A USER-CENTRIC APPROACH FOR IMPROVING A DISTRIBUTED SOFTWARE SYSTEM'S DEPLOYMENT ARCHITECTURE

by

Sam Malek

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(COMPUTER SCIENCE)

August 2007

DEDICATION

To my loving family

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor, Professor Nenad Medvidovic, for his support and guidance during the past five years. I am forever grateful to him for encouraging me to pursue my Ph.D. studies, which turned out to be one of the best decisions I have made in my life. It has been a privilege to learn so much from such an extraordinary individual. I am sure I will miss the days when I could knock on his door and pick his brain over a cup of coffee. I also would like to thank the other members of my committee, Professors Sandeep Gupta, Gaurav Sukhatme, Richard Taylor, and Barry Boehm for their feedback and devoting time from their busy schedules.

Discussions with my colleague and friend, Marija Mikic-Rakic, gave me the motivation to pursue this research topic. I am always grateful to her for the countless times she helped me with patience. I owe thanks to Nels Beckman for his significant contributions to the development of DeSi. I wish to thank Chiyoung Seo for his help with the implementation and evaluation of this dissertation using MIDAS. I have also been really fortunate to work with many other good friends and colleagues in the Software Architecture Group at USC: Roshanak Roshandel, Vladimir Jakobac, Chris Mattmann, David Woolard, Somo Banerjee, George Edwards, Daniel Popescu, and Yuriy Brun.

I have been very fortunate to have the love and support of my family and friends throughout this journey. To my dear parents, Manouchehr and Zinat, and older brother, Mike, I am forever grateful for all the sacrifices you have made for me. Finally, I would like to thank my fiance and best friend, Shadi, for all her love, support, and encouragement. Without you I could not have made it.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

ABSTRACT

The quality of service (QoS) provided by a distributed software system depends on many system parameters, such as network bandwidth, reliability of links, frequencies of software component interactions, etc. A distributed system's allocation of software components to hardware nodes (i.e., *deployment architecture*) can have a significant impact on its QoS. At the same time, often times there are many deployment architectures that provide the same functionality in large-scale software systems. Furthermore, the impact of deployment architecture on the QoS dimensions (e.g., availability, latency) of the services (functionalities) provisioned by the system could vary. In fact, some QoS dimensions may be conflicting, such that a deployment architecture that improves one QoS dimension, degrades another dimension.

In this dissertation, we motivate, present, and evaluate a framework aimed at finding the most appropriate deployment architecture with respect to multiple, and possibly conflicting, QoS dimensions. The framework provides a formal approach to modeling the problem, and a set of generic algorithms that can be tailored and instantiated for improving a system’s deployment architecture. The framework relies on system users’ (desired) degree of satisfaction with QoS improvements to resolve trade-offs between conflicting QoS dimensions. The framework is realized on top of an integrated tool suite, which further aids reusability and cross-evaluation of the solutions.

This dissertation is evaluated empirically on a large number of simulated representative scenarios. Various aspects of the framework have also been evaluated on two real distributed systems. The dissertation concludes with several open research questions that will frame our future work.

CHAPTER 1: Introduction

Software systems are continuously growing in size and complexity. In recent years, they have also increasingly migrated from the traditional, desktop setting to highly distributed, mobile, possibly embedded and pervasive computing environments. Such environments present daunting technical challenges: effective understanding of existing or prospective software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data, numbers of data types, and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute on "small" devices, characterized by highly constrained resources such as limited power, low network bandwidth, slow CPU speed, limited memory, and small display size. We refer to the development of software systems in the described setting as *programming-in-the-small-and-many (Prism)*, both for exposition purposes, but also in order to distinguish it from the traditional software engineering paradigm of programming-in-the-large (*PitL*) [7], which has been primarily targeted at desktop computing.

Software engineering researchers and practitioners have successfully dealt with the increasing complexity of PitL systems by employing the principles of software architecture. *Software architecture* provides design-level models and guidelines for composing the structure, behavior, and key properties of a software system [49]. An

architecture is described in terms of software *components* (computational elements) [61], software *connectors* (interaction elements) [37], and their *configurations* (also referred to as *topologies*) [35].

More recently, software architectural principles have also shown to be effective in overcoming the challenges facing the engineers in the Prism setting [24,28,31,33,34,36,57]. In [28,31,33,34] we showed that software-architecture based development of Prism applications is a viable approach to developing efficient and scalable solutions in this setting. Moreover, we showed that software-architecture based development directly aids the engineers in the analysis and reconfiguration of Prism software systems, possibly at runtime. As will be detailed in the remainder of this dissertation, we have directly leveraged software architectural concepts in accomplishing the objective of this thesis.

A representative Prism application family on which the author of this dissertation has worked in cooperation with a third-party organization is Troops Deployment and battle Simulations (TDS). This application family is intended to deal with situations such as distributed deployment of personnel in cases of natural disasters, search-and-rescue efforts, and military crises. A specific instance of this application family is shown in Figure 1-1 with single *Headquarters*, four *Commander*s, and 36 *Soldiers*. TDS is one of the applications we use to illustrate the concepts throughout this dissertation. The headquarters computer is networked to a set of PDAs used by

“Commanders” in the field. The commander PDAs are connected directly to each other and to a large number of “Troop” PDAs. These devices communicate and help to coordinate the actions of their distributed users. The distributed software system running on these devices provides a number of services to the users: requesting and sending information to each other, viewing the current field map, managing the resources, etc.

