Caspian: A QoS-Aware Deployment Approach for Channel-based Component-based Applications

Abbas Heydarnoori
David R. Cheriton School of Computer Science,
University of Waterloo,
Waterloo, ON, N2L 3G1, Canada
aheydarnoori@uwaterloo.ca

Technical Report - CS-2006-39

October 2006

Abstract

With significant advances in software development technologies in recent years, it is now possible to have complex software applications that include a large number of heterogeneous software components distributed over a large network of computers with different computational capabilities. To run such applications, their components must be instantiated on proper hardware resources in their target environments so that requirements and constraints are met. This process is called *software deployment*. For large, distributed, component-based applications with many constraints and requirements, it is difficult to do the deployment process manually and automated tools and techniques are required. This report presents a graph-based approach for this purpose that is not dependent on any specific component technology and does the deployment planning with respect to the communication resources, i.e. *channels*, required by application components and communication resources available on the hosts in the target environment. In our approach, component-based applications and distributed environments are modeled with the help of graphs. Deployment of an application is then defined as the mapping of the application graph to the target environment graph.

Contents

Chapter 1

Introduction

A few decades ago, software systems were only stand-alone systems, without any connections to other software systems. However, with significant advances in software development technologies, it is now possible to have complex software systems that consist of a large number of heterogeneous components distributed over many hosts with different hardware and software characteristics. However, in these applications, different components of the application may have various hardware and software requirements, and hence they may provide their desired functionality only when these requirements are answered. Furthermore, different hosts in the distributed environment may have different computational capabilities; making it impossible to install any kind of software components on them. Consequently, after the development of an application, a sequence of related activities must be done to place its components into the suitable hosts in the distributed environment, and to make the application available for use. This sequence of activities is referred to as *software deployment process*, and includes the following activities:

1. *Acquiring:* In this activity, the developed application is acquired from the software producer and is put in a software repository to be deployed.

2. *Planning:* This activity specifies different components of the application should be installed where in the distributed environment; resulting in a deployment plan.

3. *Installation:* In this activity, the deployment plan is used to install the application into the target environment.

4. *Configuration:* The deployed application is being configured in this activity.

5. *Execution:* After installing and configuring the application, it can be executed.

For simple stand-alone software systems that should be deployed only to a single computer, deployment activities can be easily done manually. But, suppose a complex component-based application is being deployed into a large distributed environment so that some QoS parameters, such as performance or reliability, are also optimized. In this situation, the deployment process is not so straightforward, and automated tools and techniques are required for this purpose. Consequently, the software deployment process has been given special attention both in research and industry in recent years, and it is possible to find many tools and papers addressing different activities of the software deployment process from different perspectives. However, to our knowledge, few if any of these deployment approaches notices the characteristics (e.g., behavior, cost, speed, security, etc.) of the interconnections among the components of the application. However, these characteristics have significant effects on the application's QoS. In this project, we intend to address this requirement. In particular, our focus is on the planning activity of this deployment process. In this project, we intend to design the required algorithms of an automated planner for the deployment of component-based applications into distributed environments that does the planning with respect to the properties of the interconnections among the application components. For this purpose, the concept of *channel* is used to model the interconnections among the application components. A channel is a peer-to-peer communication medium with well-defined characteristics and behavior [53]. Examples of channel-based models are Reo [53], MoCha [60], IWIM [61], and Manifold [62].

1.1 Deployment Planner Inputs

To generate deployment plans, the following inputs are to be specified for the planner:

1. *A specification of the channel-based, component-based application:* This specification specifies different components of the application and the channel types among them.

2. *A specification of the target environment:* This specification specifies available hosts in the distributed environment, the topology of the physical network among them, and the channel types that each host can support.

3. *A specification of the user-defined constraints and requirements:* Users may have special requirements and constraints regarding the deployment of the application that should be noticed during the deployment planning. For example, they may want certain QoS parameters to be optimized, or they may have certain constraints regarding the placement of the application components.

1.2 Deployment Planning

The deployment plan determines where different components of the application will be executed in the target environment so that all requirements and constraints are met. It is typically possible to deploy a complex component-based application into a large distributed environment in many different ways. However, when some QoS parameters are considered, some of these deployment configurations are better than others, and only a few of them may accommodate the constraints and requirements of the application. Thus, when QoS of the application is important, it should be tried to deploy the application so that its desired QoS parameter is optimized.

One naive solution for finding the best deployment configuration with the highest QoS is generating all possible configurations for the deployment of the application into the target environment and then, measuring the desired QoS parameter of each deployment configuration. Finally, the deployment configuration with the highest QoS is selected. However, when the number of possible deployment configurations is large, it is difficult to generate all of them. Thus, heuristic algorithms should be designed and applied to effectively solve this problem. For this purpose, in this project, a number of QoS parameters will be selected, and algorithms for effectively finding the deployment configurations with the highest desired QoS parameter will be designed.

1.3 Proposed Approach

A graph-based approach is currently used in this project to solve the software deployment problem. For this purpose, two graphs are made in this approach: the *Application Graph*, and the *Target Environment Graph*. The application graph models a channel-based component-based application as a graph of components connected by different channel types. The target environment graph models the distributed environment as a graph of hosts connected by different channel types that can exist between every two hosts. In other words, before starting the deployment planning, the channel types that can exist between every two hosts in the target environment are specified. Then, the deployment planning of an application is defined as the mapping of its application graph to its target environment graph, subject to optimization of the desired QoS parameter.

The approach of this work is general and is not dependent on any specific component technology or model (e.g., COM, CORBA, EJB, etc.) and can be used for deploying different kinds of component-based applications with different component technologies.

1.4 Outline

This report is organized as follows:

- Chapter 2 talks about the research fields that form the foundation of the presented work: software crisis and software engineering, software components, and the Reo coordination model.
- Chapter 3 provides a survey of the work done both in industry and academia in the area of component-based software deployment.
- Chapter 4 provides a formal description of the deployment problem we are solving in our ongoing research.
- Chapter 5 presents some algorithms for solving the deployment problem defined in Chapter 4 for QoS parameters cost and reliability when the target environment is a peer-to-peer distributed environment (e.g., Internet).
- Chapter 6 introduces our currently evolving deployment planner tool.

- Chapter 7 presents the conclusions and outlines the future work.

Chapter 2

Background

This chapter provides the required background of this technical report. This chapter is organized as follows. Section 2.1 talks about the software crisis and using the software engineering approaches during the software development process to overcome the software crisis. Then, Section 2.2 describes software components and introduces the component-based software development as a solution to software engineering problems. Finally, Section 2.3 provides a description of the Reo coordination model.

2.1 Software Crisis and Software Engineering

In 1968, NATO organized a conference to discuss the issues in software development projects in large corporations. The problem was that software projects were often underestimated in time and cost and about 80% of them were never completed. So, the phrase "*software crisis*" came into existence at this conference [1]. This phrase shows the lack of productivity and performance in software projects. It also shows that software developers are not capable of satisfying the needs of their customers and users [2]. In other words, software crisis is expressed by delays and failures in software projects tasks that result in low quality software, unpredictable costs and times, and unreached goals. The symptoms of software crisis are:

- Unacceptable quality of software;

- Not completion of software project within the estimated time and/or budget;
- Failure in software development project management;
- Abortion of software projects before completion.

There are many reasons for software crisis. Here are some of them [3]:

- No similar systems ever built before;
- Requirements are not well understood;
- Requirements change during the software life cycle;
- Software is too much flexible.

To overcome these problems, "*software engineering*" was first introduced by one of the study groups of NATO in 1967 [3]. The intent of software engineering is to use engineering principles and methods in the development of software projects. In other words, software engineering is an attempt to base the software development on an engineering approach with well defined inputs, well defined outputs and well defined methods [3]. The goal of software engineering is to produce systems that are correct, efficient, reliable, useable, maintainable, which satisfy their specification. Unfortunately, there are still some unsolved issues in the software engineering. Here are some of them:

- How to ensure the correctness, quality and maintainability of software systems?
- How to meet the estimated time and budget for the software project development?
- How to divide large systems into smaller manageable subsystems?
- How to ensure productivity of the software project development?
- How to answer the changing requirements during the software development?

Brooks in his classical paper on software crisis [4] mentioned some ways which can help us to get rid of these issues: high level languages, expert systems, software environments, incremental development, requirements refinement, prototyping, reuse and great designers [5]. In literature, some other solutions have been also proposed: outsourcing, open source, visual environments, CASE tools, software component repositories, object-oriented environments, and so on. However, none of these solutions can solve the software engineering problems definitely and there are some pros and cons with each of them. But, one of the most promising proposals is reusing existing software components in the development of new applications or component-based software development (CBSD). In the following section, we talk about software components in more detail.

2.2 Software Components

Making a system out of existing components is a common approach used in many engineering disciplines. The success of this approach in other engineering disciplines encouraged software engineers to use this idea in the software design too and thus, component-based software development methodologies came into existence. Two main reasons can be considered for this: (1) many software systems include similar or identical components and there is no need to redevelop them from scratch, and (2) because of the increasing complexity of software systems, it is becoming too expensive to develop them from scratch. However, there are a large number of different definitions for the term "software component" in literature. Here are some of them:

- Jose M. Troya and Antonio Vallecillo [6] believe that, components can be seen as encapsulation of programs. The "capsule" abstracts the program functionality, offers a common interface to the program services, hides their implementation and allows the composition and coordination of components.

- Alan W. Brown and Keith Short [7], characterize a component as an independently deliverable set of reusable services.

- H. John Reekie and Edward A. Lee [8] defined a component as a piece of software that can be plugged into some other piece of software. In their definition, a software component has clearly defined characteristics and

clearly defined behaviour in the domain of interest. Also, they told that, the context of production and consumption maybe quite different.

- Sherif Yacoub and et al. [9] defined a component as an independent replaceable part of the application that provides a clear distinct function. In their definition, a component is a unit of composition with predefined dependencies on other components.

- Clemens Szyperski in his book [10] introduced a component as a unit of composition with contractually specified interfaces and explicit context dependencies only. Also, he told that a software component can be deployed independently and is subject to composition by third parties.

- Wojtek Kozaczynski [11] provided the following definition: A component is a non-trivial, nearly independent and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Though these are different definitions of software components, there is a common idea behind all of them. We observe that in most of the existing definitions of software components, a software component exhibits the following properties:

- It is a unit of software implementation that can be reused in different applications;

- It has one or more predefined interfaces;

- The internal details of the software component are hidden;

- It does a specific function.

2.2.1 Software Components: Pros and Cons

In this section, we describe some of the advantages and disadvantages of using software components in the software development process. Some of the advantages of using software components in the software development process are the following:

- ***Modularity:*** One of the main benefits of using software components is the addition of "divide and conquer" or modularity to the software development process. By using software components, a big problem can be divided into smaller ones and then, for each subproblem, we can check whether there exists a software component for solving that subproblem or not. If yes, it can be used in the development process.

- ***Reliability:*** Software components may have been reused several times in different situations, and they may have been tested under a large variety of conditions. Thus, fewer errors would occur in the software application, and it is possible to have more reliable software systems [12, 13].

- ***Productivity:*** By using high quality software components, higher quality software systems can be developed within lower times, costs and effort. Thus, it increases efficiency and productivity.

- ***Maintainability:***If the interface of a component does not change, a component implementation can be simply substituted by a newer version of that component. Thus, a software application which is built by software components is more maintainable. Also, it is possible to add new functionalities to the application over time by adding new components to the existing application to answer changing requirements [10, 14]. Furthermore, because of the increased reliability of the software system, fewer errors would occur and maintenance costs would be reduced.

- ***Standardization:*** In CBSD, software components should follow a predefined standard to interoperate with each other in an application. Thus, another profit of software components is the standardization of software development process.

Although there are lots of advantages in using software components, there are also some disadvantages and problems too:

- ***Finding suitable components:*** One of the main difficulties in using software components is the difficulty to find suitable components. This raises risks such as using a software component which does not sufficiently satisfy the requirements regarding reliability, suitability, correctness and interoperability [15].

- ***Compositionality:*** This problem relates to the composition and packaging of components. The selected components might be compositional mismatch and it might be impossible to interconnect them successfully [16].

- ***Change:*** When the source code of the component is not available to the application developers (e.g., in the case of using COTS components), they do not have enough control over the component evolution. If that component does not satisfy their needs, they should ask the developer of that component to make the desired changes. This is often impossible. Since the level of support that is available from different component developers varies significantly, the adjustments to the components become much slower.

- ***Hidden dependencies:*** There might be some hidden dependencies among software components which software developers are unaware of them [17].

- ***Component development time and cost:*** The required time and cost for developing a reusable software component is much higher than the required time and cost for developing a special purpose software [18].

2.3 Reo Coordination Model

Reo is a channel-based coordination model that exogenously coordinates the cooperative behavior of component instances in a component-based system. From the point of view of Reo, a system consists of a number of component instances communicating through connectors that coordinate their activities. The emphasis of Reo is on connectors, their composition and their behavior. Reo does not say much about the components, whose activities it coordinates. In Reo, connectors are compositionally constructed out of a set of simple channels. Thus, channels represent atomic connectors. A channel is a communication medium which has exactly two channel ends. A channel end is either a *source* channel end or a *sink* channel end. A source channel end accepts data into its channel. A sink channel end dispenses data out of its channel. Although every channel has exactly two ends, these ends can be

of the same or different types (two sources, two sinks, or one source and one sink).

In Reo, a connector is represented as a graph of nodes and edges such that: zero or more channel ends coincide on every node; every channel end coincides on exactly one node; and an edge exists between two (not necessarily distinct) nodes if and only if there exists a channel whose channel ends coincide on those nodes.

2.3.1 Reo Operations

Reo defines two sets of operations. The first set, relates to the manipulation of the connector topology: *create*, *forget*, *join*, *split*, and *hide*. The *create* operation creates a channel of some defined type. With the *forget* operation a component instance tells Reo that it does not need a channel end anymore. The *join* operation allows joining of two nodes, each identified by one of the channel ends. The *split* operation then splits a node into two nodes by specifying the channel ends that the performer requires to coincide on the new nodes. The *hide* operation allows the performer to protect the topology of a node.

The second set of operations defined by Reo enable component instances to connect to and perform I/O on source and sink nodes. These operations are: *connect*, *disconnect*, *wait*, *read*, *take*, *write*, and *move*. The *connect* operation connects the performer to a channel end by providing exclusive access to the node (and thus to all of its coincident channel ends) on which this channel end coincides. The *disconnect* operation releases a previously established connection. The *wait* operation allows the performer to wait for some condition on a channel end. The *read* operation allows the performer to non-destructively read data from a sink. The *take* operation does the same as read but it also removes the data from the sink. The *write* operation replicates its value and atomically writes a copy of its value to every source channel end that coincides on the source node on which it is performed. The *move* operation allows the performer to move a channel end to another location. Note that changing location does not change the topology of the connector or the connection status of the moved channel end.

2.3.2 A Useful Set of Primitive Channels

Reo assumes the availability of an arbitrary set of channel types, each with well-defined behavior provided by the user. However, a set of examples in [53] show that exogenous coordination protocols that can be expressed as regular expressions over I/O operations correspond to Reo connectors which are composed out of a small set of only five primitive channel types:

- *Sync:* It has a source and a sink. Writing a value succeeds on the source of a *Sync* channel if and only if taking of that value succeeds at the same time on its sink.
- *LossySync:* It has a source and a sink. The source always accepts all data items. If the sink does not have a pending read or take operation, the LossySync loses the data item; otherwise the channel behaves as a *Sync* channel.
- *SyncDrain:* It has two sources. Writing a value succeeds on one of the sources of a *SyncDrain* channel if and only if writing a value succeeds on the other source. All data items written to this channel are lost.
- *AsyncDrain:* This channel type is analogous to *SyncDrain* except that the two operations on its two source ends never succeed simultaneously. All data items written to this channel are lost.
- *FIFO1:* It has a source and a sink and a channel buffer capacity of one data item. If the buffer is empty, the source channel end accepts a data item and its write operation succeeds. The accepted data item is kept in the internal buffer. The appropriate operation on the sink channel end (read or take) obtains the content of the buffer.

