. Otherwise, the algorithm was unstable. It would be especially interesting to search for a self-tuning solution for FDDP that could adapt the algorithm to the particular conditions of the execution environment and size of the application.

Further investigation could also be pursued in the area of deployment planning in temporal and semantic dimensions. Ability to plan in all three dimensions defined in this work can again increase usability of a deployment infrastructure. An important point is that some of the mechanisms developed in the course of this research could be used for the enacting plans in other dimensions. For example, component instance monitoring can support execution of temporal plans, whereas runtime plan updates can ease switching between service instances in semantic deployment.

Apart from these high-level extension objectives we consider two technical issues that are crucial to implement before the production use of our prototype framework is possible. First is the security which is undermined if deployment is too straightforward. Automation of deployment makes dissemination of software, including malicious code, much easier. Therefore, the key aspect is to protect host machines from hackers' attacks. Second issue is transaction-awareness. In our research we made an optimistic assumption that validation of deployment descriptors is enough to properly deploy a component. However, numerous problems can appear during component deployment even if it fits the target machine. Implementing deployment process supported by transactions is critical for its correct operation in case of errors.

In this work we have shown how adaptive deployment can improve application performance. Nevertheless, adaptation of deployment process facilitates many other interesting applications such as execution of code as close to data as possible, increasing system reliability and improving overall user experience. We hope that the presented work will inspire readers to conduct other research in this fascinating area.

Appendix A

IDL interfaces

Listing A.1: The excerpt from the `Deployment.idl` file showing the proposed extensions to the D&C models.

```
enum PatchOperation  { AddOp, DelOp, SetOp };
enum UpdateKind { InternalUpdate, ExternalUpdate };

interface RunningApplication
{
  /**
   * This operation not only updates the plan of the
   * application but also does all the required steps to
   * execute the plan and to ensure consistency between
   * the deployment plan and the managed application
   * deployment. Similarly to the launch operations this
   * operation also has two phases, one responsible for
   * instance changes (if required). The second for doing
   * proper interconnection.
   */
  CORBA::OctetSeq startUpdate(
      in  UpdateKind  kind,
      in  CORBA::StringSeq  elements,
      in  DeploymentPlan  patch,
      in  PatchOperation operation,
      out Connections providedReference)
  raises (
    ResourceNotAvailable, PlanError,
    InvalidProperty, InvalidNodeExecParameter,
    InvalidComponentExecParameter, UpdateError);

  /**
   * This is the second and the last step of the update
   * operation which mimics the launch pattern proposed in
   * D&C. This operation allows interconnection between
```

```
   * the updated instances and the rest of the system.
   */
  DeploymentPlan finishUpdate(
      in CORBA::OctetSeq  cookie,
      in Connections  providedReference)
  raises (UpdateError, InvalidConnection);

  /**
   * This is used to cancel update request.
   */
  void cancelUpdate(
      in CORBA::OctetSeq  cookie);

  /**
   * Deactivates operation of the running application.
   * All resources should be freed and the application
   * should not be accessible any more.
   */
  void stop()
  raises (StopError);

  /**
   * Provides the most recent deployment plan for the
   * application instance taking into account all
   * updates. Updating plan can cause differences between
   * this deployment plan and the plan returned by
   * NodeApplicationManager.
   */
  DeploymentPlan getDeploymentPlan();

  /**
   * Allows looking for an object instance using instance
   * name from the application deployment plan.
   */
  Object findInstanceByName(
      in string instanceName)
  raises(NoSuchName);
};

// The reason to distinguish between node and domain
// running application interfaces is to have knowledge
// what kind of object one manipulates with and to avoid
// mistakes of calling operation on the wrong
// implementation. It is expected that these
// implementations will differ significantly.
//

interface NodeRunningApplication : RunningApplication
{ };
```

```
interface DomainRunningApplication : RunningApplication
{ };
```

Listing A.2: IDL definition of Sensor and Effector interfaces

```
interface Sensor
{
  PropertyTypes getSensorPropertyNames();

  PropertyTypes getSensorPropertyArgs(
      in string propertyName )
  raises (UnknownProperty);

  any getProperty(
      in string propertyName)
  raises (UnknownProperty);

  AnySeq getProperties(
      in StringSeq propertyNames)
  raises (UnknownProperty);

  any getXProperty(
      in string propertyName,
      in Properties args)
  raises (UnknownProperty, InvalidArgument);
};

interface Effector
{
  PropertyTypes getEffectorPropertyNames();

  PropertyTypes getEffectorPropertyArgs(
      in string propertyName)
  raises (UnknownProperty);

  PropertyTypes getEffectorOperationNames();

  PropertyTypes getEffectorOperationArgs(
      in string name)
  raises (UnknownOperation);

  void setProperty(
      in string name,
      in any value)
  raises (UnknownProperty);

  void setProperties(
      in Properties props)
  raises (UnknownProperty);
```

```
  void setXProperty(
      in Property prop,
      in Properties args)
  raises (UnknownProperty, InvalidArgument);

  any invokeOperation(
      in string name,
      in Properties args)
  raises (UnknownOperation, InvalidArgument);
};
```

Listing A.3: IDL definition of the `PropertyUpdate` event

```
eventtype PropertyUpdate
{
  public string      senderName;
  public Properties  properties;
};
```

Appendix B

Description of an Execution Environment

This Appendix presents an excerpt from a description of our testing execution environment used in evaluation. The topology of the environment is presented in Figure 6.1 on page 130.