TDS’s architecture consists of the following set of components. A Map component maintains a model of the system's overall resources: terrain, personnel, tank units, and mine fields. These resources are permanently stored inside a Repository component. StrategyAnalyzerAgent, DeploymentAdvisor, and SimulationAgent components,

Figure 1-1: Instance of TDS.

respectively, (1) analyze the deployments of friendly troops with respect to enemy troops and obstacles, (2) suggest deployments of friendly troops based on their availability as well as positions of enemy troops and obstacles, and (3) incrementally simulate the outcome of the battle based on the current situation in the field. StrategyAnalysisKB and SAKBUI components store the strategy rules and provide the user interface for changing these rules, respectively. ResourceManager, CommanderManager, SoldierManager, and ResourceMonitor components enable allocation and transfer of resources and periodically update the state of resources. Weather and WeatherAnalyzer components provide weather information and analyze the effects of weather conditions. Finally, UI components provide the user interface of the application.

Prism applications, such as TDS, are highly distributed, decentralized, and mobile, and therefore highly dependent on the underlying network. They are frequently challenged by the fluctuations in the system's parameters: network disconnections, bandwidth variations, unreliability of hosts and/or network links, etc. Furthermore, the different users' usage of the functionality (i.e., *services*) provided by the system and the users' quality of service (QoS) preference for those services will differ, and may change over time. For example, in the case of a disaster search-and-rescue effort scenario, "Commander" users may require a secure and reliable messaging service with the "Headquarters" when exchanging search-and-rescue plans. On the other hand, "Troop" users may be more interested in having a low latency messaging

service with other “Troop” users when sending assistance requests. However, security of such low-latency interactions may also become imperative if, at some point, a malicious intruder is able to break into the system and introduce “false alarms”.

For any large, distributed system, such as TDS, many deployment architectures (i.e., mappings of software components onto hardware hosts) will be typically possible. Figure 1-2 shows an example of a deployment architecture for the instance of TDS application discussed earlier. Given a set of feasible deployment architectures for an application, some will be more effective in delivering the desired level of service

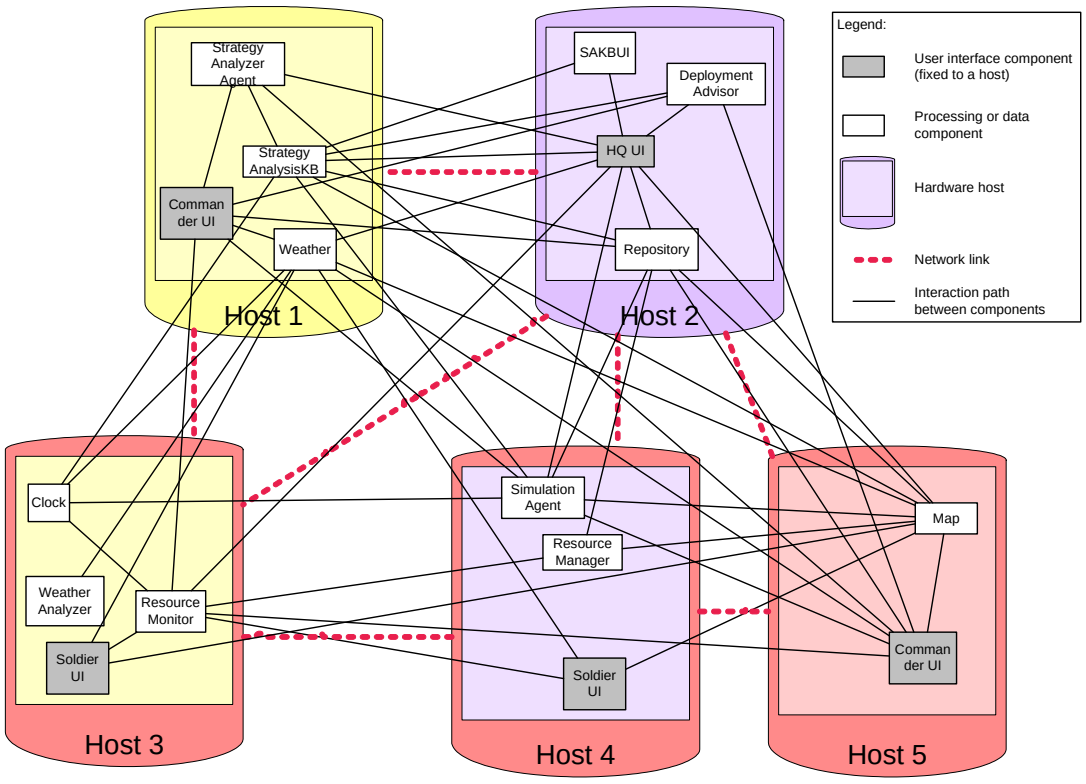


Figure 1-2: A sample deployment architecture of TDS, consisting of single Headquarters (host 2), two Commanders (hosts 1 and 5), and two Soldiers (hosts 3 and 4)

quality to the user. For example, a service's latency can be improved if the system is deployed such that the most frequent and voluminous interactions occur either locally or over reliable and capacious network links. This problem becomes quickly intractable for a human engineer, however, if multiple QoS dimensions (e.g., latency, security, availability, power usage) must be considered simultaneously, while taking into account any additional constraints (e.g., component X may not be deployed on hosts Y and Z). Figure 1-3 shows an overview of the various elements of this problem, which we will discuss in more detail below.

In this dissertation, we consider the problem of finding a deployment architecture such that the QoS preferences (i.e., *utility*) accrued by a collection of distributed end-users is maximized. We would like our solution to be applicable to a wide range of *application scenarios* (i.e., differing numbers of users, hardware hosts, software components, application services, QoS dimensions, etc.). However, a widely applicable solution to this problem is challenged by the following:

- A very large number of system parameters influence QoS dimensions of a software system. Many services and their corresponding QoS dimensions influence the users' satisfaction. Formally modeling and developing generic solutions that can be customized based on the application scenario is challenging.

- Different QoS dimensions may be conflicting, and users with different priorities may have conflicting QoS preferences. Fine-grain trade-off analysis without relying on simplifying assumption (e.g., particular definition of a QoS objective, predetermined constraints) is challenging.
- Different application scenarios require different algorithmic approaches. For example, a system's size, the users' usage of the system, stability of system's parameters, and its centralization characteristic determine the best algorithm for execution.
- Traditional software engineering tools are not applicable to this problem. Therefore, engineers have to spend a significant amount of time adapting tools intended for different purposes to the deployment improvement problem, which limits the potential for reuse and cross-evaluation of the solutions.