2.3.3 An Examples of Reo Connectors

As an example of Reo connectors, Fig. 2.1 shows a *barrier synchronization* connector in Reo. In this connector, a data item passes from a to d only simultaneously with the passing of a data item from g to j and vice versa. This is because of the "replication on *write*" property in Reo. ab is a *Sync* channel. So, writing on a succeeds only if writing on the mixed node bec succeeds. This happens only when both cd and ef are capable of consuming

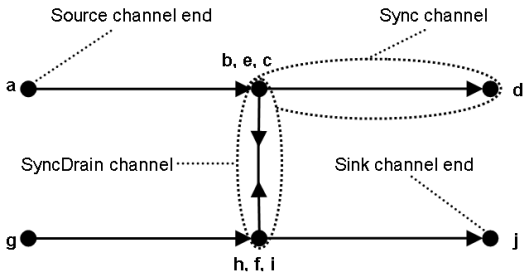

Figure 2.1: Barrier synchronization connector in Reo

a copy of the written data (definition of *write*). Similarly, writing on g succeeds only if writing on the mixed node *hfi* succeeds. Again, this happens only when both ij and fe are capable of consuming a copy of the written data. With respect to the semantic of the *SyncDrain* channel which both writes on its both source ends should be done at the same time, a data item passes from a to d only at the same time with the passing a data item from g to j.

Since the entities on a, d, g, and j do not need to know that they are communicating with each other and their communication is regulated, this connector is an example of exogenous coordination in Reo. In Reo, it is easily possible to construct different connectors by a set of simple composition rules out of a very small set of primitive channel types. One can find a more elaborate introduction to Reo in [54], and a detailed description of the language and its model in [53].

Chapter 3

Related Work

As mentioned in Chapter 1, there are typically many constraints in the deployment of large component-based applications into distributed environments. For this purpose, software deployment process is given special attention both in academia and industry, and it is possible to find a large number of tools, procedures, techniques, and papers addressing different aspects of the software deployment process from different perspectives. However, none of them is able to cover the full range of deployment activities. Furthermore, none of them is general enough so that it can be used for deploying all kinds of component-based applications with different component technologies. In this chapter, we provide a survey of existing deployment tools and techniques in industry and academia.

This chapter is organized as follows. In Section 3.1, a survey of software deployment tools which have been developed in industry is provided. Section 3.2 discusses various research approaches for component-based applications deployment. Finally, Section 3.3 presents another survey done by the University of Colorado's Software Dock research team.

3.1 Software Deployment Tools in Industry

A variety of tools and technologies exist in industry to support different software deployment activities mentioned in Chapter 1. In this section, we classify them into three main categories: *stand-alone installers*, *Web-based deployment tools*, and *systems management tools*. Then, the features of these categories are described and three sample tools are considered for each of

Deployment Technology	Sample Tool	Software Deployment Lifecycle				
		Acquiring	Planning	Installation	Configuration	Execution
Stan-alone Installers	Linux RPM			●		
	InstallShield			●		
	InstallAnywhere			●		
Web-based Deployment Tools	Java Web Start	●		●		●
	Windows Update	●		●		●
	Microsoft Clickonce	●		●		●
Systems Management Tools	Microsoft SMS		●	●	●	
	IBM TME-10		●	●	●	
	Altiris		●	●	●	

Table 3.1: Comparison of different industry-based deployment tools in terms of their support of deployment process

them. Table 3.1 represents these tools and characterizes them in terms of their support of software deployment process.

3.1.1 Stand-alone Installers

There are some tools whose main activities are installing and uninstalling stand-alone software systems from a single computer. In these tools, different files of the software system along with some semantic information about the files are packaged into a software package and is delivered to the user. Then, users use these tools to un-package this software package and install it on their intended machines. Linux RPM [19], InstallShield [20], and InstallAnywhere [56] are representatives of this kind of tools.

However, these tools have some limitations. As mentioned earlier, they can be used only for installing and uninstalling stand-alone applications from a single computer, and it is impossible to use them for distributed systems. Also, users themselves have to update their software systems whenever newer versions of those systems are available.

Linux RPM

The Redhat Package Manager or RPM is a powerful command line tool used for deployment of software packages in the Linux environment. Each RPM package contains an archive of files to be deployed along with some metadata information about the software package such as its version, cryptographic signatures for each file in the package to verify the integrity of the package, and so on. RPM supports the following activities: packaging, installing, verifying, updating, and removing of the software systems. RPM uses a number of Linux scripts to do these activities. RPM also maintains a database of all the software packages it has installed so far. Thus, information about the installed software systems is available at any time.

InstallShield

InstallShield for Windows is a commercial tool which is used widely for installing, reconfiguring, and removing Windows-based applications from a single site.

InstallAnywhere

Zero G Software Corporation's InstallAnywhere is a commercial tool that can be used by software developers to package a software system written in Java, C++, J2EE or .NET so that it can be installed or uninstalled from any major operating system (e.g., Windows, Linux, Solaris, HP-UX, Mac OS, NetWare, etc.). As its name shows, one of the main advantages of InstallAnywhere is that the developer does not need to package different distribution versions of a software system for different operating systems.

3.1.2 Web-based Deployment Tools

Web-based deployment tools try to use the connectivity and popularity characteristics of the Internet in the software deployment process. In these tools, it is not required to install or update the software system on every single host separately. Instead, the software application is deployed only to a single Web server. Then, client machines (users) connect to this server to download the application files or application updates automatically. Examples of this category of tools are Java Web Start [22], Microsoft Windows Update [23], and Microsoft ClickOnce [24]. However, one of the major limitations of these

tools is that they are used for stand-alone applications and it is impossible to use them for deploying distributed applications.

Java Web Start

Java Web Start provided by Sun Microsystems as a deployment tool for Java-based software systems that allows users to run and manage software applications right off the Web. Java Web Start guarantees that the user is always using the latest version of the application. At the first time that the application is being used, that application is downloaded from the Web server and is cached locally on the computer. Then, on each subsequent run of the application, Java Web Start checks the Web server to see whether or not there exists a newer version of the application. If yes, it automatically downloads the new version of the application and executes it.

Microsoft Windows Update

Microsoft Windows Update is a Web-based software update service for Microsoft Windows operating system. Whenever the user visits the Windows update site[1], it automatically scans the computer to see if any updates is available for the computer. If so, it downloads those updates automatically and applies them to the system. These updates usually help to protect against known security threats.

Microsoft ClickOnce

Microsoft ClickOnce is very similar to Sun Microsystems Java Web Start. It is part of the Microsoft .Net Framework (version 2.0) that allows the user to deploy Windows-based applications to a client computer by placing the application files on a Web or file server accessible to the client. Then, a link to that application is provided to the user. When user clicks this link on a Web page (or an email, etc.) the application files are downloaded to the user's computer and run. However, the subsequent executions of the application can be offline and it is not required to download the application files again. When a new version of the application is installed on the server, it can be automatically detected by connected clients, and updates can be downloaded and applied.

[1] http://update.microsoft.com

3.1.3 Systems Management Tools

The term systems management is typically used to describe a set of capabilities (e.g., tools, procedures, policies, etc.) that enable organizations to more easily support their hardware and software components throughout their life cycle [25]. More specifically, they help organizations to gather information about the existing hardware and software, make decisions to purchase new hardware and software, distribute and deploy them to wherever they should be, configure and maintain them with updates, and so on [26].

Systems management tools usually have a centralized architecture. In these tools, the IT administrator performs operations from a centralized location and it is applied automatically to many systems in the organization. So, the IT administrator is able to deploy, configure, manage, and maintain a large number of hardware and software systems from his/her own computer. Examples of these tools are Microsoft Systems Management Server [27], IBM Tivoli Management Environment [28], and Altiris Deployment Solution [29]. However, there are some limitations associated with these tools: they are often heavy and complex systems, they require reliable networks, and they require complete administration control.

Microsoft Systems Management Server

Microsoft Systems Management Server (SMS) can be used by IT administrators to manage, support, and maintain a distributed network of computer resources within their organization that are running Microsoft Windows operating system. SMS provides a comprehensive change and configuration solution for centrally managing client computers and servers, enabling organizations to provide relevant software and updates to users. SMS consists of comprehensive hardware inventory, software inventory and metering, software distribution and deployment, and remote troubleshooting tools. However, in this report, our focus is on its software deployment features. SMS provides the necessary tools to plan, test, analyze, and deploy software applications, enabling the enterprises to provide the necessary applications throughout the organization. For example, IT administrators can use the reports generated by hardware and software inventory to understand the status of the hardware and software assets and decide whether or not deploy a new application.

IBM Tivoli Management Environment

IBM Tivoli Management Environment (TME-10) is a set of management applications for managing a network of computing resources of many different types from a single point. In other words, TME-10 provides a consistent interface to different operating systems and services. It allows IT administrators to control users, systems, and applications from a single computer and provides some methods to automate time consuming tasks. One of its features is application deployment management. For this purpose, the same as Microsoft SMS, it maintains an enterprise-wide hardware and software inventory.

Altiris Deployment Solution

Altiris Deployment Solution makes it possible to remotely manage all types of hardware devices (e.g., notebooks, desktops, servers) within an organization's LAN or WAN. For example, distribute patches and drivers, image computer hard-drives, install or upgrade software systems, or migrate a large number of users to new computers while transferring their custom settings or installed programs.

3.2 Software Deployment Approaches in Research

In the previous section, we considered some of the software deployment tools and technologies that are currently available in industry. However, in recent years, component-based software deployment has been given special attention in research too and several approaches addressing the software deployment problem have been proposed. In this section, some of these approaches are discussed.

3.2.1 Deployment Frameworks

In this section, a number of research approaches are described such that they mention a sequence of activities for the software deployment process and try to provide a general framework for this purpose.

Software Dock

In the University of Colorado Software Dock research project, software deployment is defined as a collection of interrelated activities that form the *software deployment life cycle* [30]. This cycle includes the following activities: *release*, *install*, *activate*, *update*, *adapt*, *reconfigure*, *deactivate*, *remove*, and *retire*. These activities can be divided into two groups:

- **Producer-side Activities:**
  - *Release:* This includes all the tasks required to package, prepare, provide, and advertise a system for deployment to consumer sites. This activity acts as a bridge between development and deployment.
  - *Retire:* When a software system or a given configuration of a software system is no longer supported by the software producer, this activity is done.
- **Consumer-side Activities:**
  - *Install:* This activity configures and assembles all of the necessary resources for using a given software system.
  - *Activate:* This is responsible for running or executing a deployed software system.
  - *Deactivate:* This is responsible for shutting down any executing components of an activated software system.
  - *Update:* This modifies a previously installed software system and deploys a new, previously unavailable configuration of a software system.
  - *Adapt:* This activity maintains the consistency of the currently selected configuration of a deployed software system.
  - *Reconfigure:* Its purpose is to select a different configuration of a previously deployed software system from its existing semantic description.
  - *Remove:* this activity is performed when a software system is no longer required at a consumer site.

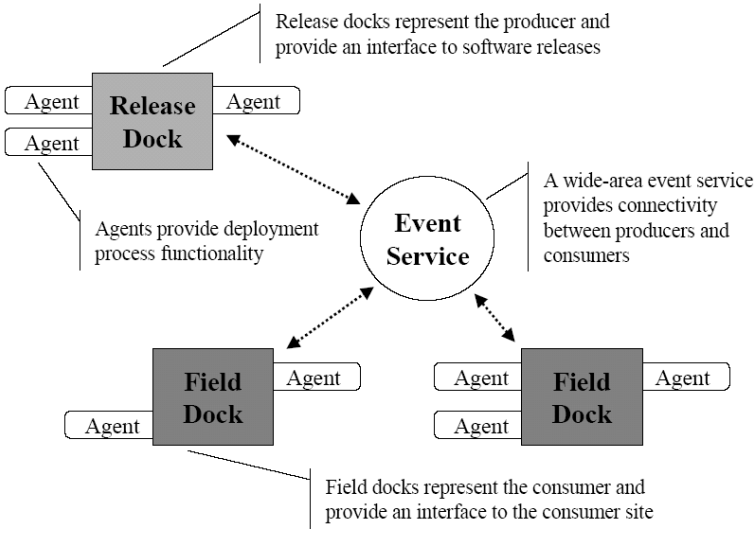

Figure 3.1: Software Dock architecture (taken from [31])

The Software Dock research project has created a distributed, agent-based deployment framework that supports cooperation among software producers themselves and between software producers and software consumers [31]. The Software Dock architecture is shown in Fig. 3.1. This architecture has the following components:

- *Release dock:* Its purpose is to serve as a release repository for the software systems provided by the software producer. In this repository, each software release is semantically described using a standard semantic schema: *Deployable Software Description* or *DSD*.
- *Field Dock:* It is a server residing at a consumer site and providing information about the resources and configuration of the consumer site.
- *Agents:* Each software release is accompanied by generic agents that perform software deployment processes with the help of interpreting the semantic description of the software release.
- *Wide-area event system:* The release dock generates events as changes are made to the software release that it hosts.

A description of the actual deployment by the architecture presented in Fig. 3.1 follows. When a software system is to be installed on a given

consumer site, initially an agent responsible for installing that software and the DSD description of that software are loaded onto the consumer site from the originating release dock. This agent docks at the local field dock and configures the software system using the DSD description of that software and the consumer site state information provided by the field dock. When this configuration is done, this agent asks the precise configuration that it requires from its release dock. It also may request other agents (such as update and adapt) from its release dock to come and dock at the local field dock and do other deployment activities. The wide-area event service provides a means of connectivity between software producers and software consumers. As we see, Software Dock uses mobile agents to do software deployment activities. We will talk more about using mobile agents in the process of software deployment later in section 3.2.2.

With the architecture presented in Fig. 3.1, Software Dock provides a comprehensive framework for deploying and configuring a software system into a single site using its DSD description. However, when the software system is large and is composed of many components, it may have many different configurations. But, with respect to some externally defined constraints, some of these configurations might be invalid. For example, combinations of specific versions of components are not acceptable. In this case, it is impossible to effectively enumerate all possible configurations manually. Another way is to analyze the DSD descriptions to detect potentially invalid configurations. In [32], D. Heimbigner et al. present their ongoing work on developing such a framework for analyzing DSD descriptions and detecting potentially invalid configurations with respect to externally defined constraints. In other words, this framework generates all of the possible configurations for a system using its DSD description and then applies specified analysis packages to each of those configurations to detect problems and conflicts with those configurations. Then, the result of this analysis could be fed back to the DSD descriptions to prevent future generation of invalid configurations.

ORYA - Open enviRonment to deploY Applications

Suppose you want to install different versions of the same application on many sites at the same time with respect to the characteristics of these sites. For this purpose, an automated tool is required and it is very hard to do this activity manually. ORYA deployment process is introduced in [33] to answer this requirement. In ORYA, the following entities are introduced as

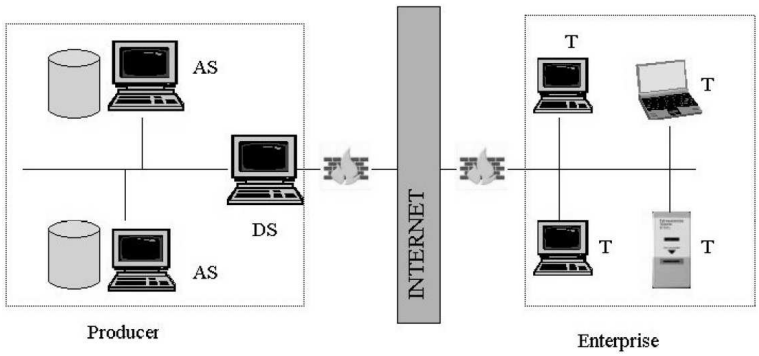

Figure 3.2: An example of ORYA deployment environment (taken from [33]))

the main entities of an automatic process for deploying applications on one or more sites:

- *Application Server (AS):* A repository of application packages. A package consists of a set of application files, a set of executable files for installing the application and a metadata file to specify the dependencies, constraints and features of the application.
- *Target (T):* The target in which the applications should be installed and run (e.g., a computer or a device). Each target is modeled by a site model. Site model represents which applications are already deployed into a target and what are its hardware properties (disk space, CPU, memory, etc.)
- *Deployment Server (DS):* this server finds the suitable packages to be deployed, transfers them to the target and installs them. DS should cope with several problems such as dependencies or shared components, or it should confirm that other existing applications continue to work.

Fig. 3.2 shows an example of the above mentioned entities. In this example, the producer side includes a number of application servers and one deployment server and is connected to the Internet via a firewall. Also, different targets which belong to an enterprise is connected to the Interne through a firewall.