Listing B.1: An example of a Domain description.

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment:Domain
  xmlns="http://www.omg.org/Deployment"
  xmlns:Deployment="http://www.omg.org/Deployment">

  <UUID>1f4303e0-2eea-47c2-be39-73f5aade410f</UUID>
  <label>A test bed domain</label>

  <infoProperty>
    <name>description</name>
    <value>
      <type><kind>tk_string</kind></type>
      <value>
        <string>
        This is full test bed domain prepared for
        evaluation of ADF building blocks and the
        framework itself.
        </string>
      </value>
    </value>
  </infoProperty>

  <node href="cx.xml" />
  <node href="cx-1.xml" />
  <node href="cx-2.xml" />
```

```
  <node href="cx-3.xml" />
  <node href="cx-4.xml" />
  <node href="cx-5.xml" />
  <node href="cx-6.xml" />
  <node href="cx-7.xml" />
  <node href="wolf.xml" />
  <node href="hare.xml" />
  <node href="iris.xml" />

  <interconnect href="net_10_1_17.xml" />
  <interconnect href="net_149_156_97.xml" />

  <bridge href="rt_dsrg-cx.xml" />
</Deployment:Domain>
```

Listing B.2 presents an excerpt from a node definition. To describe the CPU and OS resources we used the CIM schema and a set of selected classes and their properties.

Listing B.2: XML description of a selected Node.

```
<?xml version="1.0" encoding="UTF-8"?>
<Deployment:Node
  xmlns="http://www.omg.org/Deployment"
  xmlns:Deployment="http://www.omg.org/Deployment">

  <name>cx-3</name>
  <label>CX-3 Blade Machine</label>
  <connection href="net_10_1_17.xml" />

  <resource>
    <name>CPU0</name>
    <resourceType>CIM_Processor</resourceType>
    <property>
      <name>Family</name>
      <kind>Attribute</kind>
      <value>
        <type><kind>tk_short</kind></type>
        <value><short>2</short></value>
      </value>
    </property>
    <property>
      <name>MaxClockSpeed</name>
      <kind>Maximum</kind>
      <value>
        <type><kind>tk_ulong</kind></type>
        <value><ulong>650</ulong></value>
      </value>
    </property>
  </resource>
```

```
  <resource>
    <name>RunningOS</name>
    <resourceType>CIM_OperatingSystem</resourceType>
    <property>
      <name>OSType</name>
      <kind>Attribute</kind>
      <value>
        <type><kind>tk_short</kind></type>
        <value><short>30</short></value>
      </value>
    </property>
    <property>
      <name>TotalVirtualMemorySize</name>
      <kind>Maximum</kind>
      <value>
        <type><kind>tk_ulonglong</kind></type>
        <value><ulonglong>2097000</ulonglong></value>
      </value>
    </property>
    <!-- all other relevant static properties... -->
  </resource>
  <!-- all other relevant resources... -->
</Deployment:Node>
```

For describing resources of interconnects and bridges (Listings B.3 and B.4), CIM does not fit well to the approach to represent network proposed in the D&C specification. The CIM schema is very detailed in this respect (see [36]), whereas D&C proposes a general and less accurate view on a network. For this reason, we decided to describe network elements with a proprietary resources and properties. This implies that an application description needs to be aware of what resources and properties can be requested.

Listing B.3: The XML definition of a selected interconnect form the testing execution domain

```
<Interconnect
  xmlns="http://www.omg.org/Deployment"
  xmlns:Deployment="http://www.omg.org/Deployment">

  <name>10_1_17_0</name>

  <connect href="cx-1.xml" />
  <connect href="cx-2.xml" />
  <connect href="cx-3.xml" />
  <!-- all other nodes ... -->

  <connection href="rt_dsrg-cx.xml" />
```

```
  <resource>
    <name>Blades network throughput</name>
    <resourceType>network throughput</resourceType>
    <property>
      <name>LAN throughput</name>
      <kind>Capacity</kind>
      <value>
        <type><kind>tk_ulonglong</kind></type>
        <value><ulonglong>1000000</ulonglong></value>
      </value>
    </property>
  </resource>
  <!-- all other resources ... -->
</Interconnect>
```

Listing B.4: The XML definition of a bridge between DSRG's LAN and the dedicated LAN connecting Blade servers.