We have developed a tailorable framework that is targeted at the above challenges. The framework (1) provides an extensible model that supports inclusion of arbitrary system parameters; (2) supports definition of new QoS dimensions using the system parameters; (3) allows users to specify their QoS preferences in terms of QoS utility, which indicates a user's (desired) degree of satisfaction with a system; (4) provides several generic algorithms with different characteristics, where each is suitable for maximizing the users' satisfaction in a particular scenario, and none relies on simplifying assumptions that could hinder its applicability; (5) provides support for

monitoring and visualizing arbitrary system parameters; and (6) supports initial deployment, execution, and runtime redeployment of the software system.

The framework relies on the notion of *QoS utility,* which indicates a user's (desired) degree of satisfaction with a system. The framework's objective is to maximize the overall utility, i.e., the cumulative satisfaction with the system by all its users. Given

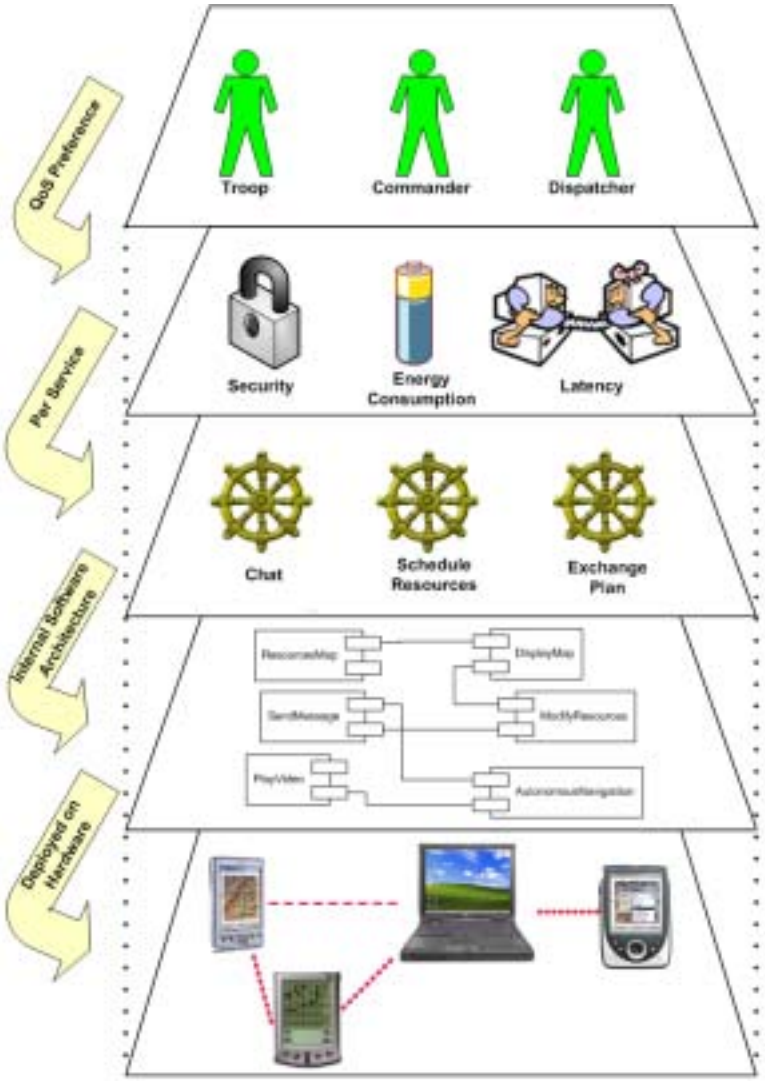


Figure 1-3: Problem overview.

an application scenario, the system architect instantiates (configures) the framework by defining the appropriate system parameters and the QoS of interest. The framework is then populated with the actual data from a distributed application and users' preferences for the QoS dimensions of each application service. Afterwards, one of the algorithms supplied by the framework is used to find an improved deployment architecture. Finally, the solution is effected by (re)deploying the system.

The theoretical aspects of the work, which include a generic formal model of a distributed system's deployment architecture and the accompanying generic algorithms, are independent of any implementation platform. The theoretical results are realized on top of an integrated tool suite, which allows the engineer to instantiate the model for an application scenario and use one of the provided algorithms for improving its architecture. We have developed a customizable deployment analysis environment (called DeSi [43]) and an extensible architectural middleware (called Prism-MW [28]). Together, DeSi and Prism-MW provide an integrated tool suite that can be leveraged by software engineers to develop reusable solutions to this problem. Prism-MW will provide the ability to (re)deploy and execute a distributed system in terms of its architectural components, and monitor arbitrary system parameters. DeSi provides the ability to model the system's deployment architecture, populate those models with the runtime monitoring data from Prism-MW, assess its architecture, improve it via one of the deployment improving algorithms, and effect the improved architecture by sending commands to Prism-MW.

We have evaluated the framework on a number of simulated and real distributed systems, including systems with 1) multiple QoS dimensions (e.g., availability, latency, communication security, and energy consumption), 2) multiple real and simulated users with varying QoS preferences, 3) systems with different characteristics (small vs. large, stable vs. unstable, centralized vs. decentralized). The algorithms are evaluated for their ability to improve the user's QoS preferences. The tool suite is evaluated for its ability to promote reusability and cross-evaluation of the solutions.

1.1 Research Hypotheses

The described research has been guided by five hypotheses that are discussed below.

1.1.1 Optimization Algorithms

Finding the optimal deployment architecture is an exponentially complex problem (in terms of the number of possible deployment architectures, h^c, where h is the number of hosts, and c is the number of components). In addition, the numbers of QoS dimensions, users, and provisioned services also have a significant impact on the complexity of finding solutions. For these reasons, it may be impossible to invest the necessary time to find the optimal solution.