Fig. 3.3 shows the deployment process being done by the deployment server to install an application into only one target. In the ORYA environment, there exist several application servers hosting several application

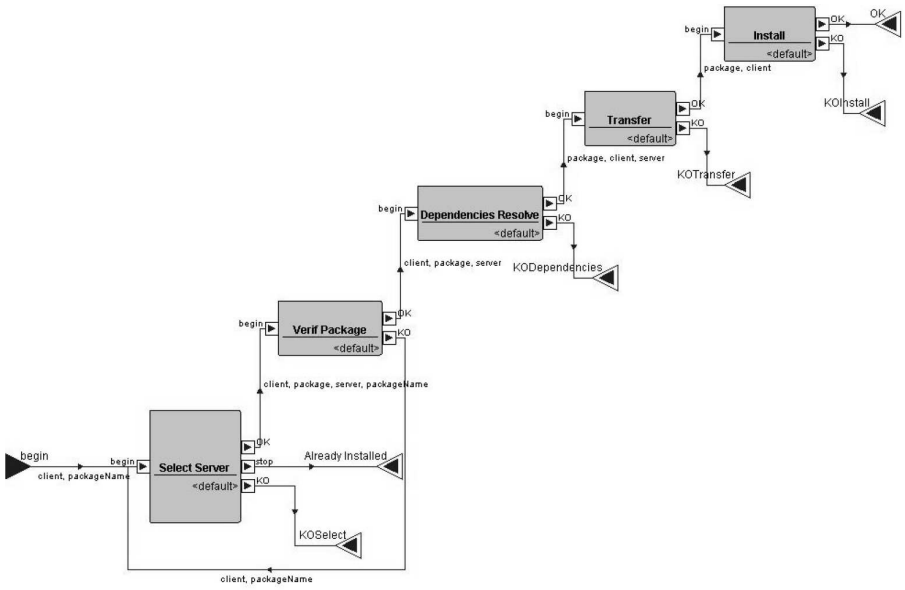

Figure 3.3: ORYA deployment process (taken from [33]))

packages. The deployment process should find the desired application package among many ones in such a way that the selected package will work truly and it will not affect other existing applications in the target. This process includes the following activities:

- *Select server:* Receives the application and target names as input and prepares the deployment process. For this purpose, first it checks whether or not that application already exists in the target. If yes, then the deployment process stops. Otherwise, it finds an application server containing that application package. Then, it considers whether this package can be deployed in the target. If not, another package will be searched. If no package is found, the deployment process terminates.

- *Verify package:* Verifies that the package can be deployed on the target. The focus of this activity is checking the hardware constraints (e.g., memory, disk space, etc.). Software constraints will be checked in the next activity.

- *Dependencies resolve:* Checks the software dependencies.

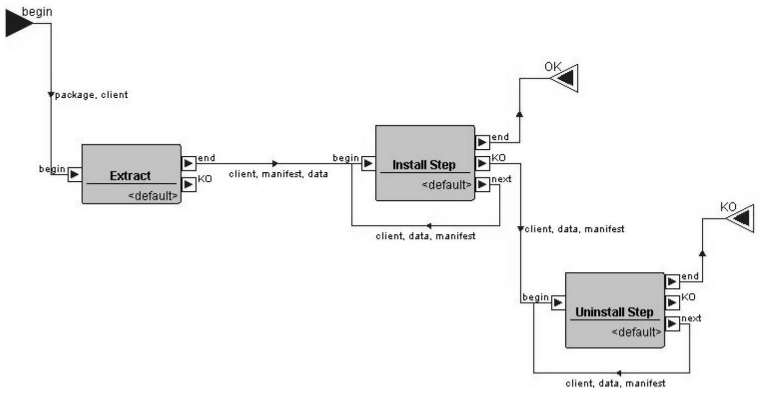

Figure 3.4: ORYA Install activity (taken from [33]))

- *Transfer:* Transfers the package from the application server to the target.
- *Install:* Does the physical installation. It is a composite activity and its constituent activities are shown in Fig. 3.4:
  - *Extract:* Extracts the package into a temporary folder on the target.
  - *Install step:* Each install step activity performs two basic activities: execution (executes the script related to the install step) and checking (performs the necessary verifications)
  - *Uninstall:* If some problems occur during the installation, this activity returns the target to a consistent state and undoes all already modifications.

OMG Deployment and Configuration Specification

The "OMG Deployment and Configuration Specification" or "OMG D&C Specification" is an attempt towards unified deployment of component-based applications into distributed environments [34]. The deployment process defined in this specification consists of five stages. But, before starting the deployment process, the metadata describing the software and the binary compiled code artifacts should be combined into a software package by the software producer. Then, this package is published by the producer and

acquired by the user (e.g., via Internet). Then the following deployment process can be started:

- *Installation:* During installation, the published software package is acquired by the user and is put into a repository of software components. This activity does not involve transfer of binary files to the hosts in which software components will actually execute.
- *Configuration:* When the software is installed in the repository, its functionality can be configured.
- *Planning:* After a software package has been installed into a repository and configured, deployment planning of the application can be started. This planning involves selection of hosts on which the software will run, the resources it will require to run, deciding which implementations will be used for component instances, and so on. This stage results in a deployment plan.
- *Preparation:* This activity prepares the target environment for execution of the software. For example, transfers binary files to the specified hosts in the target environment on which the software will run.
- *Launch:* In this stage, the application is executed. As planned, component instances are created and configured on hosts in the target environment and the connections among the component instances are established.

The OMG D&C Specification defines three platform independent models, the *component model*, the *target model*, and the *execution model*. Each of these models is also split into the data model and the runtime (management) model to reduce the complexity. The runtime models deal with runtime entities and they are outside the scope of this report. Below are brief descriptions of data models:

- *Component Data Model:* The UML diagram for Component Data Model is presented in Fig. 3.5. This model shows the information about installed and configured software packages in the software component repository. In this model, a software package may contain several implementations of the same component (e.g., to work with different operating systems). The component itself has an interface composed

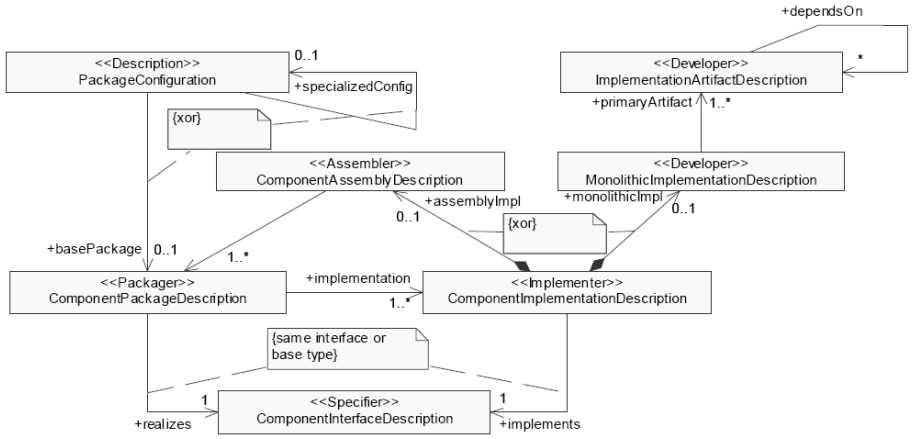

Figure 3.5: OMG component data model (taken from [34]))

of operations, attributes, and ports that may be connected to other components. The implementation of the component could be either monolithic, or an assembly of other components.

- *Target Data Model:* This model describes the target environment in which the application can be deployed. The UML diagram of this model is depicted in Fig. 3.6. The top level entity of this model is *Domain* which itself is composed of Node, Interconnect, Bridge, and SharedResource. Nodes have computational capabilities and will run component instances. Interconnects provide direct connections among nodes and connections among components will be deployed on them. Bridges act as router among interconnects and thus, provide indirect connections among nodes. SharedResources are those resources which are shared among nodes.

- *Execution Data Model:* Before starting to deploy the software system into the target environments, it should be decided which implementations to select (if there are several implementations of the same component in the package) and where to deploy each monolithic component implementation. The result of this decision making process is collected in a *Deployment Plan*. Actually, Execution Data Model is this Deployment Plan. The UML diagram of this model is shown in Fig. 3.7. In this diagram, Artifact Deployment Description specifies an artifact that is being deployed as part of the plan; Monolithic Deployment De-

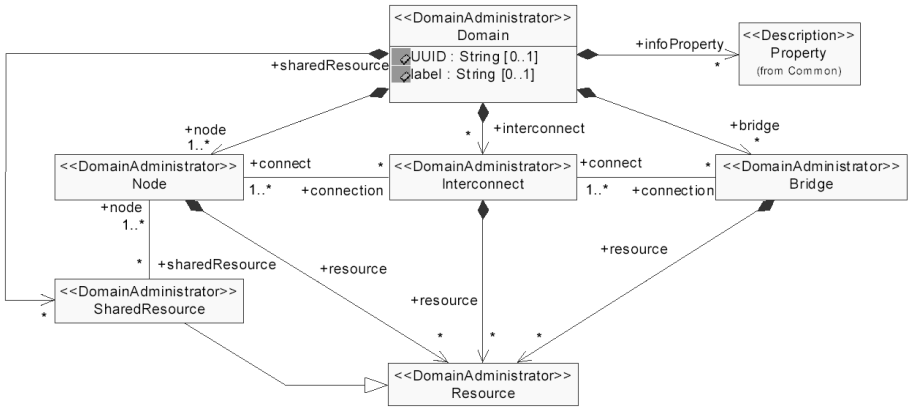

Figure 3.6: OMG target data model (taken from [34]))

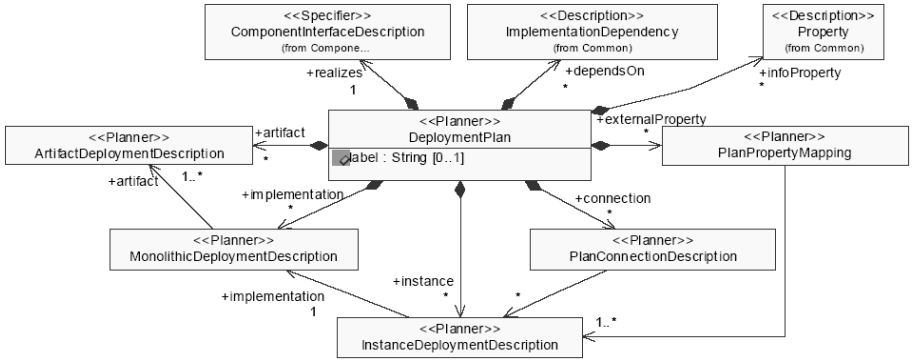

Figure 3.7: OMG execution data model (taken from [34]))

scription describes how to create a component instance; Instance Deployment Description specifies where to instantiate it; Plan Connection Description includes information about connections among component instances. This model also includes information about the component interface which is realized by this software system.

As mentioned earlier, these models are platform independent. In order to use them with a specific component model, they should be transformed to platform dependent models, capturing the specifics of the concrete platform [35]. An example of this transformation to the CORBA Component Model (CCM) can be found in [34].

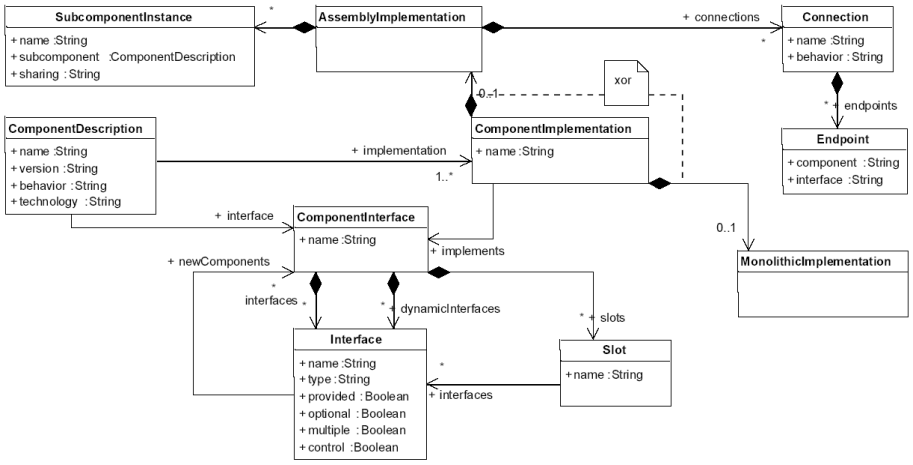

Figure 3.8: New component data model (taken from [36])

However, as mentioned earlier, OMG D&C Specification is an effort towards a unified framework for deploying component-based applications into distributed environments. But, P. Hnetynka in [36] and [37] mentions that OMG's approach is not suitable for building a single environment for unified deployment of component-based applications and it leads to several deployment environments. He mentions that in order to have a unified environment for this purpose, a generic component model that covers all the component technologies is required. In order to develop such a generic component model, he examined several component models and ADLs: CORBA Component Model (CCM) [38], Enterprise Java Beans (EJB) [39], Fractal [40], SOFA [41], Darwin [42], Wright [43], and ACME [44]. After considering these models and languages, Hnetynka noticed that a number of component features are missing in the OMG D&C Component Data Model. But, they are crucial for developing a unified deployment environment. These features include: Component technology, Control interfaces, Groups of interfaces, Behavior, Versioning, and Dynamic changes of applications at runtime. Thus, he developed a new component data model which includes these features. This model is shown in Fig. 3.8.

Another issue in the OMG D&C Specification relates to its support of heterogeneous component-based applications. This kind of applications can constitute components written using different component models (EJB, CORBA, Fractal, SOFA, etc.). Since, different component models typically

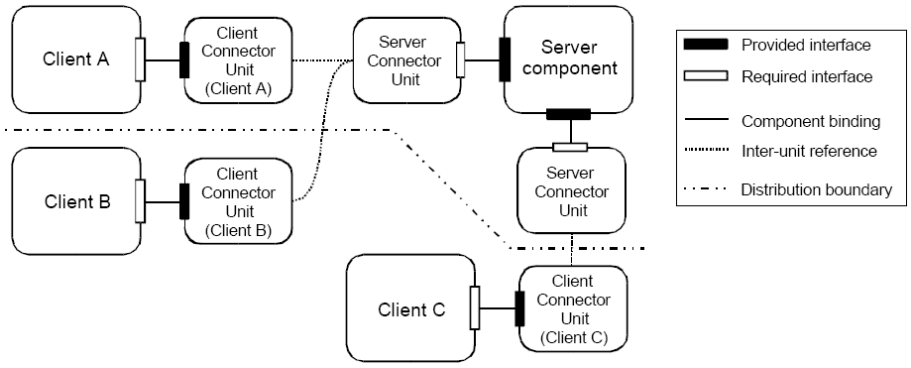

Figure 3.9: Using connectors for capturing the interactions among components (taken from [35])

have different communication middleware and their communications have different semantic, one of the main issues in making components from different component models to work together is their communication. To solve this problem, L. Bulej and T. Bures in [35] propose to use software connectors [45] to describe the semantics of connections between components from different component models. A connector itself can be viewed as a number of connector units attached to the corresponding components (Fig. 3.9). In order to use connectors, an appropriate connector should be instantiated between every two components. For this purpose, whenever a component is instantiated, for each of its interfaces a server connector unit is created and whenever an interface is being connected to another component, a client connector unit is instantiated and bound to the corresponding server connector unit. Then, the communication between components are done through these connector units.

In [35], the OMG D&C Specification is extended to support deployment of heterogeneous component applications by introducing connectors as bridges between the heterogeneous parts of an application, and by extending the model to support construction of connectors during the deployment. For this purpose, some slight changes have been made on the Component Data Model presented in Fig. 3.5. More specifically, the Assembly Connection Description in the component data model class have been extended with another association named ConnectionRequirement. This association makes available the information about the connection requirements for the deployment planner.

3.2.2 Using Mobile Agents in Software Deployment

In section 3.2.1, we talked about University of Colorado Software Dock research project and we mentioned that mobile agents are used in this project to do deployment activities. In this section, we talk about the concept of using mobile agents in the software deployment process in more detail. In [46] a mobile agent is defined as an object which migrates through many hosts in a heterogeneous network, under its own control, in order to perform tasks using resources of those hosts. In other words, a mobile agent can perform its task autonomously without any independence to the application generated it. A Mobile Agent System or MAS is defined as a computational framework that implements the mobile agent paradigm [47]. This framework provides services and primitives that help in the implementation, communication, and migration of software agents.

A. Carzaniga et al. in [30] mention that there are some issues and challenges in the software deployment process presented in the Software Dock research project (section 3.2.1). In the following, we consider some of them:

- *Large-scale Content Delivery:* It refers to transferring a large volume of software packages from producers to consumers.

- *Heterogeneous Platforms:* Nowadays, it is possible to have a network of heterogeneous hardware platforms (e.g., Mainframes, PCs, etc.), all possibly running different operating systems. As a result, this makes new challenges for the software deployment process.

- *Integration with the Internet:* With the arrival of the Internet, there exists a virtual, worldwide marketplace. So, producers can advertise their products and customers can evaluate those products. Thus, the deployment process should be tightly integrated with the Internet.