```
<Bridge
  xmlns="http://www.omg.org/Deployment"
  xmlns:Deployment="http://www.omg.org/Deployment">

  <name>Router DSRG-CX</name>
  <label>A router between DSRGs and 10.1.17.0 LANs</label>

  <connect href="net_149_156_97.xml" />
  <connect href="net_10_1_17.xml" />

  <resource>
    <name>bridge throughput</name>
    <resourceType>network throughput</resourceType>
    <property>
      <name>LAN throughput</name>
      <kind>Capacity</kind>
      <value>
        <type><kind>tk_ulonglong</kind></type>
        <value><ulonglong>10000000</ulonglong></value>
      </value>
    </property>
  </resource>
</Bridge>
```

Appendix C

Support for the Planning Dimensions

Spatial Deployment Planning

This dimension is already supported by the D&C specification and no further changes are required to its models.

Semantic Deployment Planning

The semantic planning dimension, although not considered by the D&C specification, can be easily supported with only minor structural changes. The basic elements needed for semantic planning are already present in the specification:

- a component implementation can declare its capabilities,
- a component implementation can declare dependencies on other applications,
- an assembly component implementation can declare references to external applications' ports.

Originally, the capabilities declared by a component implementation are means to enable selection of an implementation from any of the equivalent implementations included in an application package being deployed. There is no reason, however, why they cannot be used also to select an active application from several already running instances. To do this we need

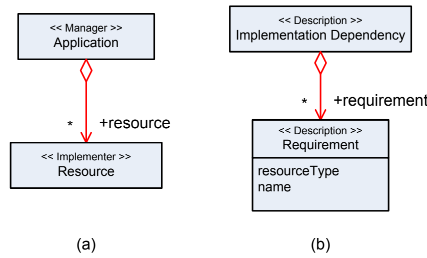

Figure C.1: Extensions to the D&C models enabling semantic deployment planning; (a) an additional association added to the *Execution Management Model*; (b) an additional association between elements from the *Common Elements* model

to add an association between the `Application` entity and the `Resource` structure (Fig. C.1a).[1]

To be able to express dependencies on other running applications, a component implementation being deployed must have measures to provide desired requirements. Therefore, dependencies included in the `ComponentImplementationDescription` structure and described by `ImplementationDependency` may not only determine required application types but also should include additional requirements against running application resources (Fig. C.1b). Later they will be matched in the planning phase.

Finally, when a planner has selected the running application instances that are to be connected with the newly deployed application, it can refer to their ports using the `ExternalReferenceEndpoint` structure.

These minor corrections presented above enable a semantic binding strategy. A packager or developer define dependencies on types and resources of external services (denoted by `Application` structure), then the planner selects the most appropriate running instances and the executor binds these services together with the newly deployed application. The ability to specify service resources is means to realize robust semantic deployment planning. During service execution, resources can be updated by a monitoring facility and convey dynamic information about current QoS characteristics like mean response time, service availability, number of transactions per second, etc.

[1]Capabilities differ from resources only in that they are not consumable and cannot include dynamic properties. The reason why originally a component implementation includes capabilities but not resources stems from the fact that it describes static attributes that cannot change. However, in this case we consider running applications which attributes are dynamic and, therefore, are better described by resources.

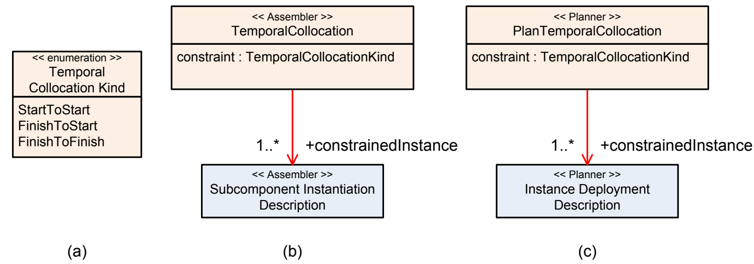

Figure C.2: Extensions to the D&C models enabling temporal deployment planning; (a) The collocation kind added to the *Common Elements* model; (b) The collocation structure added to the *Component Data Model*; (c) The collocation structure added to the *Execution Data Model*.

Temporal Deployment Planning

Comparing to imperative language-based approach, the existing ADLs do not offer as much flexibility as required to support deployment in time. Support for temporal deployment planning is a possible solution for this limitation. A deployment plan in the D&C specification is a collection of component instance descriptions that are assigned to nodes in the target execution environment. In this specification the assumption is that all application components declared in the plan shall be activated in the same time when the deployment process begins.

To support temporal planning our extensions are based on the work from the job scheduling area [86, 107] and are analogous to the structures that define locality constraints in the D&C models. By introducing another type of collocation constraints we enable defining temporal dependencies modelled as a DAG. The extensions include `TemporalCollocation` and `PlanTemporalCollocation` structures and the `TemporalCollocationKind` enumeration and are depicted in Fig. C.2.