Hypothesis #1: *An algorithm of at most polynomial complexity in the number of components and hosts, and linear in the numbers of QoS dimensions, users, and*

services can be devised with the ability to find a deployment architecture such that (1) the overall utility to system users will be within 20% of a target (e.g., known optimal) architecture's utility, or (2) when a known optimal architecture does not exist, the overall utility will improve on average 30% over the statistical average utility, which is the average utility of a set of randomly selected deployment architectures.

1.1.2 Sensitivity to Users and Services

Some users of the system may be more important than others, which implies that their preferences should be given a higher priority. Similarly, some services may be more critical than others, denoted by the amount of QoS preferences associated with them, which implies that they should be given a higher priority.

Hypothesis #2: *An optimization algorithm can be devised that performs fine-grain trade-off analysis, such that given two identical application scenarios X and Y that only differ in the priority of a user U, if U has a higher priority in X than in Y, then after executing the algorithms on both X and Y, the overall utility gain for U in X is greater than or equal to its utility gain in Y.*

Hypothesis #3: *An optimization algorithm can be devised that performs fine-grain trade-off analysis, such that given two identical application scenarios X and Y that differ only in the criticality of service S, if S is more critical in X than in Y, then after executing the algorithm on both X and Y, the overall QoS improvement for S in X is greater than or equal to its improvement in Y.*

1.1.3 Decentralization

Centralized algorithms depend on the existence of a host with the global knowledge of the system. However, this is not feasible in a growing class of decentralized systems, where each host has only partial knowledge of the system.

Hypothesis #4: *A decentralized optimization algorithm can be devised that (1) when there is a modest 20% lack of knowledge (i.e., each host on average does not know about 20% of the hosts in the system) finds solutions that on average come within 10% of the best solution produced by the centralized optimization algorithms, and (2) while there are no completely disconnected hosts, the solution accuracy degrades gracefully as the lack of knowledge increases on each host, such that the decrease rate in the solution accuracy is significantly lower than the increase rate in the lack of knowledge.*

1.1.4 Algorithmic Trade-Offs

There exist inherent trade-offs among the deployment improvement algorithms. Each deployment improvement algorithm has its own unique property that makes it more suitable to a class of systems.

Hypothesis #5: *It is possible to determine the best algorithm for execution in terms of the accuracy of the solution and the performance of the algorithm given the system's architectural style (e.g., client-server vs. peer-to-peer), its stability (amount of*

fluctuation in system parameters, which impacts the available time for estimation), centralization, the number of system parameter constraints (highly vs. lightly constrained), and the complexity of the application scenario (which includes number of hosts, components, logical links, physical links, users, services, and QoS dimensions).

The remainder of the dissertation is organized as follows. Chapter 2 discusses the challenges of analyzing a system's deployment architecture, and the approach we have taken in overcoming those challenges. Chapter 3 presents an overview of the related work. Chapter 4 describes the different elements of the deployment improvement framework and their relationships. Chapter 5 presents the framework's underlying formal model of the problem, as well as its instantiation for a particular application scenario. Chapter 6 describes five classes of algorithms provided by the framework. Chapter 7 presents the framework's tool support in the Prism setting. Chapter 8 presents a detailed evaluation of the framework, including the algorithms and the tool-support. Finally, Chapter 9 presents concluding remarks and an overview of the future work.

CHAPTER 2: Analyzing Deployment Architecture

In this section, we discuss the challenges of analyzing and improving a system's deployment architecture via several simple examples. We also present our approach to overcoming these challenges. This section also provides the appropriate background for understanding the formal descriptions of the framework's underlying model and algorithms, which are presented in Chapters 5 and 6 respectively.

Figure 2-1 shows a very simple application scenario of one QoS dimension (*latency*), one service (*Schedule Resources*), two software components (*ModifyResourceMap* and *ResourceMonitor*), and two hardware hosts. In this scenario, the number of possible deployment architectures can be calculated as: $number\ of\ hosts^{number\ of\ comps} = 2^2 = 4$. Our objective is to find the deployment architecture that minimizes the latency of the *Schedule Resources* service. Figure 2-2 shows the latency of the four deployment architectures. A QoS dimension is quantified based on the various system properties. For example, latency of a service can be quantified as the product of the number of messages exchanged between software components and the network transmission delays. We postpone the details of how we quantify the QoS dimensions of a deployment architecture to Chapter 5. As shown in Figure 2-2, deployment 1 has the smallest latency. Most likely this corresponds to the configuration where the two software components are collocated on the host with the fastest CPU (i.e., laptop). Therefore, it is the

optimal deployment (i.e., a solution that dominates every other solution) for this application scenario. At first blush this scenario may seem very trivial. However, the problem becomes very challenging for larger scenarios. This is due to the fact that