- *Security:* The use of Internet as a deployment media causes a variety of security concerns (e.g., privacy, authentication, and integrity concerns). More specifically, since installation or update activities have access to system resources, it is very important to monitor these activities during the deployment process.

In [47], some ideas of using mobile agents to deal with these issues are presented:

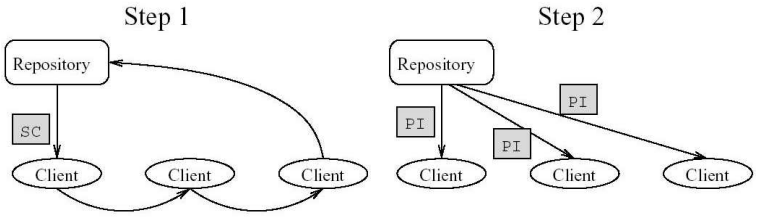

Figure 3.10: TACOMA software deployment architecture (taken from [48])

- *Large-scale Content Delivery:* Mobile agents can prevent the transfer of already installed components by requesting just the necessary components from the consumer host using the local description of the consumer site.

- *Heterogeneous Platforms:* If mobile agent systems use interpreted languages as Java, then those systems can cope with the hardware heterogeneity and the same mobile agent can be executed in different hardware/software platforms.

- *Integration with the Internet:* Again, MASs can be implemented in Java and then they can communicate with each other by CORBA or RMI. These middleware layers are implemented on top of the standard Internet protocols.

- *Security:* The deployment of software as a mobile agent allows the system to monitor installation procedures, and preventing illegal operations.

As another work of using mobile agents in the software deployment process, N.P. Sudmann and D. Johnson in [48] show how mobile agents can be used for updating or installing software components in a distributed environment such as Internet. In their paper, they talk about a middleware toolkit called TACOMA which is a mobile agent system built to support software deployment over the network.

Software deployment infrastructure in TACOMA includes one or more repositories that contain the latest versions of software packages. These packages can be used to update or install new software at a number of hosts that have subscribed to the repository service. Fig. 3.10 shows the software

deployment architecture of TACOMA. As shown in this figure, it has two different agents: the state collector agent (*SC*) and the installer agent (*PI*). *SC* agent probes all hosts and collects information about the required updates for each host. Then, it returns to the repository. After that, with respect to the information collected by the *SC* agent, one *PI* agent will be generated for each host. These *PI* agents include the required packages for their intended hosts along with the install logic of those packages. Then, these *PI* agents will travel to their intended hosts in parallel and do the actual installations or updates.

3.2.3 QoS-Aware Deployment

Typically, it is possible to deploy a large component-based application which constitutes a large number of components into a distributed environment in many different ways. Obviously, some of these deployment configurations are better than others in terms of some quality of service (QoS) attributes as efficiency, availability, reliability, fault tolerance, and so on. Thus, the deployment configuration has significant effects on the system behavior and it is necessary to consider the issues related to the quality of a deployment. M. M. Rakic et al. in [49] claim that existing tools for showing software deployments lack support for specifying, visualizing, and analyzing different factors that influence the quality of a deployment and they try to provide such an environment to answer this requirement for large-scale, highly distributed systems. They named their environment DeSi.

As mentioned earlier, it is possible to consider different quality attributes for a deployment. However, in [49], the emphasize is on the "availability" and it is defined as the ratio of the number of successfully completed inter-component interactions in the system to the total number of attempted interactions over a period of time. Because of the so many parameters that influence the availability of a system, it is too much hard to find the best deployment architecture that maximizes the availability. One naive solution is to repetitively redeploy the system actually in order to gain the desired availability. Another approach is to model the system with some parameters and assess the deployment architecture prior to the actual deployment. DeSi can be used for this purpose. It supports specification, manipulation, visualization, and assessment of deployment architectures for large-scale, highly distributed systems. DeSi provides a GUI in which it is possible to explore candidate deployments for a given system, specify the deployments that will

result in the highest improvements in availability, and assess the influence of changes in specific parameters and deployment constraints on the system and visualize them. In DeSi, a valid candidate deployment should satisfy the following four conditions: (1) the total required memories of the components deployed onto a host should not exceed the available memory of that host, (2) total volume of data exchanged over a link should not exceed its bandwidth, (3) a component can only be deployed onto a host that belongs to a set of allowed hosts for that component, and (4) two components should be deployed on the same or different hosts if it is specified to do so.

In DeSi, it is possible to integrate, evaluate, and compare different algorithms for improving systems availability in terms of their feasibility, efficiency and precision. Six of such algorithms are:

- *Exact Algorithm:* It tries every possible deployment, and selects the best one that has maximum availability and satisfies the constraints.

- *Unbiased Stochastic Algorithm:* It randomly assigns each component to a single host from the set of available hosts for that component and randomly generates a deployment. If this deployment satisfies all the constraints, its availability will be calculated. This process is done for a given number of times and the best deployment is selected.

- *Biased Stochastic Algorithm:* First, it randomly orders all the hosts and all the components. Then, for each host in this order, it assigns as many components as possible in such a way that all constraints are satisfied. This process continues until all components have been deployed.

- *Greedy Algorithm:* At each step of the algorithm, the best host for the best component is selected. The best host is one that has the highest sum of network reliabilities with other hosts in the system and the highest memory capacity. The best software component is the one that has the highest frequency of interaction with other components in the system, and the lowest required memory. This process continues with searching for the next best component until the best host is full. Then, again the best host is selected. This process proceeds until all components are assigned to hosts.

- *Clustering Algorithm:* In this algorithm, components with high frequencies of interaction are grouped into component clusters. Also, hosts

with high connection reliability are grouped into host clusters. Then, when a member of a component cluster requires to be redeployed to a new host, then all members of that cluster should be redeployed.

- *Decentralized Algorithm:* The above mentioned algorithms suppose the existence of a central host with reliable connections to every other host in the system. But, for some distributed systems as mobile networks which there does not exist such reliable connections, a decentralized algorithm is required. In this algorithm, each host acts as an agent and may conduct or participate in auctions (auction algorithm). In this algorithm, if none of the hosts connected to the desired host is already conducting an auction, the desired host initiates an auction by sending a message to all the neighboring hosts that carries information about a component needs to be redeployed. Each recipient host calculates an initial bid for the auctioned component by considering the frequency and value of interaction between components on its host and the auctioned component. The bidding agent sends this bid together with some local information such as its network reliability and its bandwidth with the neighboring hosts to the auctioneer. The auctioneer selects the highest bid as the winner. If the winner has enough resources to host the auctioned component, then the component is redeployed to it and the auction is closed. Otherwise, the winner and the auctioneer try to find another component on the winner host to be swapped with the auctioned component.

Another work in the area of QoS-aware deployment is done by D. Wichadakul and K. Nahrstedt. In [50], they talk about a translation system for enabling the deployment of QoS-aware applications that can be deployed into different ubiquitous environments with different middleware services. Ubiquitous systems are those that can be instantiated and accessed anytime, anywhere, and by using any computing devices. Examples of such systems are e-business audio/video streaming, and world-wide web.

Assuming the availability of QoS-oriented middleware services in different ubiquitous computing environments, they intend to answer the following question: "How to develop a QoS-aware application which can be deployed flexibly and efficiently in different environments, with different available middleware services, and satisfy acceptable quality of service?".

To answer this question, they introduce the concept of *middleware abstraction layer* or MAL. MAL represents a high-level functional view of

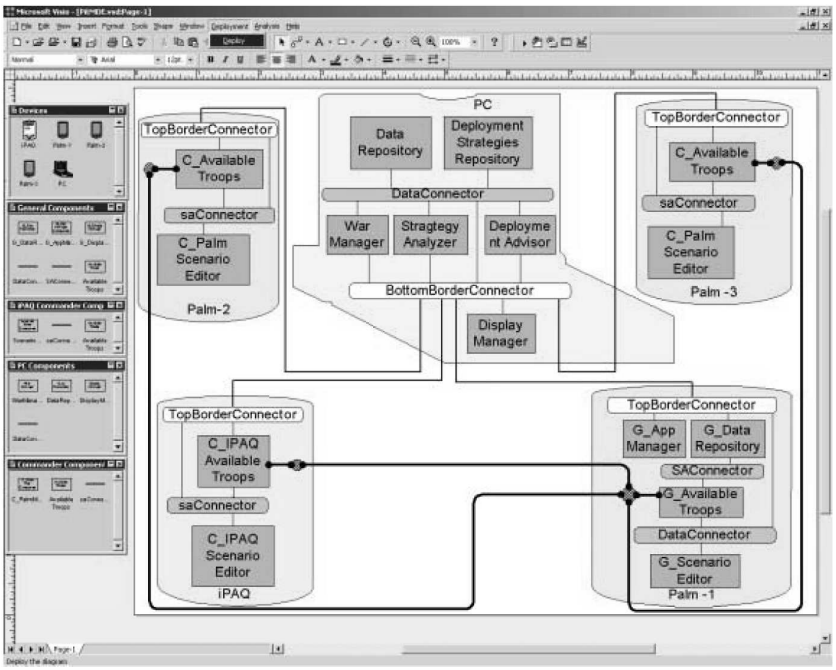

Figure 3.11: Prism deployment environment (Prism-DE) (taken from [51]))

the application and abstracts from individual middleware implementations. Then, the mapping (translation) between application services and a specific middleware services is done through the following steps. First, the QoS-aware application is mapped to configurations of generic middleware services without any respect to any specific implementations, resulting in the MAL representation of that application. In the second step, the MAL representation is customized with respect to a specific deployment environment.

3.2.4 Architecture Driven Deployment

In [51], software architecture is defined in terms of components, connectors, and configurations which provide high-level abstractions for showing the structure, behavior, and key properties of a software system. In this definition, components specify the computations and state of a system; connectors specify the mechanisms of communication among the components; configuration specify the topologies of components and connectors. In [51], M. M. Rakic and N. Medvidovic provide an approach for software deploy-

ment based on the principles of software architecture. They use the concepts of software architecture to (1) initially deploy a software system into a target environment, (2) update the components of an already deployed system, (3) analyze the likely effects of a deployment on a system before deploying it, and (4) analyze the effects of a deployment on a running system after its deployment. For this purpose, they introduce *Prism* architectural style which is suitable for heterogeneous, highly distributed, highly mobile, resource constrained systems. Prism also provides guidelines for software architects to design the architecture of systems. If a software system is designed based on these guidelines, then it is possible to use the Prism infrastructure developed by the authors of [51] to deploy that software system. More specifically, a GUI named Prism deployment environment (*Prism-DE*) has been implemented by them(Fig. 3.11). Users can use this GUI to design the architecture of an application based on the Prism architecture style guidelines. For this purpose, the architect is able to specify the configuration of hardware, devices (PCs, Laptops, etc.) by dragging their icons onto the convas and connecting them. Then, the architect provides software components and connectors that may be placed on top of the hardware devices. Once a designed software configuration is created and validated in Prism-DE, it can be deployed onto the depicted hardware configuration automatically.

3.2.5 Deployment into Computational Grids

A computational grid is defined as a set of efficient computing resources connected to the Internet and managed by a middleware that gives transparent access to resources wherever they are located on the network [52]. In such environments, the application deployment phase must be as automated as possible while taking into account application constraints (CPU, Memory, etc.) and/or user constraints, to prevent the user from directly dealing with a large number of hosts and their heterogeneity within a grid.

In [52], a framework for automatic deployment of component-based applications into computational grids in the context of the CORBA component model (CCM) is presented. However, it does not say anything about the necessary algorithms to develop such an automatic deployment tool. The authors of this paper mention that this architecture should have three entities: the inputs, the planner, and the actual deployment. In the following, these entities are described:

- The inputs:

    1. A description of the component-based application being deployed: In the context of the CCM, each application consists of a set of components (component assembly package). In this package, there exists an assembly description file which describes the components and their relationships. Also, each component in the CCM has a deployment descriptor that specifies how that component can be installed, configured, and launched on a machine.

       The component and application descriptors may express different requirements such as the CPU, Memory, Operating system, etc.
    2. A description of the grid resources on which the application may be deployed: This description includes information about computing resources, storage resources, and network description.

- The deployment planner uses the above inputs to generate a deployment plan in which:

    1. The computers that will run the components and the component servers are selected;
    2. The network links or network technology to interconnect the components are selected;
    3. The component servers are mapped onto the selected computers.

- Actually executing components on the computational grid

    1. In this step, the deployment plan is used to launch the component-based application and to configure it according to the CCM.

3.3 Software Dock Characterization Framework

In section 3.2.1, we talked about University of Colorado Software Dock research project. As part of this project, its research team did a survey of different software deployment technologies and provided a framework for characterizing them [30]. This framework has four factors for this characterization:

- *Process Coverage:* This factor specifies a given technology to what degree covers each of the constituent activities of the deployment process mentioned in the Software Dock research project. This factor can have each of these 3 values: no support for the given activity, minimal support, and full support.

- *Process Changeability:* sometimes it is impossible to anticipate all the requirements of the deployment process. As an example, it might be required to do some special tests at some points during the deployment process. This factor specifies whether or not it is possible to change the deployment process after its definition. For this purpose, the deployment process should be represented in an explicit and manipulable way.

- *Interprocess Coordination:* suppose you want to install a composite system which includes a number of subsystems. In this situation, first you need to deploy its subsystems. Thus, different deployment activities might be associated with different systems and so a coordination mechanism is required for cooperation and synchronization among these activities. In this framework, it is examined whether or not a technology supports synchronization and data exchange among different activities, and whether this support is extended over a wide-area network.

- *Site, Product, and Policy Abstraction:* Fig. 3.12 shows an example of deploying m products on n sites with three policies: confirm (dashed arrow) which requires a confirmation by the consumer before taking any action; notify (dotted arrow) which informs the consumer of every step taken; and automatic (solid arrow) which performs every action silently. In such a case, in the worst case, $m \times n \times 3$ specific deployment procedures for each deployment activity (e.g. Installation) is required. It means, in such a case, it is similar to having a deployment system that requires separate scripts for each activity, for every product, at every consumer site, and with every kind of execution policy. Obviously, the number of these scripts could become large. So, a good idea is considering to what extent different deployment technologies can provide abstract models for site, product, and policy information. For example, in Fig. 3.12, by moving from left to right in the figure and by

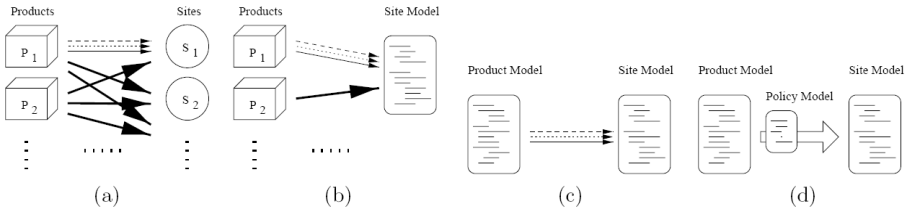

Figure 3.12: An Example of site, product, and policy factoring (taken from [30])

introducing abstract models for site, product, and policy, the number of specialized deployment procedures reduces.

A site model is an abstraction of a consumer site's resources and configuration. For example, the site model for a single computer can include information such as the machine type, the operating system, the available hardware resources, and the available software resources. This abstraction allows us to treat all different consumer sites in the same manner.

The product model is an abstraction of the constraints and dependencies of the system to be deployed in such a way that all deployable systems can be reasoned about in a consistent manner by a particular deployment procedure. This abstraction includes information as producer contact information, subsystem dependency specification, the set of constituent files, and documentation.

Both site model and product model should provide a standard schema together with some access and query functions so that deployment activities can use.

Policy model can contain information for describing scheduling, ordering, preferences, and security control. Unlike site and product models, it is impossible to abstract policy information into a generic schema. So, in [30], just the policy issues related to parameters for scheduling and control of resource usage are examined.

Deployment Technologies Examined. There are many different technologies both in academia and industry which support different parts of the deployment process. However, [30] classifies them into 5 major categories:

- *Installers:* They package a stand-alone software system into a self-installing archive. Then, This package can be distributed to its users.

Usually, they also provide some uninstallation features by undoing the changes they made during installation.

- *Package managers:* Many operating systems have some utilities to help system administrators to install, update, and manage software systems.
- *Application Management Systems:* Their original purpose was to support the management of corporate LANs in order to detect hardware and network failures and report them to administration centers. But, recently they have started to address some software management issues in medium- or large-scale organizations.
- *System Description Standards:* these standards specify various hardware and software properties. Thus, they could be used to specify both the site and product models.
- *Delivery of Content:* In this category, the information being deployed is transferred directly from one or more information servers to a number of client sites.