We distinguished three kinds of temporal collocations. The `StartToStart` and `FinishToFinish` allow creating synchronization barriers, whereas `FinishToStart` provides means to create deployment task sequences. Using these collocation kinds[2] together with temporal collocation structures a packager and planner can easily represent a broad range of DAG graphs modelling temporal dependencies between deployment of component instances. To ensure backward compatibility with the original D&C model we assume that component instances not bound with any temporal constraint are activated at the beginning, together with all root components of DAG graphs.

[2]All of them are borrowed from the project management discipline where they are used in defining task dependencies.

Bibliography

[1] M. Aida, N. Miyoshi, and K. Ishibashi. A scalable and lightweight qos monitoring technique combining passive and active approaches. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications Societies. IEEE*, volume 1, pages 125–133, April 2003.

[2] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonellotto. Behavioural skeletons in GCM: autonomic management of grid components. In *16th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, 2008.

[3] M. Aldinucci, M. Danelutto, H.L. Bouziane, and C. Pérez. Towards software component assembly language enhanced with workflows and skeletons. Technical Report TR-0153, CoreGRID - Network of Excellence, July 2008.

[4] P. Antoniewski, Ł. Cygan, J. Cała, and K. Zieliński. Extension of a CCM environment with and adaptive planning mechanism. In *CISSE 2006*, 2006.

[5] S.B. Atallah, D. Hagimont, S. Jean, and N. de Palma. A first step towards autonomous clustered J2EE applications management. In *First International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters,* Saint-Malo, June 2004.

[6] D. Ayed and Y. Berbers. Dynamic adaptation of CORBA component-based applications. In *SAC*, pages 580–585, 2007.

[7] P. Backx, B. Van Den Bossche, and B. Dhoedt et. al. Aadd: Autonomic adaptive distributed deployment of component-based services. In *IMSA 2005*, 2005.

[8] J. Bang-Jensen and G. Gutin. *Digraphs Theory, Algorithms and Applications*. Springer-Verlag, August 2007.

[9] S. Bouchenak, N. de Palma, and D. Hagimont. Autonomic administration of clustered J2EE applications. In *IFIP/IEEE International Workshp on Self-Managed Systems & Services*, 2005.

[10] S. Bouchenak, N. de Palma, D. Hagimont, and C. Taton. Autonomic management of clustered applications. In *IEEE International Conference on Cluster Computing*, Barcelona, September 2006.

[11] H.L. Bouziane, C. Pérez, and T. Priol. A software component model with spatial and temporal compositions for grid infrastructures. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 698–708. Springer-Verlag, 2008.

[12] L. Broto, D. Hagimont, P. Stolf, N. de Palma, and S. Temate. Autonomic management policy specification in tune. In *23rd Annual ACM Symposium on Applied Computing*, Fortaleza, Ceará, Brazil, March 2008. ACM.

[13] J. Cała. Migration of components with OpenCCM, March 2006. Internal report; http://hare.ics.agh.edu.pl/migration_report.pdf.

[14] J. Cała. Migration in CORBA component model. In J. Indulska and K. Raymond, editors, *Distributed Applications and Interoperable Systems*, volume 4531 of *LNCS*, Paphos, Cyprus, June 2007. IFIP, Springer-Verlag.

[15] J. Cała, Ł. Kubica, W. Wiśniowski, and K. Zieliński. Adaptation of CCM applications based on lightweight OS virtualization. In *FeBID 2007 — Workshop on Feedback Control Implementation and Design in Computing Systems and Networks*, May 2007.

[16] J. Cała and K. Zieliński. Influence of virtualization on process of grid application deployment — CCM case study. In *Cracow Grid Workshop 2006*, 2006.

[17] C. Canal, J.M. Murillo, and P. Poizat. Software adaptation. *J. UCS*, 14(13):2107–2109, 2008.

[18] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton. Proactive: an integrated platform for programming and running applications on grids and P2P systems. *Computational Methods in Science and Technology*, 12(1):69–77, 2006.

[19] A. Carzaniga, A. Fuggetta, R.S. Hall, A. van der Hoek D. Heimbigner, and A.L. Wolf. A characterization framework for software deployment technologies. Technical Report CU-CS-857-98, University of Colorado, Department of Computer Science, April 1998. Air Force Material

Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253.

[20] T. Chan, J. Cong, and K. Sze. Multilevel generalized force-directed method for circuit placement. In *Proceedings of the International Symposium on Physical Design*, April 2005. Best paper award at ISPD'2005.

[21] O. Chebaro, L. Broto, J.-P. Bahsoun, and D. Hagimont. Self-tuneing of a J2EE clustered application. In *2009 Sixth IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, 2009.

[22] U. Chukmol. A framework for web service discovery: service's reuse, quality, evolution and user's data handling. In *IDAR '08: Proceedings of the 2nd SIGMOD PhD workshop on Innovative database research*, pages 13–18, New York, NY, USA, 2008. ACM.