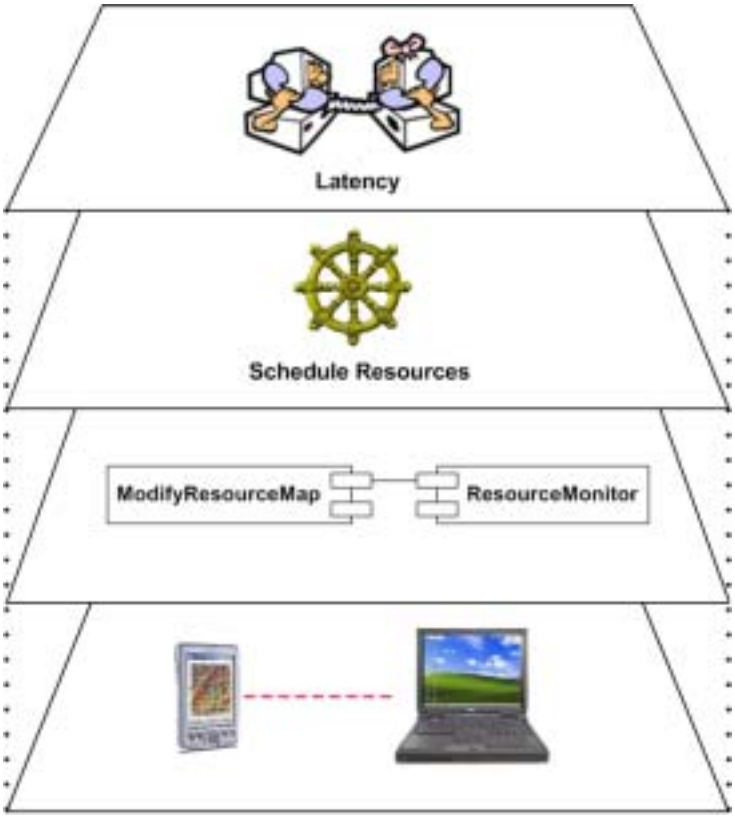


Figure 2-1: A very simple application scenario of one QoS dimension.

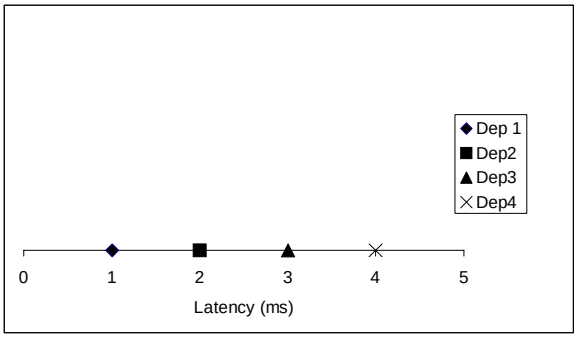


Figure 2-2: Latency of service *Schedule Resources* in the four possible deployment architectures of application scenario shown in Figure 2-1.

this problem grows exponentially in the number of components and hosts. In fact, as will be discussed in Chapter 3, all previous works [1,17,22], including our own work [29,40,42], have dealt with similar variations of the problem depicted in Figure 2-1. In this dissertation, not only do we want the framework to be applicable to large problems of the type shown in Figure 2-1, but also to more complex problems that are composed of multiple QoS dimensions and services, which we discuss next.

Figure 2-3 shows a more complex application scenario that is composed of two QoS dimensions: *latency* and *durability.* Durability in the context of this example is very

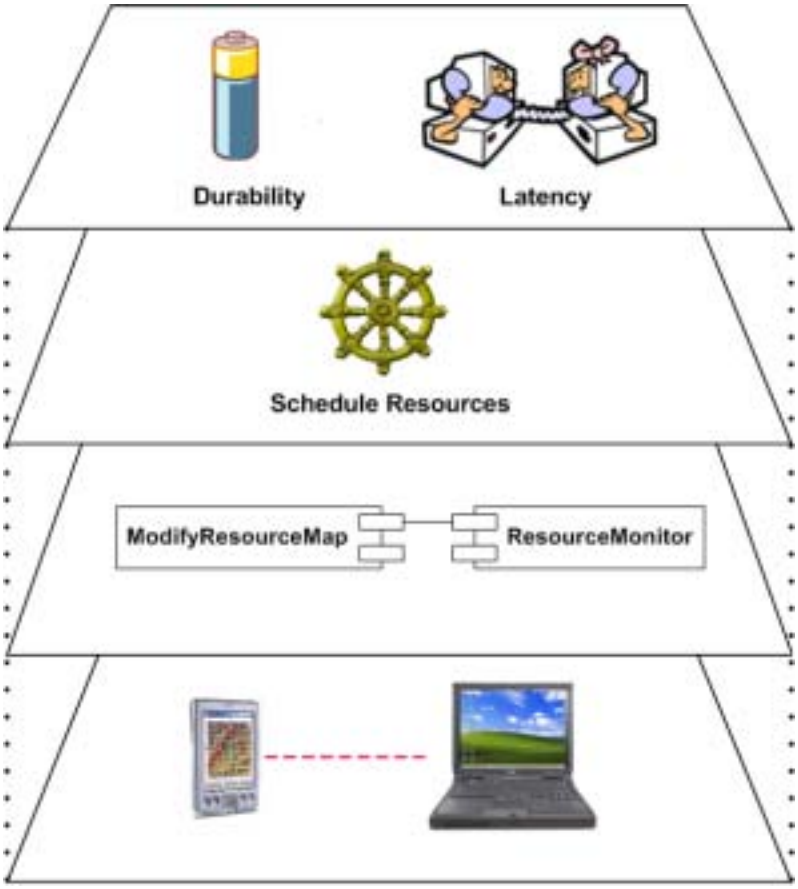


Figure 2-3: A more complicated application scenario of two QoS dimensions.

closely related to the concept of energy consumption (i.e., the amount of time a service is available for use depends on the amount of available battery power on the devices that provide that service). Therefore, unlike latency, we would like to maximize durability. As will be described in more detail later on, latency and durability are conflicting QoS dimensions (i.e., improving one may come at the expense of the other). Figure 2-4 shows the latency and durability of the 4 possible deployments for the *Schedule Resources* service. Unlike the application scenario of Figure 2-1, where there is always an optimal solution, in the application scenario of Figure 2-3 there is no optimal solution. With the exception of deployment 4 (that has both a higher latency and a lower durability than deployment 3), all the other deployments present trade-offs between latency and durability. This is a frequently encountered phenomenon in multi-dimensional optimization problems [65]. Basically it is not possible to improve any one objective without compromising another objective.