For each of these categories a number of samples are selected which are shown in Table 3.2 and Table 3.3. Table 3.2 shows the characterization of the technologies in terms of deployment process coverage. Table 3.3 shows their support for changeability, coordination, and model abstraction. In these tables, a filled circle (•) shows full support and an empty circle (◦) shows minimal support, and the absence of a circle shows no support at all.

		release		install		act	de-act	update		adapt	de-inst	de-rel
	System	pack	adv	trans	conf			trans	conf			
Installers	PC-Install 6	●			○						●	
	InstallShield 5	●			●						●	
	netDeploy 3	●		○	○	○		○	○		●	
Package managers	RPM	●		○	●			○	●	○	●	
	HP-UX SD	●			●				●	○	●	
	SUN *pkg*	●			●				●	○	●	
Application management	TME-10			●	●	●	●	●	●	○	●	
	SystemView			●	●	●	●	●	●	○	●	
	OpenView			●	●	●	●	●	●	○	●	
	Platinum			●	●	●	●	●	●	○	●	
	EDM				●	●	●		●		●	
Standards	MIF											
	AMS											
	Autoconf				○				○			
Content delivery	Castanet		○	●				●				
	PointCast		●	●								
	Rsync			●				●				
	Rdist			●								

Table 3.2: Software Dock's deployment process coverage in different technologies (taken from [30])

	System	changeability	coordination	model abstraction		
				site	product	policy
Installers	PC-Install 6				○	
	InstallShield 5	○			○	
	netDeploy 3				○	○
Package managers	RPM	○		○	●	○
	HP-UX SD	○		○	●	○
	SUN *pkg*	○		○	●	○
Application management	TME-10	●	○	●	●	
	SystemView	●	○	●	●	
	OpenView	●	○	●	●	
	Platinum	●	○	●	●	
	EDM	●	○	●	●	
Standards	MIF			●	○	
	AMS			●	●	
	Autoconf				○	
Content delivery	Castanet					
	PointCast					
	Rsync					
	Rdist					

Table 3.3: Changeability, coordination, and model abstraction in different technologies (taken from [30])

Chapter 4

Problem Description

This chapter provides a description of the deployment problem we intend to solve in this research. The focus of this research is on the planning activity of the software deployment process introduced in Chapter 1. In this research, we intend to develop the required algorithms for the deployment of channel-based component-based applications into distributed environments so that some QoS parameters (e.g., cost, reliability, etc.) are also optimized.

This chapter is structured as follows. Section 4.1 provides a simple example of modeling a flight reservation system with the Reo coordination model which is used as the running example throughout this chapter. Then, Section 4.2 provides a description of the software deployment process and its activities. Section 4.3 talks about the user-specified inputs that are used during the deployment planning. Then, in Section 4.4, these inputs are modeled with the help of graphs. Finally, in Section 4.5, a formal description of the deployment problem we intend to solve is provided.

4.1 An Example of Composing Web Services Using Reo

In the following, we provide a simple example of how a Reo connector such as barrier synchronization can be used to compose a number of Web services together. Web services refer to accessing services over the Web [55, 56]. In this example, they are treated as black-box software components.

Suppose a travel agency wants to offer a Flight Reservation Service (FRS). For some destinations, a connection flight might be required. Suppose some

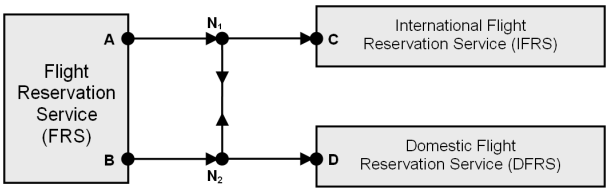

Figure 4.1: Modeling the flight reservation system with Reo

other agencies offer services for International Flight Reservation (IFRS) and Domestic Flight Reservation (DFRS). Thus, FRS commits successfully whenever both IFRS and DFRS services commit successfully. This behavior can be easily modeled by a barrier synchronization connector in Reo (Fig. 4.1). The FRS service makes commit requests on channel ends A and B. These commits will succeed if and only if the reservations at the IFRS and DFRS services succeed at the same time. This behavior is because of the semantic of the barrier synchronization connector in Reo. This example shows how Reo succeeds in modeling complex behaviors.

4.2 Deployment Process

Software deployment is a sequence of related activities for placing a developed application into its target environment and making the application available for use. Though this definition of software deployment is reasonable and clear, for developing an automated deployment planner, the characteristics and nature of the deployment activities must be described more clearly.

Different sequences of activities are mentioned in literature for the software deployment process. Some of them are mentioned in Section 3.2.1. However, in our view, the software deployment process should include at least the following activities: *Acquiring*, *Planning*, *Installation*, *Configuration*, and *Execution*. Below are brief descriptions of these activities:

1. *Acquiring:* In this activity, the components of the application being deployed and the metadata specifying the application are acquired from the software producer and are put in a repository to be used by other activities of the deployment process.

2. *Planning:* Given the specifications of the component-based application, a target environment, and user-defined constraints, this activity deter-

mines where different components of the application will be executed in the target environment, resulting in a deployment plan.

3. *Installation:* This activity uses the deployment plan generated in the previous activity to install the application into the target environment. More specifically, this activity transfers the components of the application from the repository to the hosts in the target environment.

4. *Configuration:* After installing the application components into the target environment, it might be necessary to modify its settings and configurations. For example, after installing an application, one may want to set different welcome messages for different users.

5. *Execution:* following the installation and configuration of the software application, it can be run. More specifically, the installed application components into the hosts are launched, the interconnections among them are instantiated, the components are connected to the interconnections, and the software application actually starts to work.

The focus of this project is on the planning activity of this process. In the following sections, we talk about this in more detail.

4.3 Deployment Planner Inputs

To generate deployment plans, the following inputs should be specified by the user: (1) the component-based application being deployed, (2) the distributed environment in which the application will be deployed, and (3) the user-defined constraints regarding this deployment. In the following, these inputs are described in more detail.

4.3.1 Specification of the Application Being Deployed

This input specifies the software application being deployed into the target environment. In the view of this project, a software application comprises a number of software components connected by a number of channels with different characteristics. The nature of these software components are irrelevant to this specification; they are treated as black box software entities that read

data from their input ports and write data to their output ports. How they manipulate the data or their internal details are not important. For example, they can be processes, Web services, Java beans, CORBA components, and so on.

In this project, the communications among these black box software entities is done via channels among them. However, these channels can have different characteristics and implementations. For example, in the case of using the Reo coordination middleware, channel types $T_1 - T_5$ could be defined as the following channel types (or implementations):

- T_1: *Sync* channel type implemented by shared memory;
- T_2: *Sync* channel type implemented by encrypted peer-to-peer connection;
- T_3: *Sync* channel type implemented by simple peer-to-peer connection;
- T_4: *SyncDrain* channel type;
- T_5: *SyncSpout* channel type.

Logically, $T_1 - T_3$ are all implementations of the same channel type (*Sync*). However, their hardware requirements and QoS characteristics differ.

Furthermore, it is possible to model the primitives of other communication models (such as message passing, shared spaces, or remote procedure calls) by the channel-based communication model [53]. Thus, other kinds of component-based applications can also be seen as some sorts of channel-based component-based applications.

In summary, the specification of the application should specify different components of the application and the channel types among them (e.g., Fig. 4.1).

4.3.2 Specification of the Target Environment

In this project, the target environment for the deployment of the application is a distributed environment consisting of a number of hosts with computational capabilities (e.g., PCs, laptops, servers, etc.) connected by a network. Furthermore, the required software for the communication among the application components has been already installed on them. However, since

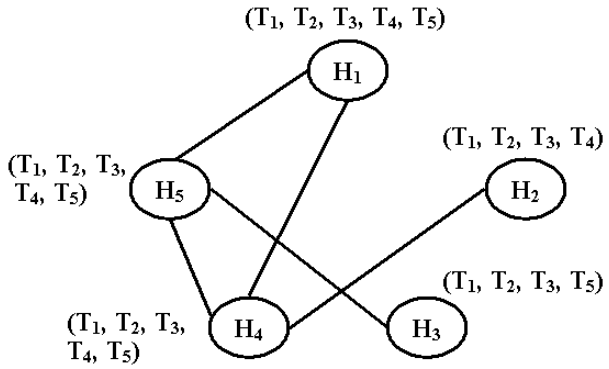

Figure 4.2: A sample target distributed environment for the deployment of the flight reservation system

different hosts may have different hardware properties, they may not be able to support some features of the communication software installed on them, or it may be even impossible to install some sorts of communication software on them. It is also possible that different features/versions of the communication software have been installed on different hosts intentionally because of some reasons (e.g., cost, security, etc.). With respect to this discussion, available hosts in the target environment might be able to support different sets of channel types (or implementations) with different behaviors and QoS characteristics. As an example, Fig. 4.2 shows a sample target environment for the flight reservation system consisting of five hosts $H_1 - H_5$, connected by a network (solid lines). In this figure, T_ds represent different channel types (or implementations) that different hosts can support.

In summary, the specification of the target environment specifies available hosts in the distributed environment for the deployment of the application, the topology of the physical network among them, and the channel types (or implementations) that each of them can support.

4.3.3 Specification of the User-defined Constraints

Users may have special requirements and constraints regarding the deployment of the application that should be taken into account during the deployment planning. For example, users may want a special component to be run on a certain host, or they may have certain QoS requirements such as security, cost, or reliability. The deployment planner needs this information

to generate a plan that answers these requirements too.

For example, in the flight reservation system, suppose users require the transfer of data between FRS and IFRS to be encrypted. In addition, they want FRS to be run on H_1, IFRS to be run on either H_2 or H_3.

4.4 Modeling the Deployment Planner Inputs

The deployment planner inputs should be modeled with well-defined structures in order to be used for effective deployment planning purposes. In this section, we show that it is easily possible to develop graph representations of these inputs. This graph-based modeling can have several advantages. First, it is possible to have visual representation of the inputs. Second, graph theory algorithms can help us in designing deployment planning algorithms. Third, it is possible to use graph theory symbols to formally represent deployment planner inputs and to prove the correctness of the designed deployment planning algorithms.

4.4.1 Modeling the Application Being Deployed

In section 4.3.1, we mentioned that channel-based component-based applications are viewed in this project as a number of components connected by a number of channels with different characteristics through which they communicate. With respect to this description of component-based applications, it is possible to model any channel-based component-based application as a graph whose nodes are application components and its edges are channels among these components.

Definition 4.4.1 ***(Application Graph)*** *Suppose C_is represent different components of the application, and T_ds represent different channel types. Then, application graph* $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$ *is defined as a graph on* $V_{\mathrm{AG}} = \{C_1, C_2, ..., C_n\}$ *in which each edge* $e \in E_{\mathrm{AG}}$ *has a label* $l_e \in \{T_1, T_2, ..., T_k\}$.

For example, Fig. 4.3 shows the application graph for the flight reservation system. This graph is built with respect to both the specifications of the application being deployed, and user-defined constraints regarding this deployment. For example, in the specification of the application (Fig. 4.1),

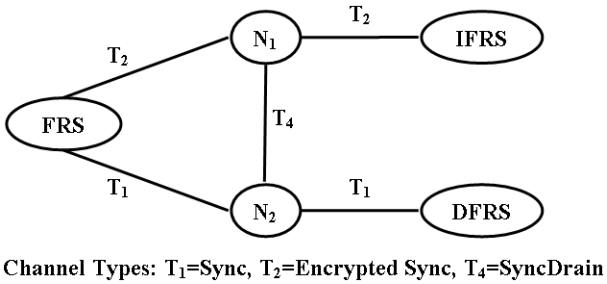

Figure 4.3: Application graph for the flight reservation system

Sync channels are used to connect FRS and IFRS components. But, as mentioned in section 4.3.3, users want the transfer of data between FRS and IFRS to be encrypted. Thus, in the application graph presented in Fig. 4.3, *Encrypted Sync* channel type is used between FRS and IFRS components.

4.4.2 Modeling the Target Environment

As mentioned in section 4.3.2, in this project the target environment for the deployment of the application is a number of hosts with different computational capabilities connected by a network in a distributed environment and each of them can support a set of channel types. With respect to this description of the target environment, it is possible to model the target environment with the help of a graph in which:

- Nodes represent available hosts in the distributed environment;
- Edges represent different channel types that can exist between every two hosts.

To generate such a graph, first it is required to notice to the following definitions.

Definition 4.4.2 ***(Physically Connected)*** *Two distinct hosts H_x and H_y are physically connected if there is a direct physical link between them in the distributed environment.*

As an example, hosts H_1 and H_4 in Fig. 4.2 are physically connected.

Definition 4.4.3 ***(Virtually Connected)*** *Two distinct hosts H_x and H_y are virtually connected if there is not any direct physical link between them in the distributed environment. But, they are connected indirectly through intermediate hosts.*

As an example, hosts H_1 and H_2 in Fig. 4.2 are virtually connected.

Definition 4.4.4 ***(Transitive Channel Type)*** *Suppose two hosts H_x and H_y are virtually connected. A channel type T_d is transitive if it is possible to create a channel of type T_d between them when (1) both of them can support channel type T_d, and (2) all intermediate hosts between them can also support channel type T_d.*

For example, in the Reo coordination model, channel type *Sync* is a transitive channel type.

Definition 4.4.5 ***(Non-transitive Channel Type)*** *A channel type T_d is non-transitive if it is possible to create a channel of type T_d between two hosts H_x and H_y only when (1) both of them can support channel type T_d, and (2) they are physically connected.*

As an example, in the Reo coordination model, channel type *SyncDrain* is a non-transitive channel type.

With respect to the above definitions, target environment graph is defined in the following way:

Definition 4.4.6 ***(Target Environment Graph)*** *Suppose H_is represent different hosts in the target environment, T_ds represent different channel types, and e_{H_x,H_y,T_d} represents an edge from node H_x to node H_y with label T_d. Then, the target environment graph $\mathrm{TG} = (V_{\mathrm{TG}}, E_{\mathrm{TG}})$ is defined as a graph on $V_{\mathrm{TG}} = \{H_1, H_2, ..., H_m\}$ in which the set of edges $E_{\mathrm{TG}} = \bigcup\{e_{H_x,H_y,T_d}\}$ is determined in the following way:*

- *If T_d is a transitive channel type, then there exists an edge e_{H_x,H_y,T_d} between two distinct nodes H_x and H_y only if (1) both of them are physically or virtually connected, (2) both of them support channel type T_d, and (3) if they are virtually connected, all intermediate hosts support channel type T_d.*

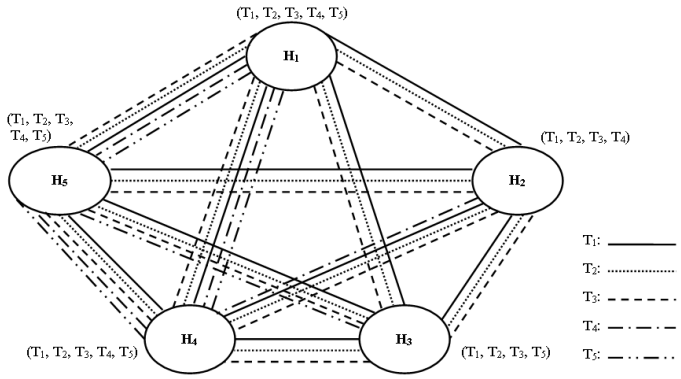

Figure 4.4: Target environment graph for the distributed environment presented in Fig. 4.2. $T_1 - T_3$ are transitive channel types. $T_4 - T_5$ are non-transitive channel types. For simplicity, loopback edges are not shown.

- *If T_d is a non-transitive channel type, then there exists an edge e_{H_x,H_y,T_d} between two distinct nodes H_x and H_y only if (1) they are physically connected, (2) both of them support channel type T_d.*
- *If T_d can be supported by host H_x, then there is an edge e_{H_x,H_x,T_d} from H_x to H_x (loopback edge).*

As an example, Fig. 4.4 shows the target environment graph generated by this method for the distributed environment presented in Fig. 4.2. To make the figure simpler, loopback edges are not shown. For a more specific example, consider hosts H_1 and H_2 which are virtually connected (i.e., through host H_4). As mentioned in section 4.3.1, in this example, $T_1 - T_3$ are different implementations of the *Sync* channel type which is a transitive channel type. Thus, it is possible to have channels of types $T_1 - T_3$ between H_1 and H_2. Furthermore, both H_1 and H_2 support channel type T_4 (i.e., *SyncDrain*) which is a non-transitive channel type. However, since H_1 and H_2 are not physically connected, it is impossible to have a channel of type T_4 between them.