[23] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. *Approximation Algorithms for NP-Hard Problems*, chapter Approximation Algorithms for Bin Packing: A Survey, pages 46–93. PWS Publishing Co., Boston, MA, USA, 1997.

[24] CoreGRID. Deliverable d.pm.04 — basic features of the grid component model. Technical report, CoreGRID Network of Excellence, March 2007.

[25] R.L. Cottrell. Passive vs active monitoring. http://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html, March 2001.

[26] G. Coulson. What is reflective middleware? *IEEE Distributed Systems Online*, 2000.

[27] G. Czajkowski. Resource consumption management API. final release. http://jcp.org/en/jsr/detail?id=284, January 2009.

[28] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce. A resource management interface for the java platform. Technical Report TR-2003-124, Sun Microsystems, May 2003.

[29] A. Dearle. Software deployment, past, present and future. In *International Conference on Software Engineering*, pages 269–284. IEEE Computer Society, 2007.

[30] A. Dearle and S. Eisenbach, editors. *Component Deployment, Third International Working Conference, CD 2005*, volume 3798 of *LNCS*, Grenoble, France, November 2005. Springer-Verlag.

[31] G. Deng, J. Balasubramanian, W. Otte, D.C. Schmidt, and A. Gokhale. Dance: A qos-enabled component deployment and configuration engine. In *in Proceedings of the 3rd Working Conference on Component Deployment*, pages 67–82, 2005.

[32] M. des Jardins, J. MacGlashan, and J. Ferraioli. Interactive visual clustering. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*, pages 361–364, New York, NY, USA, 2007. ACM.

[33] Distributed Management Task Force, Inc. *CIM Tutorial*, June 2003.

[34] Distributed Management Task Force, Inc. *Understanding the Application Management Model*, June 2003. DSP0140, Version 1.2.

[35] Distributed Management Task Force, Inc. *CIM Schema*, 2008. Version 2.18.1.

[36] DMTF. *CIM Network Model White Paper*, December 2003.

[37] J. Dowling. *The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems*. PhD thesis, University of Dublin, Trinity College, October 2004.

[38] J. Dubus and P. Merle. Autonomous deployment and reconfiguration of component-based applications in open distributed environments. In *Proceedings of the 8th International OTM Symposium on Distributed Objects and Applications (DOAï£¡06)*, volume 4277 of *Lecture Notes in Computer Science*, pages 26–27, Montpellier, France, nov 2006. Springer-Verlag.

[39] H.J. Eisenmann and M. Frank. Generic global placement and floorplanning. In *DAC '98: Proceedings of the 35th annual conference on Design automation*, pages 269–274. ACM, 1998.

[40] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S.L. Price. Grid service orchestration using the business process execution language (BPEL). *Journal of Grid Computing*, 3(3–4):283–304, Semptember 2005.

[41] W. Emmerich and A.L. Wolf, editors. *Component Deployment, Second Internation Working Conference, CD 2004*, volume 3083 of *LNCS*, Edinburgh, UK, May 2004. Springer-Verlag.

[42] L. Epstein and A. Levin. On bin packing with conflicts. *SIAM Journal on Optimization*, 19(3):1270–1298, 2008.

[43] C. Erten, A. Efrat, D. Forrester, A. Iyer, and S. G. Kobourov. Force-directed approaches to sensor localization. In *Proceedings of 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 108–118, 2006.

[44] C. Perkins et al. *IP Mobility Support for IPv4*. The Internet Engineering Task Force, August 2002. Request for Comments: 3344.

[45] D.G. Feitelson, L. Rudolph, and U. Schwiegelshohn. Parallel job scheduling — a status report. In *Job Scheduling Strategies for Parallel Processing*, volume 3277/2005, pages 1–16. Springer Berlin / Heidelberg, May 2005.

[46] N. Férey, P.E. Gros, J. Hérisson, and G. Gherbi. Visual data mining of genomic databases by immersive graph-based exploration. In *GRAPHITE 2005, 3rd International Conference on Computer Graphics and Interctives Techniques In Australia and Southeast Asia*, 2005.

[47] A. Flissi, J. Dubus, N. Dolet, and P. Merle. Deploying on the grid with deployware. In *CCGRID*, pages 177–184. IEEE Computer Society, 2008.

[48] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual ogranizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[49] E. Foster-Johnson. *RPM Guide*. Red Hat, Inc., 2005.

[50] M. Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, January 2004.

[51] G.C. Fox, R.D. Williams, and P.C. Messina. *Parallel Computing Works*. Morgan Kaufmann Publishers, Inc., 1994.

[52] T.M.J. Fruchterman and E.M. Reingold. Graph drawing by force-directed placement. *Software — Practice and Experience*, 21(1):1129–1164, November 1991.

[53] A. Fuggeta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, ?(5):342–361, May 1998.

[54] P. Gajer, M.T. Goodrich, and S.G. Kobourov. A fast multi-dimensional algorithm for drawing large graphs. In *Graph Drawing: 8th International Symposium (GD'00)*, pages 211–221, 2000.