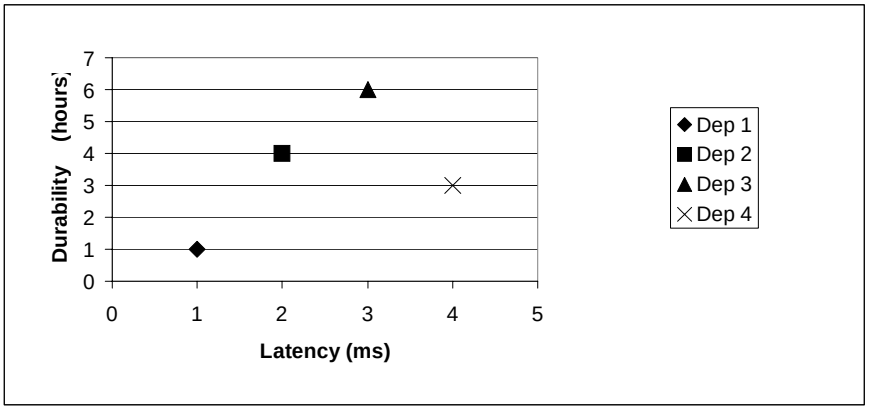


Figure 2-4: Latency versus durability of service *Schedule Resources* for the four possible deployment architectures of application scenario *shown in* Figure 2-3.

One approach to resolving the QoS trade-offs is to transform the multi-dimensional objective (i.e., the vector of QoS dimensions) to a scalar value via *utility functions*. A utility function denotes a user's preference, expressed as a scalar value, for improving a given QoS dimension. Figure 2-5 shows a modified version of the application scenario of Figure 2-3. This application scenario introduces the notion of a user. Our problem lends itself directly to the notion of utility, as the users of services typically have varying preferences for the QoS dimensions of the provisioned services. Figure 2-6 shows the `Commander`'s utility functions for the two QoS dimensions of

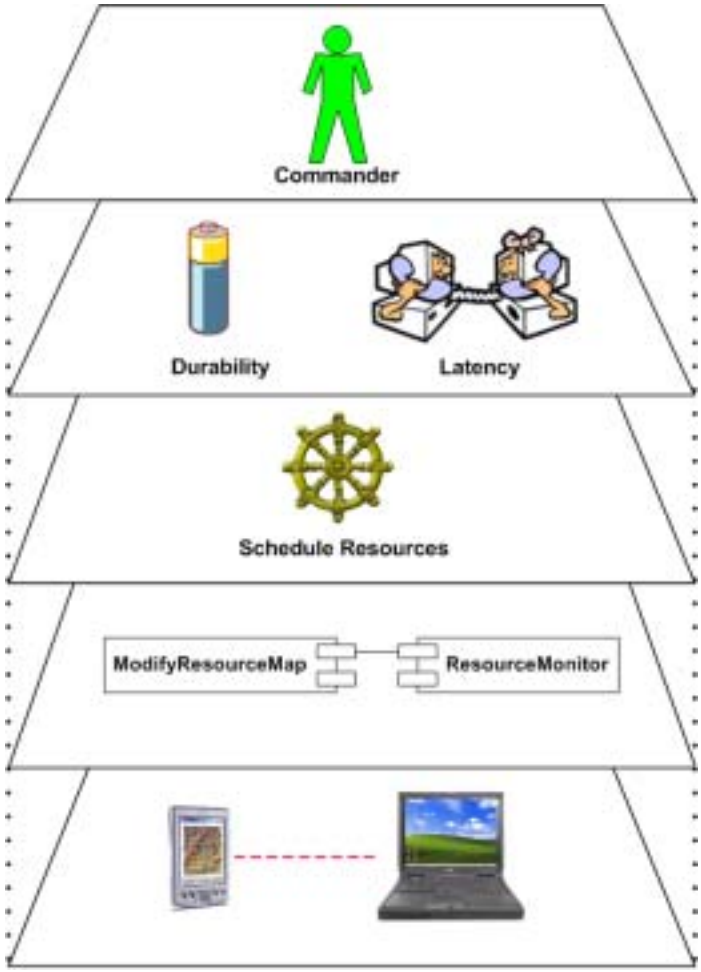
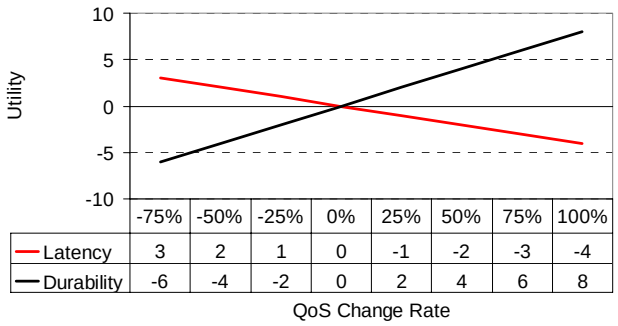


Figure 2-5: An application scenario that includes the notion of user and user preferences.

Schedule Resources. For example, it shows that for 25% increases in latency and durability, the user has specified utilities of -1 and 2, respectively. The type and slope of a utility function corresponds to the user's objectives. Since the commander prefers to decrease the latency of *Schedule Resources*, the corresponding utility function has a negative slope. On the other hand, since the commander prefers to increase the durability of *Schedule Resources*, the corresponding utility function has a positive slope.

The utility functions shown so far are linear, but the framework places no restrictions on the type of functions that represent users' preferences. In fact, typically users may not be able to express their preferences in terms of complex mathematical functions. Often times requirements documents specify constraints on QoS properties of services, which can be used to determine the initial utility functions for the various stakeholders in the system. The utility functions can also be derived by eliciting user preferences and expressing them in terms of a set of discrete data points (e.g., 50%



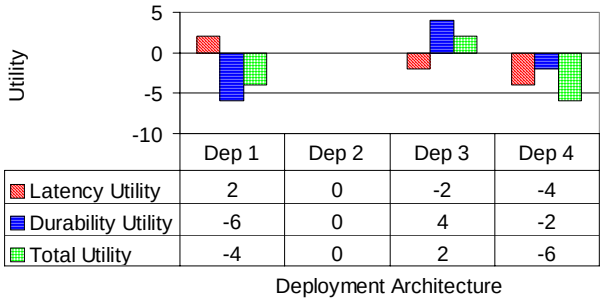
	-75%	-50%	-25%	0%	25%	50%	75%	100%
Latency	3	2	1	0	-1	-2	-3	-4
Durability	-6	-4	-2	0	2	4	6	8

Figure 2-6: The utility functions representing the *Commander*'s QoS preferences.

decrease in latency has a utility of 2 and so on), and then using one of the numerous curve fitting techniques (e.g., regression, interpolation) to determine a function that approximates the data points most accurately.