4.5 Definition of the Deployment Problem

After specifying the deployment planner inputs and modeling them with the help of graphs, they can be used to generate the actual deployment plan.

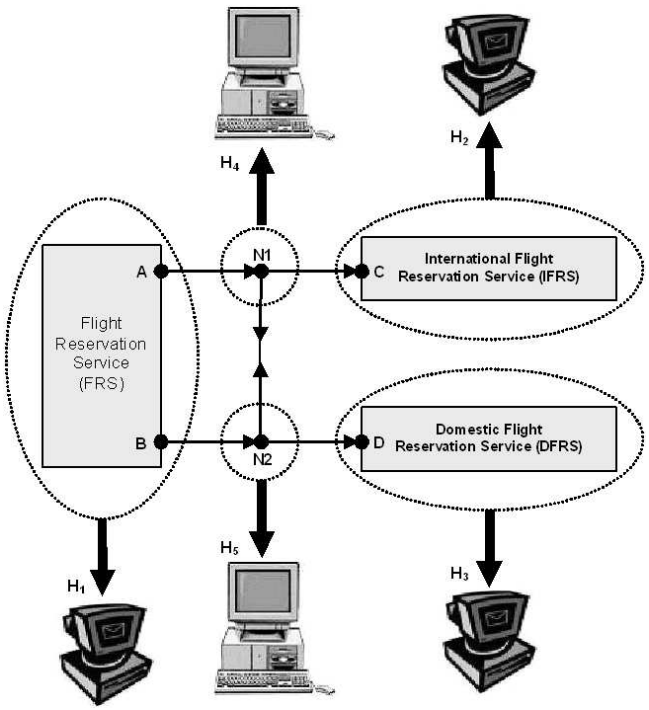

Figure 4.5: A sample deployment for the flight reservation system

This deployment plan determines where different components of the application will be executed in the target environment so that all requirements and constraints are met. The deployment planning approach of this project is based on the communication resources different application components require, and different communication resources available on different hosts in the target environment.

Fig. 4.5 shows one sample deployment for the flight reservation system. As can be seen in this figure, different components of the application and channels among them are mapped to different hosts in the target environment and network links among them for the purpose of this deployment. Actually, you may notice that in this deployment, different nodes and edges of the application graph AG shown in Fig. 4.3 are mapped to different nodes and edges of the target environment graph TG presented in Fig. 4.4. This mapping is shown in Fig. 4.6. In this way, it is possible to see the deployment planning as a graph mapping problem from the application graph to the target environment graph. In this section, we intend to formally define

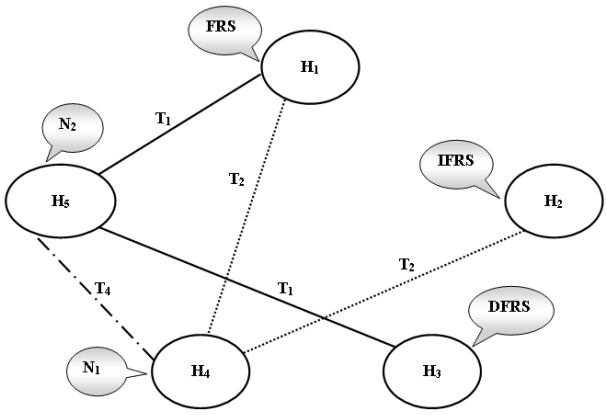

Figure 4.6: Mapping application graph (Fig. 4.3) to the target environment graph (Fig. 4.4) to generate a deployment plan for the flight reservation system

Component Name	Candidate Hosts
FRS	H_1
IFRS	H_2, H_3
DFRS	H_1, H_2, H_3, H_4, H_5
N_1	H_1, H_2, H_4, H_5
N_2	H_1, H_2, H_4, H_5

Table 4.1: Candidate hosts for the deployment of the flight reservation system components

this graph mapping problem. However, before everything, we begin with defining some general terms which are used in the rest of this report.

Definition 4.5.1 ***(Candidate Host)*** *Let* $T_{C_i} = \{T_d | T_d \in T, \exists \{C_i, C_j\} \in E_{\mathrm{AG}} : l_{\{C_i,C_j\}} = T_d\}$ *represent all required channel types by component* C_i *in the application graph* $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$ *and let* $T_{H_x} = support(H_x)$ *represent the set of channel types that host* H_x *can support. Then, host* H_x *is a candidate host for the deployment of component* C_i*, only if (1)* $T_{C_i} \subseteq T_{H_x}$*, and (2) host* H_x *satisfies user-defined constraints regarding the deployment of component* C_i*.*

This definition implies that a host H_x is a candidate host for the deployment of component C_i if it supports all required channel types by component

C_i in the application graph and also the deployment of component C_i on host H_x meets user-defined constraints. As an example, Table 4.1 shows the candidate hosts for the deployment of the flight reservation system components. For a more specific example, consider component IFRS. In the application graph presented in Fig. 4.3, IFRS just requires channel type T_2 and all of the hosts in the target environment presented in Fig. 4.2 support this channel type. But, as mentioned in section 4.3.3, users want IFRS to be deployed on either hosts H_2 or H_3. So, with respect to this constraint, candidate hosts for the deployment of component IFRS are H_2 and H_3.

Definition 4.5.2 ***(Candidate Deployment)*** *Suppose* CH_{C_i} *represents the set of candidate hosts for the deployment of component* C_i*. Then, a candidate deployment* D_c *is a set of pairs* (C_i, H_x) *in which every component* C_i *in the application graph* $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$ *is mapped to a host* H_x *in the target environment graph* $\mathrm{TG} = (V_{\mathrm{TG}}, E_{\mathrm{TG}})$ *so that host* H_x *is a candidate host for the deployment of component* C_i*, i.e.,* $D_c = \{(C_i, H_x) | C_i \in V_{\mathrm{AG}}, H_x \in V_{\mathrm{TG}}, H_x \in \mathrm{CH}_{C_i}\}$.

For example, $\{(\mathrm{FRS} \mapsto H_1), (\mathrm{IFRS} \mapsto H_2), (\mathrm{DFRS} \mapsto H_3), (\mathrm{N}_1 \mapsto H_4), (\mathrm{N}_2 \mapsto H_5)\}$ and $\{(\mathrm{FRS} \mapsto H_1), (\mathrm{IFRS} \mapsto H_3), (\mathrm{DFRS} \mapsto H_3), (\mathrm{N}_1 \mapsto H_4), (\mathrm{N}_2 \mapsto H_5)\}$ are two candidate deployments for the flight reservation system.

Definition 4.5.3 ***(Valid Deployment)*** *A candidate deployment* D_c *is a valid deployment, if for all edges* e_{C_i,C_j,T_d} *in the application graph* $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$ *if components* C_i *and* C_j *are mapped to two not necessarily distinct hosts* H_x *and* H_y *in the target environment, then it should be possible to create a channel of type* T_d *between hosts* H_x *and* H_y*, i.e., there should be an edge* e_{H_x,H_y,T_d} *in the target environment graph* $\mathrm{TG} = (V_{\mathrm{TG}}, E_{\mathrm{TG}})$*. Formally speaking,* $\forall e_{C_i,C_j,T_d} \in E_{\mathrm{AG}} \Rightarrow \exists e_{D_c(C_i),D_c(C_j),T_d} \in E_{\mathrm{TG}}$.

As an example, $D_c = \{(\mathrm{FRS} \mapsto H_1), (\mathrm{IFRS} \mapsto H_2), (\mathrm{DFRS} \mapsto H_1), (\mathrm{N}_1 \mapsto H_1), (\mathrm{N}_2 \mapsto H_2)\}$ is an invalid deployment for the flight reservation system, because there is an edge e_{N_1,N_2,T_4} in the application graph presented in Fig. 4.3. But, there is not an edge $e_{D_c(N_1),D_c(N_2),T_4} = e_{H_1,H_2,T_4}$ in the target environment graph presented in Fig. 4.4. In other words, with respect to the specification of the target environment presented in Fig. 4.2, it is impossible to create a channel of type T_4 between hosts H_1 and H_2.

With respect to above definitions, it is typically possible to deploy a complex component-based application into a large distributed environment

```
for each component C_i in the application do
    Find set of candidate hosts, CH_{C_i};
    if CH_{C_i} == null then
        return "No Answer!";
    end
end
Generate all permutations of CH_{C_i}s and set them as candidate
deployments;
initialize Best_QoS;
for each candidate deployment D_c do
    if D_c is a valid deployment then
        QoS_{D_c} = Compute_QoS(D_c);
        if QoS_{D_c} > Best_QoS then
            Best_QoS = QoS_{D_c};
        end
    end
end
return Best_QoS
```

Algorithm 1: Exhaustive algorithm for the deployment planning

in many different ways. As an example, consider again the candidate hosts for deploying each of the components of the flight reservation system shown in Table 4.1. As can be understood from this table, it is possible to deploy this application into the target environment in at most $160 = 1 \times 2 \times 5 \times 4 \times 4$ different ways (because some of them are invalid deployments). Obviously, this number is much bigger for complex applications deployments. However, when some QoS parameters, such as cost, performance, reliability, etc., are considered, some of these candidate deployments are equivalent, some are better than others and only a few of them may accommodate the constraints and requirements of the application. Thus, when QoS of the application is important, it should be tried to deploy the application so that its desired QoS parameter is optimized.

The pseudocode presented in Algorithm 1 shows one naive solution to the deployment problem when QoS of the application should be noticed. In this pseudocode, first the sets of candidate hosts for deploying different components of the application are found. Then, all candidate deployments are

generated by permuting the sets of candidate hosts for different components of the application. Then, the desired QoS parameter of all valid candidate deployments is measured and the best one is selected. The complexity of this algorithm is $O(mn + m^n) = O(m^n)$, where m is the number of available hosts in the target environment and n is the number of components of the application. As we see, this is an exponentially complex solution to the deployment problem. Thus, when the number of candidate deployments is large, it is very difficult to generate all of them and then select the best one. So, a set of algorithms and heuristics should be designed and applied to effectively solve such an exponentially complex problem. The following definition, provides a formal definition of the deployment problem we intend to solve.

Definition 4.5.4 ***(Deployment Problem)*** *Suppose deployment planner inputs are used to build the application graph and the target environment graph according to the methods presented in section 4.4.* CH_{C_i} *also represents the set of candidate hosts for the deployment of component* C_i*. Then, for the given application graph* $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$*, target environment graph* $\mathrm{TG} = (V_{\mathrm{TG}}, E_{\mathrm{TG}})$*, and QoS parameter* Q*, the problem is to find an efficient mapping function* $D : V_{\mathrm{AG}} \rightarrow V_{\mathrm{TG}}$ *such that the application's* Q *parameter is optimized, and the following two conditions are also satisfied:*

1. $D(C_i) = H_x \Rightarrow H_x \in \mathrm{CH}(C_i)$*. This means that all components of the application must be mapped to one of their respective candidate hosts for the deployment;*

2. $\forall e_{C_i,C_j,T_d} \in E_{\mathrm{AG}} \Rightarrow \exists e_{D(C_i),D(C_j),T_d} \in E_{\mathrm{TG}}$*. This means that the deployment* D *must be a valid deployment.*

This definition implies that during the deployment, it is possible to map several application components to a single host if that host is a candidate host for the deployment of those components. Furthermore, if there exists a channel of type T_d between two components in the application graph, then those components can be mapped to two different hosts only if there exists a channel of type T_d between them in the target environment graph.

As an example of how such efficient algorithms and techniques can be applied to effectively solve the deployment problem, in the following chapter, the deployment problem is solved for the QoS parameters cost and reliability, when the target environment is a peer-to-peer (P2P) distributed environment.

Chapter 5

Deployment Algorithm for P2P Target Environments

In this chapter, we intend to solve the deployment problem defined in chapter 4 for QoS parameters cost and reliability when the target environment is a peer-to-peer (P2P) distributed environment (e.g., Internet). In a P2P distributed environment, two or more computers (called nodes) can directly communicate with each other, without the need for any intermediary devices [57]. In other words, it is assumed in this architecture that every two computers in the network are directly connected. In particular, in contrast to the client/server architecture, in a P2P architecture, nodes have equivalent responsibilities, enabling applications that focus on collaboration and communication [56]. In this situation, it is not required to consider the issues related to the physical connectivity among hosts, i.e., transitive property of channel types. In other words, the physical topology of the network is not important for us. As an example, Fig. 5.1 shows a sample P2P distributed environment consisting of four hosts $H_1 - H_4$, each of them can support different subsets of channel types $T_1 - T_7$. In this case, the definition of the target environment graph becomes much simpler than that provided in Chapter 4.

Definition 5.0.5 ***(P2P Target Environment Graph)*** *The target environment graph* $\mathrm{TG} = (V_{\mathrm{TG}}, E_{\mathrm{TG}})$ *for a P2P distributed environment is a graph on* $V_{\mathrm{TG}} = \{H_1, H_2, ..., H_m\}$ *in which there exists an edge* e_{H_x,H_y,T_d} *between two not necessarily distinct nodes* H_x *and* H_y *if and only if both of them can support channel type* T_d*. Formally speaking:*

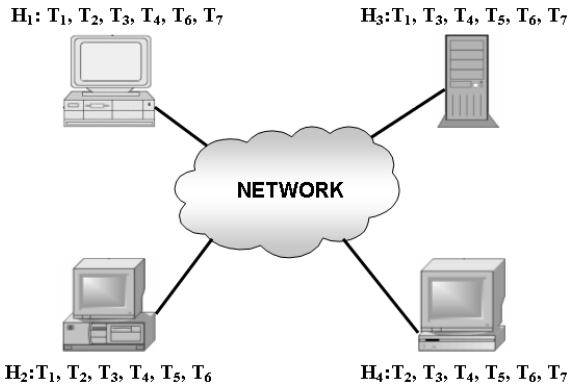

Figure 5.1: A sample distributed environment consisting of fours hosts $H_1 - H_4$ supporting different channel types $T_1 - T_7$.

$$
\begin{array}{lcl}
E_{\mathrm{TG}} & = & \bigcup\{e_{H_x,H_y,T_d}\} \ \textit{where} \\
 & & H_x, H_y \in V_{\mathrm{TG}} \wedge \\
 & & T_d \in \mathit{support}(H_x) \cap \mathit{support}(H_y) \\
T & = & \{T_1, T_2, ..., T_k\} \\
\mathit{support} & : & V_{\mathrm{TG}} \longrightarrow 2^T \\
\mathit{support}(H_x) & = & \textit{The set of channel types} \\
 & & \textit{that host } H_x \textit{ can support;} \\
e_{H_x,H_y,T_d} & = & \textit{An edge from node } H_x \textit{ to} \\
 & & \textit{node } H_y \textit{ with label } T_d\textit{;}
\end{array}
$$

5.1 Minimum Cost Deployment

Suppose different hosts in the target environment have different costs and whenever they are being used, their costs should be paid to their administrator(s). In this situation, one QoS parameter of a deployment is its cost and should be minimized in the deployment plan. For this, two different cases can be considered:

Case 1: The cost should be paid for each component. In this case, for every component to be run on each host, its cost should be paid separately. For example, for each component to be run on host H_1, \$1000 should be paid to its administrator(s). Thus, if five components to be run on host H_1, $5 \times \$1000 = \$5000$ should be paid. The required algorithm of this case is

```
for each component C_i in the application do
    Find the set of candidate hosts, CH_{C_i};
    if CH_{C_i} == null then
        return "No Answer!";
    end
    else
        H_x = cheapest host in the set CH_{C_i};
        Output: C_i ↦ H_x
    end
end
```

Algorithm 2: Minimum cost deployment algorithm when the cost should be paid for each component

very simple. In this case, in the set of candidate hosts for the deployment of each of the application components, the cheapest one is selected and that component is deployed on it. The pseudocode of this algorithm is shown in Algorithm 2. This algorithm has the polynomial complexity $O(mn)$.
Case 2: The cost should be paid for each host, no matter how many components will be run on it. In this case, the number of components will be run on each host is not important; if the cost of one host is paid, it is possible to run as many components as you want on it. The complexity of this case is much more than the previous one. In this case, it should be tried to select a subset of available hosts in the target environment so that the total cost of the deployment is minimized and all the components of the application are also assigned to a host. This case of the minimum cost deployment problem is defined formally in the following definition.