[55] C. Gavoille, R. Klasing, A. Kosowski, Ł. Kuszner, and A. Navarra. On the complexity of distributed graph coloring with local minimality

constraints. Research Report RR-1440-07, Laboratoire Bordelais de Recherche en Informatique, December 2007.

[56] A. Gokhale, K. Balasubramanian, and J. Balasubramanian et al. Model driven middleware: A new paradigm for deploying and provisioning distributed real-time and embedded applications. *Elsevier Journal of Science of Computer Programming: Special Issue on Model Driven Architecture*, 2004.

[57] D. Harel and Y. Koren. A fast multi-scale method for drawing large graphs. In *Graph Drawing: 8th International Symposium (GD'00)*, pages 183–196, 2000.

[58] M. Henning. Binding, migration, and scalability in corba. *Communications of the ACM*, 41(10):62–71, October 1998.

[59] A. Herrick. Java network launching protocol & API specification. http://jcp.org/en/jsr/detail?id=056, July 2008. Version 6.0.10, Maintenance Review Draft 4 Specification.

[60] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[61] IBM. An architectural blueprint for autonomic computing, June 2005.

[62] InstallShield Software and Zero G Software, Inc. A common deployment framework: Solution installation for autonomic computing, July 2004.

[63] A.-A. Ivan. *Partitionable Services Framework: Seamless Access to Distributed Applications*. PhD thesis, Department of Computer Science, New York University, September 2004.

[64] R. Kapitza, H. Schmidt, and F.J. Hauck. Platform-independent object migration in CORBA. In *OTM Conferences (1)*, pages 900–917, 2005.

[65] R. Kapitza, H. Schmidt, G. Söldner, and F.J. Hauck. A framework for adaptive mobile objects in heterogeneous environments. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1739–1756. Springer, 2006.

[66] T. Kichkaylo. Timeless planning and the component placement problem. In *ICAPS Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[67] T. Kichkaylo. *Construction of Component-Based Applications by Planning*. PhD thesis, Department of Computer Science, New York University, January 2005.

[68] M.-O. Killijian, J.-C. Ruiz-Garcia, and J.-C. Fabre. Portable serialization of CORBA objects: a reflective approach. In *OOPSLA*, pages 68–82, Seattle, USA, 2002.

[69] J. Kosiński. *Zarządzanie zasobami gridowymi z użyciem parawirtualizacji*. PhD thesis, AGH-University of Science and Technology, January 2009.

[70] E. Kotsovinos. *Global Public Computing*. PhD thesis, University of Cambridge; Computer Laboratory, JJ Thomson Avenue, Cambridge, UK, January 2005.

[71] T. Kowalczewski. Extensions of a CCM platform with support for continuous data streams. Master thesis, AGH-University of Science and Technology, al. Mickiewicza 30, 30-059 Kraków Poland, September 2008. Advisor: K. Zieliński; Consultation: J. Cała.

[72] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16:1293–1306, 1990.

[73] M. Kubale and B. Jackowski. A generalized implicit enumeration algorithm for graph coloring. *Communications of the ACM*, 28(4):412–418, 1985.

[74] S. Lacour, C. Pérez, and T. Priol. Deploying CORBA components on a computational grid: General principles and early experiments using the globus toolkit. In Emmerich and Wolf [41], pages 35–49.

[75] S.M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Also available at http://planning.cs.uiuc.edu/.

[76] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. The Internet Engineering Task Force, July 2005. Request for Comments: 4122.

[77] F. Lécué, A. Delteil, and A. Léger. Towards the composition of stateful and independent semantic web services. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 2279–2285, New York, NY, USA, 2008. ACM.

[78] P. Lee, C.-M. Wang, and J.-J. Wu. *High-Performance Computing: Paradigm and Infrastructure*, chapter Compiler and Run-Time Parallelization Techniques for Scientific Computations on Distributed-Memory Parallel Computers, pages 135–182. John Wiley and Sons Ltd., 2006.

[79] P. Lemaire, G. Finke, and N. Brauner. Models and complexity of multibin packing problems. *Journal of Mathematical Modeling and Algorithms*, 5:353–370, September 2006.

[80] X/Open Company Limited. *Distributed Transaction Processing: The XA Specification*. X/Open Company Ltd., February 1992.

[81] P. Maes. Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155, 1987.

[82] K. El Maghraoui. *A Framework for the Dynamic Reconfiguration of Scientific Applications in Grid Environments*. PhD thesis, Rensselaer Polytechnic Institute Troy, New York, April 2007.

[83] K. El Maghraoui, T.J. Desell, B.K. Szymanski, and C.A. Varela. Dynamic malleability in iterative MPI applications. In *Proc. 7th IEEE International Symposium on Cluster CCGrid2007*, Rio de Janerio, Brazil, May 2007.

[84] R. Maier. How OpenPKG can support IT business, 2006.

[85] M. Malawski. *Component-based Methodology for Programming and Running Scientific Applications on the Grid*. PhD thesis, AGH University of Science and Technology, Institute of Computer Science, al. Mickiewicza 30, 30-059 Kraków, Poland, 2008. Advisor: Jacek Kitowski.