To determine the utility of changing the initial deployment, we first determine the rate of change for each QoS dimension (i.e., the amount of change in a QoS dimension if we were to modify the current deployment) from Figure 2-4, then look up the utility associated with each rate of change from Figure 2-6, and finally aggregate the utilities. Optimal deployment for the system is the one that has the highest total utility. Figure 2-7 shows the total utility of changing the system's deployment based on the assumption that deployment 2 is the initial deployment of the system. As shown in Figure 2-7, deployment 3 achieves a total utility of 2, which is the optimal deployment for this system.

Figure 2-8 shows a more complicated scenario, where there are two users as opposed to a single user. Figure 2-9 shows the newly added *Troop* user's utility functions. Note



	Dep 1	Dep 2	Dep 3	Dep 4
Latency Utility	2	0	-2	-4
Durability Utility	-6	0	4	-2
Total Utility	-4	0	2	-6

Figure 2-7: The utility of the four possible deployments for application scenario shown in Figure 2-5.

that the *Troop* user has an identical utility function to that of the *Commander* for durability. However, the *Troop* user has specified more utility for improvements (i.e., decrease) in latency than the *Commander* has. To determine the optimal deployment architecture when there is more than one user, the importance as well as the preferences of each user should be considered. However, for clarity in this example we assume the users have the same level of priority, and will revisit this issue in Chapter 5. Since both users have the same priority, we calculate the total accrued utility by both users for each of the 4 possible deployments, as shown in Figure 2-10. Unlike the application scenario of Figure 2-5, in this scenario the optimal

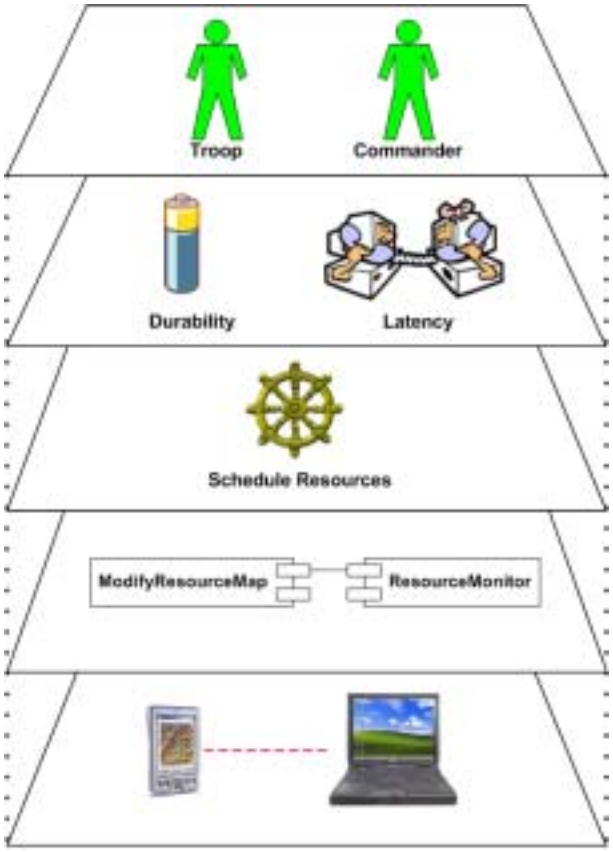
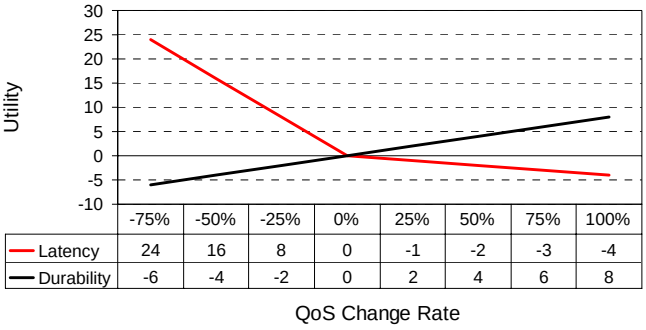


Figure 2-8: An application scenario that includes two users with different preferences.

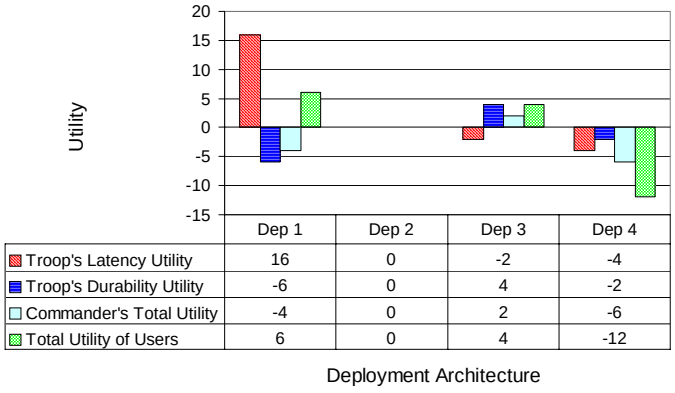
configuration is deployment 1. This is attributed to the fact that the *Troop* user has specified more utility for decreases in latency, which has resulted in the deployment architecture with the lowest latency to have the highest total utility.