Definition 5.1.1 ***(Minimum Cost Deployment Problem)*** *Suppose the following inputs are given:*

1. *A finite set V_{AG} of n components, $V_{\text{AG}} = \{C_1, C_2, ..., C_n\}$;*

2. *A collection of subsets of V_{AG}, $S = \{CS_{H_1}, CS_{H_2}, ..., CS_{H_m}\}$, in which each CS_{H_x} corresponds to host H_x, and it represents the subset of application components that can be run on host H_x. Also, every component C_i belongs to at least one CS_{H_x};*

$X = \emptyset, \tau = \emptyset$;
while $X \neq V_{\text{AG}}$ **do**
  Find the set $\omega \in S$ that minimizes $c(\omega)/|\omega \backslash X|$;
  $X = X \cup \omega, \tau = \tau \cup \{\omega\}$;
end
Output: τ

Algorithm 3: Greedy approximation algorithm for the minimum set cover problem

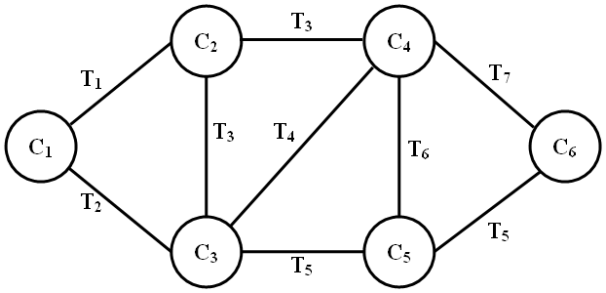

Figure 5.2: A sample application graph. C_is represent application components. T_ds represent different channel types among them.

3. *A cost function* $c : S \longrightarrow R$ *so that* $c(CS_{H_x}) = c'(H_x)$. *Function* $c' : H \longrightarrow R$ *returns the cost of each host.*

The minimum cost deployment problem is to find a minimum cost subset of S *(a subset of available hosts) that covers all components of* V_{AG}.

This problem is exactly one of the existing problems in graph theory that is called *Minimum Set Cover* problem [58]. However, it is proved that minimum set cover problem is a NP-hard problem and it can not be solved in polynomial time [58]. But, there exist some greedy approximation algorithms that can find reasonably good answers in polynomial time. One of the key algorithms for solving this problem is provided in Algorithm 3. The main idea in this algorithm is to iteratively select the most minimum cost $s_i \in S$ and remove the covered elements until all elements are covered. The complexity of this algorithm is $O(log(|V_{\text{AG}}|))$.

As an example of using this greedy approximation algorithm, suppose that we want to find the minimum cost deployment of the application whose

Component Name	Candidate Hosts
C_1	H_1, H_2
C_2	H_1, H_2, H_3
C_3	H_2, H_4
C_4	H_1, H_3, H_4
C_5	H_2
C_6	H_3, H_4

Table 5.1: Candidate hosts for the deployment of the application components presented in Fig. 5.2 into the target environment presented in Fig. 5.1.

graph is presented in Fig. 5.2 into the target environment shown in Fig. 5.1. Also, suppose that users want the component C_5 to be run on either H_1 or H_2. Table 5.1 shows the candidate hosts for deploying the components of this application graph into the target environment presented in Fig. 5.1. With respect to Table 5.1, the elements of the minimum cost deployment problem are defined in the following way:

- $V_{\text{AG}} = \{C_1, C_2, C_3, C_4, C_5, C_6\}$;
- $S = \{\{C_1, C_2, C_4\}, \{C_1, C_2, C_3, C_5\}, \{C_2, C_4, C_6\}, \{C_3, C_4, C_6\}\}$;
- $c'(H_1) = \$1000$, $c'(H_2) = \$2500$, $c'(H_3) = \$2000$, $c'(H_4) = \$1500$.

By applying the greedy approximation algorithm, we will have the following results and the minimum cost will be $c'(H_2) + c'(H_4) = \$2500 + \$1500 = \$4000$:

- $\{(C_1 \mapsto H_2), (C_2 \mapsto H_2), (C_3 \mapsto H_2), (C_4 \mapsto H_4), (C_5 \mapsto H_2), (C_6 \mapsto H_4)\}$;
- $\{(C_1 \mapsto H_2), (C_2 \mapsto H_2), (C_3 \mapsto H_4), (C_4 \mapsto H_4), (C_5 \mapsto H_2), (C_6 \mapsto H_4)\}$.

Note that it is possible to use the algorithm presented here more generally for some other QoS parameters too, when you want to minimize the total usage of some resources of available hosts in the target environment. In this situation, it is possible to define the cost function c to return the amount of that resource for each host and then use the greedy approximation algorithm presented in Algorithm 3 to find the solution.

5.2 Reliable Deployment

In this section, our focus is on maximizing the reliability of the software application, defined as the probability of failure-free software operation for a specified period of time in a specified environment [59]. In the context of distributed environments, one potential problem is network failures. In these environments, connectivity losses can lead to disastrous effects on the system's reliability, and the software application may not provide its desired functionality. To reduce the risks of this problem, one solution is to make the communications among the application components as local as possible. In this way, components located in the same host can communicate without any respect to the network's status. Thus, we define the most reliable deployment configuration as one with the least amount of communications among the hosts in the distributed environment. From another point of view, this can be seen as the increased performance. This is because of the fact that in distributed environments, network communications have some overheads on the software application. Thus, reduced communication among hosts can result in a higher performance.

5.2.1 Use of Multiway Cut Problem in Reliable Deployment Planning

In this section, we show that the reliable deployment problem corresponds to the multiway cut problem in graph theory [63].

Definition 5.2.1 ***(Multiway Cut Problem)*** *Let $G = (V, E)$ be an undirected graph on $V = \{v_1, v_2, ..., v_n\}$ in which each edge $e \in E$ has a nonnegative weight $w(e)$, and let $T = \{t_1, t_2, ..., t_m\} \subseteq V$ be a set of terminals. Multiway cut is the problem of finding a set of edges $E' \subseteq E$ such that the removal of E' from E disconnects each terminal from all other terminals, and solution cost* $\mathrm{MC} = \sum_{e \in E'} w(e)$ *is also minimized.*

Suppose $\mathrm{AG} = (V_{\mathrm{AG}}, E_{\mathrm{AG}})$ is the application graph of the software application being deployed, $V_{\mathrm{TG}} = \{H_1, H_2, ..., H_m\}$ represents the set of available hosts in the target environment, and CH_{C_i} represents the set of candidate hosts for the deployment of component C_i. To solve the reliable deployment problem, a graph $G = (V, E)$ is made in the following way:

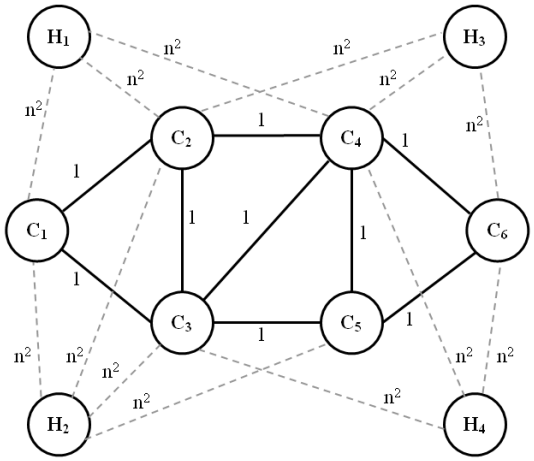

Figure 5.3: A graph built for finding the most reliable deployment configuration of the application presented in Fig. 5.2 into the target environment shown in Fig. 5.1.

- $V = V_{\text{AG}} \cup V_{\text{TG}}$. This means that the components of the application and hosts of the target environment are set as the nodes of graph $G = (V, E)$.

- $E = E_{\text{AG}} \cup E_H$, where $E_H = \{\{C_i, H_x\} | C_i \in V_{\text{AG}}, H_x \in \text{CH}_{C_i}\}$. This means that E includes both the edges of the application graph, and edges that connect each component to its respective candidate hosts for the deployment.

- $w(e) = \begin{cases} 1 & e \in E_{\text{AG}} \\ n^2 & e \in E_H \end{cases}$. Here n^2 shows a large number. This means that the weight of the application graph edges are set as 1 and the weight of the edges connecting application components to their candidate hosts for the deployment are set as n^2.

Fig. 5.3 shows an example of a graph developed in this way for the application graph presented in Fig. 5.2, and the target environment presented in Fig. 5.1. In this graph, if we set hosts as the terminals of the multiway cut problem, we prove in the following theorem that the solution of the multiway cut problem is the solution of the reliable deployment problem we intend to solve.

Theorem 5.2.1 *Suppose graph $G = (V, E)$ is built in the way mentioned earlier, and hosts of the target environment are set as the terminals. Then, the multiway cut solution of this graph is the solution of the reliable deployment problem we are looking for. This means that the application components that lie in the same subgraph with a host should be deployed on that host, and this deployment configuration has the least number of channels among hosts.*

Proof Suppose n represents the number of components of the application, and k represents the size of the E_H, i.e., $k = |E_H|$. In the multiway cut solution we are looking for, each component must be assigned to exactly one host. Thus, $(k-n)$ edges whose total weight is $n^2(k-n)$ will be removed from the E_H in the cut. Also, suppose L_{OPT} represents the solution of the reliable deployment problem, i.e., the least number of channels among hosts after the deployment of the application. Actually, these channels are those application graph edges that lie in the cut, and their total weight is $L_{\mathrm{OPT}} \times 1 = L_{\mathrm{OPT}}$. Thus, our goal is to prove that $\mathrm{MC} = n^2(k-n) + L_{\mathrm{OPT}}$. For this purpose, we should prove that $\mathrm{MC} \leq n^2(k-n) + L_{\mathrm{OPT}}$ and $\mathrm{MC} \geq n^2(k-n) + L_{\mathrm{OPT}}$.

Case A: $\mathbf{MC \leq n^2(k-n) + L_{OPT}}$.

Suppose a deployment $D : V_{\mathrm{AG}} \longrightarrow V_{\mathrm{TG}}$ whose cost is optimum is done, i.e., it has L_{OPT} number of channels among hosts. Now, assume that $\mathcal{C}$ is its corresponding cut in the graph $G = (V, E)$:

$$\mathcal{C} = \underbrace{\{\{C_i, H_x\} | D(C_i) \neq H_x, \{C_i, H_x\} \in E_H\}}_{\mathcal{M}} \bigcup \underbrace{\{\{C_i, C_j\} | D(C_i) \neq D(C_j), \{C_i, C_j\} \in E_{\mathrm{AG}}\}}_{\mathcal{N}}$$

$\mathcal{M}$ represents the set of edges of E_H that lie in the cut, and $\mathcal{N}$ represents the set of edges of E_{AG} that lie in the cut. The size of $\mathcal{M}$ is $(k-n)$ and the size of $\mathcal{N}$ is L_{OPT}. Furthermore, the weight of the edges in $\mathcal{M}$ is n^2 and the weight of the edges in $\mathcal{N}$ is 1. With respect to this description:

$$\begin{aligned} w(\mathcal{C}) &= w(\{\{C_i, H_x\} | D(C_i) \neq H_x, \{C_i, H_x\} \in E_H\}) + \\ &\quad w(\{\{C_i, C_j\} | D(C_i) \neq D(C_j), \{C_i, C_j\} \in E_{\mathrm{AG}}\}) \\ &= n^2 \times |\{\{C_i, H_x\} | D(C_i) \neq H_x, \{C_i, H_x\} \in E_H\}| + \\ &\quad 1 \times |\{\{C_i, C_j\} | D(C_i) \neq D(C_j), \{C_i, C_j\} \in E_{\mathrm{AG}}\}| \\ &= n^2(k-n) + L_{\mathrm{OPT}} \end{aligned}$$

Since MC is the cost of the optimum multiway cut, for sure, MC $\leq$ $w(\mathcal{C})$. Therefore, MC $\leq n^2(k-n) + L_{\text{OPT}}$.

Case B: MC $\geq \mathbf{n^2(k-n) + L_{OPT}}$.

Suppose $\mathcal{C}$ is the optimum multiway cut for graph $G = (V, E)$ whose cost is MC. Now, we want to use this cut to generate its corresponding deployment D. For this purpose, we prove the following subcases:

Subcase B.1: Cut $\mathcal{C}$ includes at most $(k-n)$ edges of E_H.

Suppose we want to find a cut whose cost is the heaviest. In the deployment configuration we are looking for, each component should be assigned to exactly one host. For this purpose, for each component C_i in graph G, we keep an arbitrary edge connecting that component to an arbitrary host, and we cut the rest of the edges in E_H and E_{AG}. Since the maximum number of edges in the application graph is $\binom{n}{2}$, the cost of this cut is at most $\binom{n}{2} + n^2(k-n)$. Thus, the cost of the multiway cut $\mathcal{C}$ can not be more than $\frac{n^2}{2} + n^2(k-n)$. This means that the cut $\mathcal{C}$ includes at most $(k-n)$ edges of E_H. Because, for example, if it includes $(k-n+1)$ edges of E_H, then the cost of the cut would be $\binom{n}{2} + n^2(k-n+1)$ which is more than the maximum cost we found here.

Subcase B.2: Each component C_i is connected to at most one host in the cut $\mathcal{C}$.

Suppose a component C_i is connected to two different hosts H_x and H_y in the cut. This means that H_x and H_y are connected together in the cut. However, since H_x and H_y belong to the set of terminals, this is impossible. Therefore, C_i is connected to at most one host in the cut.

Subcase B.3: Each component C_i is connected to exactly one host in the cut $\mathcal{C}$.

From subcases B.1 and B.2 together, it can be easily understood that each component C_i is connected to exactly one host in the cut $\mathcal{C}$. $D(C_i)$ represents the host on which component C_i is mapped.

By using the subcase B.3, cut $\mathcal{C}$'s corresponding deployment configuration D can be made. Suppose $L_D = |\{\{C_i, C_j\} | D(C_i) \neq D(C_j), \{C_i, C_j\} \in E_{\text{AG}}\}|$ represents the cost of the deployment configuration D, i.e., the

1. For each terminal $t_i \in T$, find a minimum-cost set of edges $\mathcal{C}_{t_i}$ whose removal disconnects t_i from the rest of the terminals;
2. Discard cut $\mathcal{C}_{t_x}$ whose cost $w(\mathcal{C}_{t_x})$ is the heaviest;
3. Output the union of the rest, call it $\mathcal{C}$.

Algorithm 4: Approximation algorithm for solving the multiway cut problem.

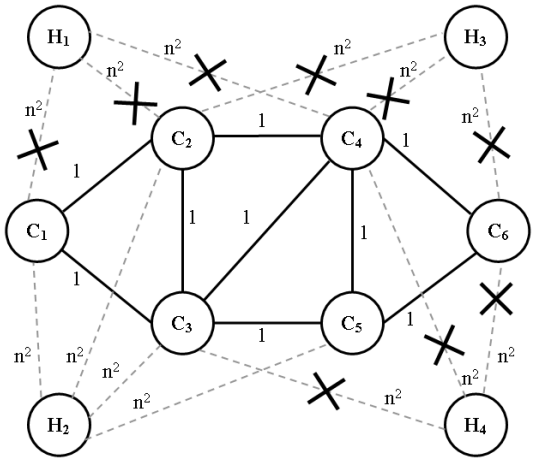

Figure 5.4: An approximation for the multiway cut of the graph presented in Fig. 5.3.

number of channels among the hosts in the deployment configuration D. In the following, we prove the correctness of case B:

$$\begin{aligned} \text{MC} &= n^2(k-n) + |\{\{C_i, C_j\} | \{C_i, C_j\} \in \mathcal{C}, \{C_i, C_j\} \in E_{\text{AG}}\}| \\ &\geq n^2(k-n) + |\{\{C_i, C_j\} | D(C_i) \neq D(C_j), \{C_i, C_j\} \in E_{\text{AG}}\}| \\ &= n^2(k-n) + L_D \Longrightarrow \\ \text{MC} &\geq n^2(k-n) + L_D \geq n^2(k-n) + L_{\text{OPT}} \end{aligned}$$

Cases A and B together imply that $\text{MC} = n^2(k-n) + L_{\text{OPT}}$. Therefore, the correctness of theorem 5.2.1 is proved. ∎

In theorem 5.2.1, we showed that the solution of the reliable deployment problem can be found by solving the multiway cut problem in graph theory. However, it is proved that the multiway cut problem is an NP-hard problem

when the number of terminals is greater than two. Thus, unless P=NP, it does not have a polynomial time solution [63]. However, it is possible to find many approximation algorithms for the multiway cut problem in literature [63, 64, 65]. One of the well-known and very simple approximation algorithms developed by Dalhaus et al. is provided in Algorithm 4 [63]. This algorithm finds a minimum cost cut for each terminal t_i separating it from the remaining terminals. Then, outputs the union of the $m-1$ cheapest of the m cuts. As an example, Fig. 5.4 shows an example of applying this algorithm on the graph presented in Fig. 5.3. As we see in this figure, one of the main problems of these approximation algorithms is that some components may not be assigned to any hosts (e.g., C_4 and C_6). To solve this problem, after applying the multiway cut approximation algorithm on the graph, we check whether or not all components are assigned to a host. If there are some components which are not assigned to any hosts, we connect those components to one of their candidate hosts for the deployment, and we cut all the application graph edges that are connected to those components. This approach not only will solve the problem, but also will improve the approximation of the multiway cut. Because, we are actually removing from the multiway cut approximation some heavy edges that connect the components to the hosts. Thus, the cost of the multiway cut is being reduced. Consequently, the result is closer to the optimum solution we are looking for. After applying this improvement on the multiway cut approximation presented in Fig. 5.4, one possible solution for the reliable deployment problem is $\{(C_1 \mapsto H_2), (C_2 \mapsto H_2), (C_3 \mapsto H_2), (C_4 \mapsto H_4), (C_5 \mapsto H_2), (C_6 \mapsto H_4)\}$.