[86] G. Malewicz, I. Foster, A.L. Rosenberg, and M. Wilde. A tool for prioritizing dagman jobs and its evaluation. *Journal of Grid Computing*, 5(2):197–212, June 2007.

[87] D. Marx. Graph colouring problems and their applications in scheduling. In *Periodica Polytechnica Ser. El. Eng.*, volume 48, pages 11–16, 2004.

[88] P.K. McKinley, S.M. Sadjadi, and E.P. Kasten et al. Composing adaptive software. *Computer*, pages 56–64, 2004.

[89] P. Merle, S. Leblanc, M. Vadet, F. Pilhofer, T. Ritter, and H. B ohme. *CORBA Component Model Tutorial*. OMG CCM Implementers Group, MARS PTC & Telecom DTC, Yokohama, Japan, April 2002. OMG TC Document ccm/2002-04-02.

[90] M. Mikić-Rakić. *Software Architectural Support for Disconnected Operation in Distributed Environments*. PhD thesis, University of Southern California, Los Angeles, CA, USA, December 2004. Adviser: Nenad Medvidovic.

[91] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 276–287, New York, NY, USA, 1997. ACM.

[92] OASIS. *Solution Deployment Descriptor Specification 1.0*, May 2008. sdd-spec-v1.0-cs01.doc.

[93] The OASIS Research Team and ActiveEon Company. *ProActive Programming*, April 2009.

[94] The OASIS Research Team and ActiveEon Company. *ProActive Scheduling*, April 2009.

[95] Object Management Group, Inc. *Externalization Service Specification*, April 2000. Version 1.0.

[96] Object Management Group, Inc. *Property Service Specification*, April 2000. Version 1.0.

[97] Object Management Group, Inc. *Life Cycle Service Specification*, September 2002. Version 1.2.

[98] Object Management Group, Inc. *Transaction Service Specification*, September 2003. Version 1.4.

[99] Object Management Group, Inc. *Streams for CCM*, July 2005. Final Adopted Specification.

[100] Object Management Group, Inc. *Deployment and Configuration of Component-based Distributed Applications Specification*, April 2006. Version 4.0.

[101] Object Management Group, Inc. *Quality of Service for CORBA Components, Beta 2*, September 2007.

[102] Object Management Group, Inc. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1; Part 1: CORBA Interfaces*, January 2008.

[103] Object Management Group, Inc. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1; Part 3: CORBA Component Model*, January 2008.

[104] Object Management Group, Inc. *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification*, April 2008. Version 1.1.

[105] K. Palacz. Application isolation API specification. final specification. http://jcp.org/en/jsr/detail?id=121, June 2006.

[106] P. Pissias and G. Coulson. A framework for quiescence management in support of reconfigurable multithreaded component-based systems. Computing Department, Lancaster University, UK, 2008.

[107] A. Pugliese, D. Talia, and R. Yahyapour. Modeling and supporting grid scheduling. Technical Report TR-0056, CoreGRID — Network of Excellence, August 2006.

[108] V. Quéma, R. Balter, L. Bellissard, D. Féliot, A. Freyssinet, and S. Lacourte. Asynchronous, hierarchical, and scalable deployment of component-based applications. In Emmerich and Wolf [41], pages 50–64.

[109] A.J. Quigley. Experience with FADE for the visualization and abstraction of software views. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 11, Washington, DC, USA, 2002. IEEE Computer Society.

[110] A.J. Quigley and P. Eades. FADE: Graph drawing, clustering and visual abstraction. In *Graph Drawing*, volume 1984/2001, pages 77–80, 2000.

[111] T. Ritter, R. Schreiner, and U. Lang. Integrating security policies via container portable interceptors. *IEEE Distributed Systems Online*, 7(7):1, 2006.

[112] S. Sicard, F. Bover, and N. de Palma. Using components for architecture-based management. In *ICSE'08: Proceedings of the 30th international conference on Software engineering*, pages 101–110, New York, NY, USA, 2008. ACM.

[113] A.C. Sodan. Loosely coordinated coscheduling in the context of other approaches for dynamic job scheduling: a survey. *Concurrency Computat.: Pract. Exper.*, 17(15):1725–1781, 2005.

[114] R. Sterritt, M. Parashar, H. Tianfield, and R. Unland. A concise introduction to autonomic computing. *Advanced Engineering Informatics*, pages 181–187, 2005.

[115] V. Subramonian, G. Deng, C. Gill, J. Balasubramanian, L.J. Shen, W. Otte, D.C. Schmidt, A. Gokhale, and N. Wang. The design and

performance of component middleware for qos-enabled deployment and configuration of dre systems. *Journal of Systems and Software*, 80(5):668–677, 2007.

[116] Sun Microsystems, Inc. *JDK 6 Documentation*, 2006.

[117] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA95054 USA. *Sun N1 Service Provisioning System 5.2. Plan and Component Developerï£¡s Guide*, April 2006. Part No: 819ï£¡4446ï£¡10.

[118] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, CA95054 USA. *Sun N1 Service Provisioning System 5.2. Plug-in Development Guide*, April 2006. Part No: 819ï£¡4448ï£¡10.

[119] C. Szyperski. *Oprogramowanie komponentowe; obiekty to za mało*. WNT, Warszawa, Polska, 2001.

[120] V. Talwar, D. Milojicic, Q. Wu, C. Pu, W. Yan, and G. Jung. Approaches for service deployment. *IEEE Internet Computing*, 9(2):70–80, 2005.

[121] R. Tamassia. Handbook of graph drawing and visualization. Online, not finished yet, 2008.

[122] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.

[123] VMWare, Inc. Virtualization overview. http://www.vmware.com, 2006.

[124] W3C. *Installable Unit Deployment Descriptor Specification*, June 2004. Version 1.0.

[125] N. Wang and C. Rodrigues. *Tutorial on CORBA Component Model*. DOC Group, Washington University St. Louis, July 2003.

[126] N. Wang, D.C. Schmidt, and C. O'Ryan. Overview of the CORBA component model, 2000.

[127] X. Wang, W. Li, H. Liu, and Z. Xu. A language-based approach to service deployment. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, pages 69–76, Washington, DC, USA, 2006. IEEE Computer Society.

[128] P. Watson, C. Fowler, C. Kubicek, A. Mukherjee, J Colquhoun, M. Hewitt, and S. Parastatidis. Dynamically deploying web services on a grid using dynasoar. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 151–158, Washington, DC, USA, 2006. IEEE Computer Society.

[129] Xen Source, Inc. Xen: Enterprise grade open source virtualization. http://xen.xensource.com, 2006.

[130] S. Zaidenberg, P. Reignier, and J.L. Crowley. An architecture for ubiquitous applications. *Ubiquitous Computing and Communication Journal*, 4, 2009.

Acronyms

AC Autonomic Computing

ACP Application Configuration Problem

ADF Adaptive Deployment Framework — the framework for adaptive deployment of distributed systems that resulted from the research presented in this work.

ADL Architecture Description Language

AJAX Asynchronous Javascript And XML

AM Autonomic Manager

AOM Active Object Map

API Application Programming Interface

APT Advanced Packaging Tool

ART Asymmetric Ray Tracing — one of our applications used for testing the ADF.

BPP Bin Packing Problem

BFS Best-First Search

BPEL Business Process Execution Language

CBSE Component-Based Software Engineering

CCA Common Component Architecture

CCM CORBA Component Model

CIM Common Information Model

CLR Common Language Runtime

CMIP Common Management Information Protocol

CMS Component Migration Service

COA Component-Oriented Architecture

COPI Container Portable Interceptor

CORBA Common Object Request Broker Architecture

COTS Commercial Off-The-Shelf

CPP Component Placement Problem

CPU Central Processing Unit

DAG Direct Acyclic Graph

DAnCE Deployment And Configuration Engine

DCF Distributed Coordination Function

D&C Deployment and Configuration of Component-based Applications — the specification proposed by OMG in [100] which is a basis for the model-based deployment approach used in this work.

DMI Desktop Management Interface

DMTF Distributed Management Task Force

DRE Distributed Real-time and Embedded

DSM Domain-Specific Modeling

DSRG Distributed Systems Research Group — the research group which the author is a member of.

DTD Document Type Definition

DTP Distributed Transaction Processing

ECA Event Condition Action

EJB Enterprise Java Beans

FDA Force-Directed Algorithm

FDDP Force-Directed Deployment Planning — proposed in this work, a novel approach to deployment planning based on FDA.

GCM Grid Component Model

GUI Graphical User Interface

HLA Home Location Agent

HTTP Hyper-Text Transfer Protocol

IBM International Business Machines

ICE Internet Communication Engine

IDL Interface Definition Language

IOS Internet Operating System

IP Internet Protocol

IUDD Installable Unit Deployment Descriptor

J2EE Java 2 Enterprise Edition

JEE Java Enterprise Edition

JMX Java Management eXtension

JNLP Java Network Launching Protocol

JRE Java Runtime Environment

JVM Java Virtual Machine

LAN Local Area Network

MAPE Monitor, Analyse, Plan and Execute

MDA Model Driven Architecture

MOM Message-Oriented Middleware

MPI Message Passing Interface

MSI Microsoft Installer

N1 SPS N1 Service Provisioning System

OMG Object Management Group

ORB Object Request Broker

OS Operating System

OTS Object Transaction Service

PCF Point Coordination Function

PCM Process Checkpointing and Migration

POA Portable Object Adapter

PC Personal Computer

PI Portable Interceptor

PIM Platform Independent Model

PSM Platform Specific Model

QoS Quality of Service

REST Representational State Transfer

RPC Remote Procedure Call

RIA Rich Internet Application

RPM RPM Package Manager

RTT Round Trip Time

SaaS Software as a Service

SALSA Simple Actor Language System and Architecture

SCA Service Component Architecture

SDD Solution Deployment Descriptor

SGE Sun Grid Engine

SNMP Simple Network Management Protocol

SOA Service-Oriented Architecture

SOAP Simple Object Access Protocol

STCM Spatio-Temporal Component Model

TCP Transmission Control Protocol

UML Unified Modeling Language

URL Uniform Resource Locator

UUID Universally Unique IDentifier

YUM Yellowdog Updater Modified

VLSI Very Large Scale of Integration

VM Virtual Machine

WBEM Web-Based Enterprise Management

WCF Windows Communication Foundation

WDL Wrapping Description Language

WS Web Services

WSDL Web Services Description Language

XML eXtensible Markup Language