Chapter 6

Prototype Implementation

At the time of writing this report, we have implemented a deployment planner tool by Java using the algorithms provided in Chapter 5. The inputs of this tool are "Application.inf" and "Target.inf" files. In "Application.inf", the components of the application being deployed and the channel types among them are specified. In other words, the topology of the application to be deployed is specified in this file. In the "Target.inf" file, available hosts in the target environment and their properties are specified. These properties include different channel types they can support, their costs, their IPs, and so on.

The data structure used to hold the information about the application and the target environment is *Linked-List*. In this structure, the topology of the application is kept as a linked-list of components. Each component itself

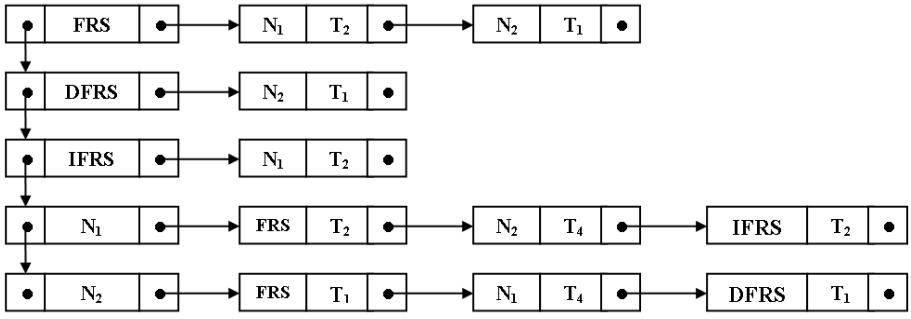

Figure 6.1: Linked-list holding the application graph of the flight reservation system

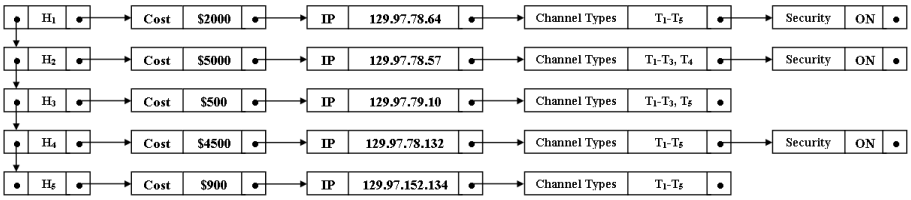

Figure 6.2: Linked-list holding the properties of available hosts in the target environment of the flight reservation system

points to a linked-list containing the information about adjacent components and the channel types used to connect the current component to them (Fig. 6.1). Also, the information about the target environment is kept as a linked-list of hosts. Each host points to a linked-list holding the properties of that host (Fig. 6.2). As we see, this linked-list data structure is flexible and gives us the freedom to define as many properties as we want for different hosts. After processing the input files and generating these linked-lists, the deployment planner tool uses them and starts to generate the actual deployment plan.

Chapter 7

Conclusions and Future Work

The software deployment process is defined as a sequence of related activities that makes an already developed application available for use in its operational environment. For simple stand-alone applications that should be installed only on a single computer, this process is easy. However, for complex component-based applications that should be deployed into a large distributed environment and some QoS parameters have to be also optimized, the deployment process is not that straightforward. In this report, we presented our ongoing work on addressing this problem. The main aim of this work is to design and develop an automated planner for the deployment of channel-based, component-based applications into distributed environments. For this purpose, we used the concept of peer-to-peer communication channels to capture the properties of interconnections among the components of the application. Then, our deployment planner did the planning with respect to the various channel types (or channel implementations) required by different components of the application, and various channel types (or channel implementations) that different hosts in the target environment can support.

In this report, we proposed a graph-based approach to address this deployment problem. In this approach, the component-based application to be deployed is modeled as a graph of components connected by different channel types. The target environment is also modeled as a graph of hosts connected by different channel types that can exist among them. Then, the deployment problem is defined as finding efficient algorithms for mapping the application graph to the target environment graph so that the desired QoS parameter is optimized. This report presented required algorithms for optimizing the cost and reliability of the deployments.

For future work, we intend to solve the deployment problem for other QoS parameters such as performance and security. More specifically, we have the following research objectives in our work:

- Developing a number of general and efficient algorithms for different QoS parameters to effectively deploy channel-based, component-based applications into distributed environments;
- Developing a deployment planner tool by using the algorithms designed in step one;
- Proving the correctness of our algorithms by applying them on a number of case studies.

Bibliography

[1] Gibbs, W. W. Software's Chronic Crisis. *Scientific American*, September 1994, pp. 86-95

[2] Karlsson, E. A., Sorumgard S., and Tryggeseth, E. Classification of Object-Oriented Components for Reuse. In *Proceedings of the 7th International Conference on Technology of Object-Oriented Languages and Systems* , Dortmund, Germany, 1992, pp. 21-31.

[3] Leue, S. *Software Engineering: Introduction and History*, http://tele.informatik.uni-freiburg.de/ leue/IU/it460.part1.pdf.

[4] Brooks, J. F. P. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20, 4 (April 1987), pp. 10-19.

[5] Butler, G. Quality and Reuse in Industrial Software Engineering. In *Proceedings of Asia-Pacific Software Engineering Conference and International Computer Science Conference*, December 1997, Hong Kong, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 3-12.

[6] Troya J. M., and Vallecillo, A. On the Addition of Properties to Components. In *Proceedings of the 2nd International Workshop on Component-Oriented Programming (WCOP'97)*, June 1997, Jyvskyl, Finland, pp. 95-103.

[7] Brown A. W., and Short, K. On Components and Objects: The Foundations of Component-Based Development. In *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*, June 1997, Pittsburgh, PA, USA, IEEE Computer Society Press, pp. 112-121.

[8] Reekie H. J., and Lee, E. A. *Lightweight Component Models for Embedded Systems*. Technical report, Electronics Research Laboratory, University of California at Berkeley, UCB ERL M02/30, October 2002.

[9] Yacoub, S., Ammar, H., and Mili, A. Characterizing a Software Component. In *Proceedings of the 2nd International Workshop on Component-Based Software Engineering, in conjunction with IEEE/ACM 21st International Conference on Software Engineering (ICSE99)*, Los Angeles, CA, USA, May 1999.

[10] Szyperski, C. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 1999.

[11] Kozaczynski, W. Composite Nature of Component. In *Proceedings of the 1999 International Workshop on Component-based Software Engineering*, May 1999, pp. 73-77.

[12] Goulo, M., and Abreu, F. B. Towards a Components Quality Model. *Work in Progress Session of the 28th Euromicro Conference (Euromicro 2002*, Dortmund, Germany, 2002.

[13] Yacoub, S., Mili, A., Kaveri, C., and Dehlin, M. A Hierarchy of COTS Certification Criteria. In *Proceedings of the 1st Software Product Line Conference (SPLC1)*, Denver, Colorado, USA, August 2000.

[14] Aitken, A. M. Components and Component-Based Software Development. In *Proceedings of the 3rd Western Australian Workshop on Information Systems Research (WAWISR)*, Edith Cowan University, Perth, WA, USA, November 2000.

[15] Ochs, M., Pfahl, D., Chrobok-Diening, G., Nothhelfer-Kolb, B. A COTS Acquisition Process: Definition and Application Experience. In *Proceedings of the 11th ESCOM Conference*, Munich, Germany, Shaker Publications, April 2000, pp. 335-343.

[16] Wuyts R., and Ducasse, S. Composition Languages for Black-Box Components. In *Proceedings of the 1st OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa Bay, Florida, USA, October 2001.

[17] Orso, A., Harrold, M. J., and Rosenblum, D. S. Component Metadata for Software Engineering Tasks. In *Proceedings of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000)*, LNCS 1999, Davis, CA, USA, November 2000, pp. 126-140.

[18] Crnkovic, I., Component-based Software Engineering - New Challenges in Software Development, *Software Focus*, 2, 4 (Winter 2001), 127-133.

[19] *RPM Package Manager*, http://www.rpm.org/.

[20] *InstallShield Developer*, http://www.installshield.com/isd/.

[21] *Zero G Software Deployment and Lifecycle Management Solutions*, http://www.zerog.com/.

[22] *Java Web Start Technology*, http://java.sun.com/products/javawebstart.

[23] *Microsoft Windows Update*, http://update.microsoft.com.

[24] *ClickOnce: Deploy and Update Your Smart Client Projects Using a Central Server*, MSDN Magazine, May 2004, http://msdn.microsoft.com/msdnmag/issues/04/05/clickonce/default.aspx.

[25] *Dell OpenManage Systems Management*, http://www1.us.dell.com/content/topics/global.aspx/solutions/en/openmanage.

[26] *IBM eServer BladeCenter Systems Management*, http://www.redbooks.ibm.com/redpapers/pdfs/redp3582.pdf.

[27] *Systems Management Server Home*, http://www.microsoft.com/smserver/.

[28] *IBM Tivoli Software*, http://www.tivoli.com/.

[29] *Altiris Deployment Solution*, http://www.altiris.com/products/deploymentsol/.

[30] Carzaniga, A., Fuggetta, A., Hall, R. S., Hoek, A. V. D., Heimbigner, D., Wolf, A. L. *A Characterization Framework for Software Deployment Technologies*. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

[31] Hall, R.S., Heimbigner, D., and Wolf, A.L. A Cooperative Approach to Support Software Deployment Using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, ACM Press, New York, May 1999, pp. 174-183.

[32] Heimbigner, D., Hall, R.S., and Wolf, A.L. A Framework for Analyzing Configurations of Deployable Software Systems. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems*, October 1999, p. 32.

[33] Lestideau, V. and Belkhatir, N. Providing Highly Automated and Generic Means for Software Deployment Process. In *Proceedings of the 9th International Workshop on Software Process Technology (EWSPT 2003)*, Helsinki, Finland, September 1-2, 2003, pp. 128-142.

[34] Object Management Group, *Deployment and Configuration of Component-based Distributed Applications Specification*, http://www.omg.org/docs/ptc/04-05-15.pdf.

[35] Bulej, L. and Bures, T. Using Connectors for Deployment of Heterogeneous Applications in the Context of OMG D&C Specification. In *Proceedings of the INTEROP-ESA 2005 Conference*, Geneva, Switzerland, Feb 2005.

[36] Hnetynka, P. Making Deployment of Distributed Component-based Software Unified. In *Proceedings of CSSE 2004 (part of ASE 2004)*, Linz, Austria, Austrian Computer Society, Sep 2004, pp. 157-161.

[37] Hnetynka, P. *Component Model for Unified Deployment of Distributed Component-based Software*. Technical Report No. 2004/4, Department of Software Engineering, Charles University, Prague, Jun 2004.

[38] *CORBA Component Model (v3.0)*, http://www.omg.org/technology/documents/formal/components.htm.

[39] *Enterprise JavaBeans Technology*, http://java.sun.com/products/ejb/.

[40] Brunneton, E., Coupaye, T., Stefani, J. B. *The Fractal Component Model*, http://fractal.objectweb.org/.

[41] Plasil, F., Balek, D., Janecek, R. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In Proceedings of the Fourth IEEE International Conference on Configurable Distributed Systems (ICCDS'98), Annapolis, USA, 1998.

[42] Magee, J., Kramer, J. Dynamic Structure in Software Architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE-4)*, San Francisco, USA, 1996.

[43] Allen, R. *A Formal Approach to Software Architecture*, PhD thesis, Carnegie Mellon University, 1997.

[44] Garlan, D., Monroe, R. T., Wile, D. Acme: Architectural Description of Component-based systems. In *Foundation of Component-based Systems*, G. T. Leavens and M. Sitaraman (eds), Cambridge University Press, 2000.

[45] Balek, D., Plasil, F. Software Connectors and Their Role in Component Deployment, In *Proceedings of DAIS'01*, Krakow, Kluwer, Sep. 2001.

[46] Rus, D., Gray, R., Kotz, D. Transportable Information Agents. In *Proceedings of the first ACM International Conference on Autonomous Agents*, 1997, pp. 228-236.

[47] Silva Filho, R.S. *Mobile Agents and Software Deployment*. ICS280-Configuration Management and Runtime Change Final Paper, Information and Computer Science Department, University of California Irvine, Fall 2000.

[48] Sudmann, N.P. and Johansen, D. Software Deployment Using Mobile Agents. In *Proceedings of First International IFIP/ACM Working Conference on Component Deployment (CD 2002)*, Berlin, Germany, June 20-21, 2002.

[49] Mikic-Rakic, M., Malek, S., Beckman, N. and Medvidovic, N. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. In *Proceedings of the Second International Working Conference on Component Deployment (CD 2004)*, Edinburgh, UK, May 20-21, 2004.

[50] Wichadakul, D., and Nahrstedt, K. A Translation System for Enabling Flexible and Efficient Deplyoment of QoS-aware Applications in Ubiquitous Environments. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD 2002)*, Berlin, Germany, 2002.

[51] Mikic-Rakic, M. and Medvidovic, N. Architecture-Level Support for Software Component Deployment in Resource Constrained Environments. In *Proceedings of the First International IFIP/ACM Working Conference on Component Deployment (CD 2002)*, Berlin, Germany, June 20-21, 2002.

[52] Lacour, S., Prez, C., and Priol, T. A Software Architecture for Automatic Deployment of CORBA Components Using Grid Technologies. In *Proceedings of the First Francophone Conference On Software Deployment and (Re)Configuration (DECOR 2004)*, Grenoble, France, October 2004, pp. 187-192.

[53] Arbab, F. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science*, 14, 3 (June 2004), pp. 329-366.

[54] Arbab, F. and Mavaddat, F. Coordination through channel composition. In *Proceedings of the 5th International Conference on Coordination Models and Languages (Coordination 2002)*, LNCS 2315, Springer-Verlag, pp. 21-38.

[55] WEB SERVICES: FOOD FOR THOUGHT. http://www.cutter.com/webservices/wss0209.pdf.

[56] Web Services Conceptual Architecture. http://www-306.ibm.com/software/solutions/webserv ices/pdf/WSCA.pdf.

[57] Schollmeier, R. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In *Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001)*, Linkping, Sweden, August 27-29, 2001.

[58] Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. *Introduction to Algorithms*, Second edition, MIT Press, 2001.

[59] Lyu, M. R. *Handbook of Software Reliability Engineering*, IEEE Computer Society Press and McGraw-Hill, 1996.

[60] Arbab, F., de Boer, F. S., Bonsangue, M. M., and Guillen-Scholten, J. V. MoCha: A Middleware Based on Mobile Channels. In *Proceedings of COMPSAC 2002*, IEEE Computer Scociety Press, 2002.

[61] Katis, P., Sabadini, N. and Walters, R. F. C. A Formalization of the IWIM Model. In *Proceedings of the 4th International Conference on Coordination Languages and Models (COORDINATION 2000)*, Limassol, Cyprus, September 11-13, 2000.

[62] Bonsangue, M. M., Arbab, F., de Bakker, J. W., Rutten, J., Scutell, A. and Zavattaro, G. A Transition System Semantics for the Control-driven Coordination Language Manifold. *Theoretical Computer Science*, 240, 1 (June 2000), pp. 3-47.

[63] Dahlhaus, E., Johnson, D. S., Papadimitriou, C. H., Seymour, P. D. and Yannakakis, M. The Complexity of Multiterminal Cuts. *SIAM Journal on Computing*, 23, 4 (August 1994), 864-894. Preliminary version appeared in STOC'92.

[64] Calinescu, G., Karloff, H., and Rabani, Y. An Improved Approximation Algorithm for Multiway Cut. *Journal of Computer and System Sciences*, 60, 3 (June 2000), 564-574. Preliminary version in STOC'98.

[65] Vazirani, V. V. *Approximation Algorithms*, Second Edition, Springer, 2002.