As mentioned earlier, we intentionally chose very simple application scenarios in the above examples to introduce the various elements of our problem. However, often times the application scenarios are significantly more complex than the ones depicted



	-75%	-50%	-25%	0%	25%	50%	75%	100%
Latency	24	16	8	0	-1	-2	-3	-4
Durability	-6	-4	-2	0	2	4	6	8

Figure 2-9: The utility functions representing the Troop's QoS preferences.



	Dep 1	Dep 2	Dep 3	Dep 4
Troop's Latency Utility	16	0	-2	-4
Troop's Durability Utility	-6	0	4	-2
Commander's Total Utility	-4	0	2	-6
Total Utility of Users	6	0	4	-12

Figure 2-10: The utility of the four possible deployments for application scenario shown in Figure 2-8.

so far. Therefore, it becomes infeasible for the engineer to determine the optimal (or even a good) deployment architecture without relying on the appropriate tool support. We demonstrate this (as shown in Figure 2-11) by simply adding one more user, QoS dimension, software component, and hardware host to the problem of Figure 2-8. Figure 2-12 shows the 18 resulting utility functions (a utility function per user, QoS dimension, and service) that would have to be considered in the evaluation of candidate deployment architectures. Furthermore, by simply adding one more component and host, we have increased the number of possible deployment

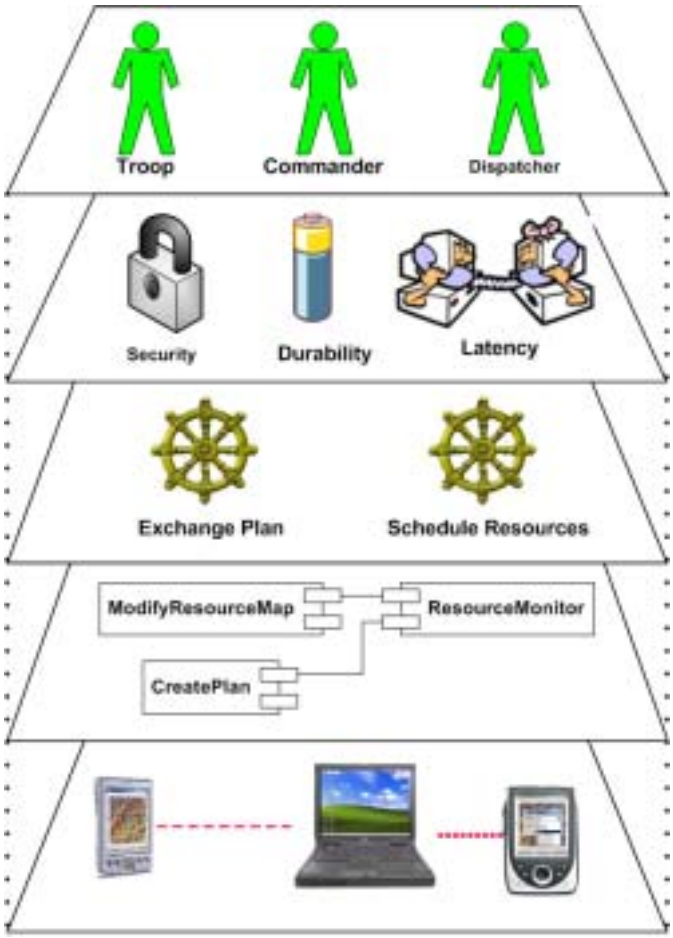


Figure 2-11: A more complicated application scenario.

architecture to 27 ($number\ of\ hosts^{number\ of\ comps} = 3^3$). Figure 2-13 shows the pair-wise QoS trade-offs between the 27 deployment architectures. Similar to the previous example, by leveraging the 18 utility functions of Figure 2-12 and the QoS trade-offs shown in Figure 2-13 it is possible to determine the optimal deployment architecture for this scenario as well. However, it demonstrates that as the problems grow in size, manual selection of an optimal deployment becomes impossible. As another example, consider the small problem of Figure 1-3, where 4096 deployment architectures would have to be considered. In fact, the exponential nature of this problem makes it also infeasible for an algorithmic solution to determine the optimal deployment of any sizable system. Therefore, we rely on optimization algorithms for finding "good" solutions that come close to the optimal solution, but can be computed very fast.

As discussed above, the complexity of improving system's deployment architecture has been one of the primary motivations behind the proposed research. Furthermore,

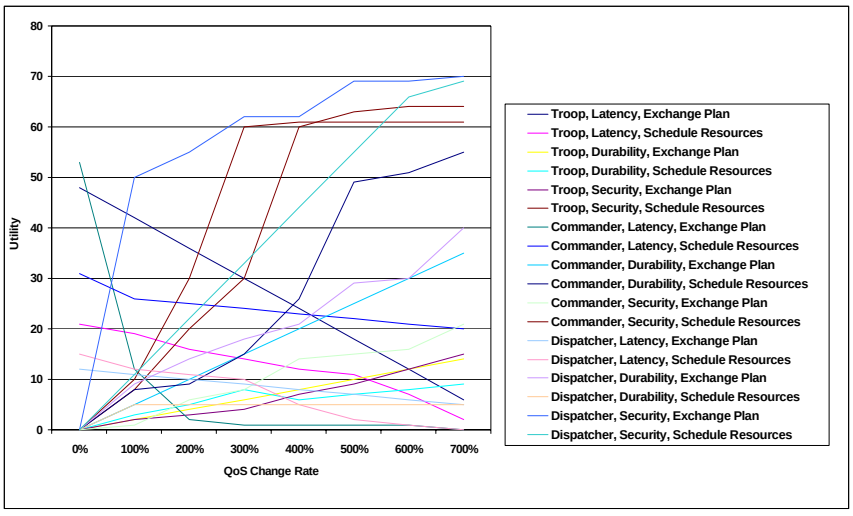


Figure 2-12: The utility functions representing the QoS preferences of the users.

as mentioned earlier, the simplifying assumptions made by previous works have resulted in solutions that are not generally applicable, which has been our motivation to solve this problem in its most general form.

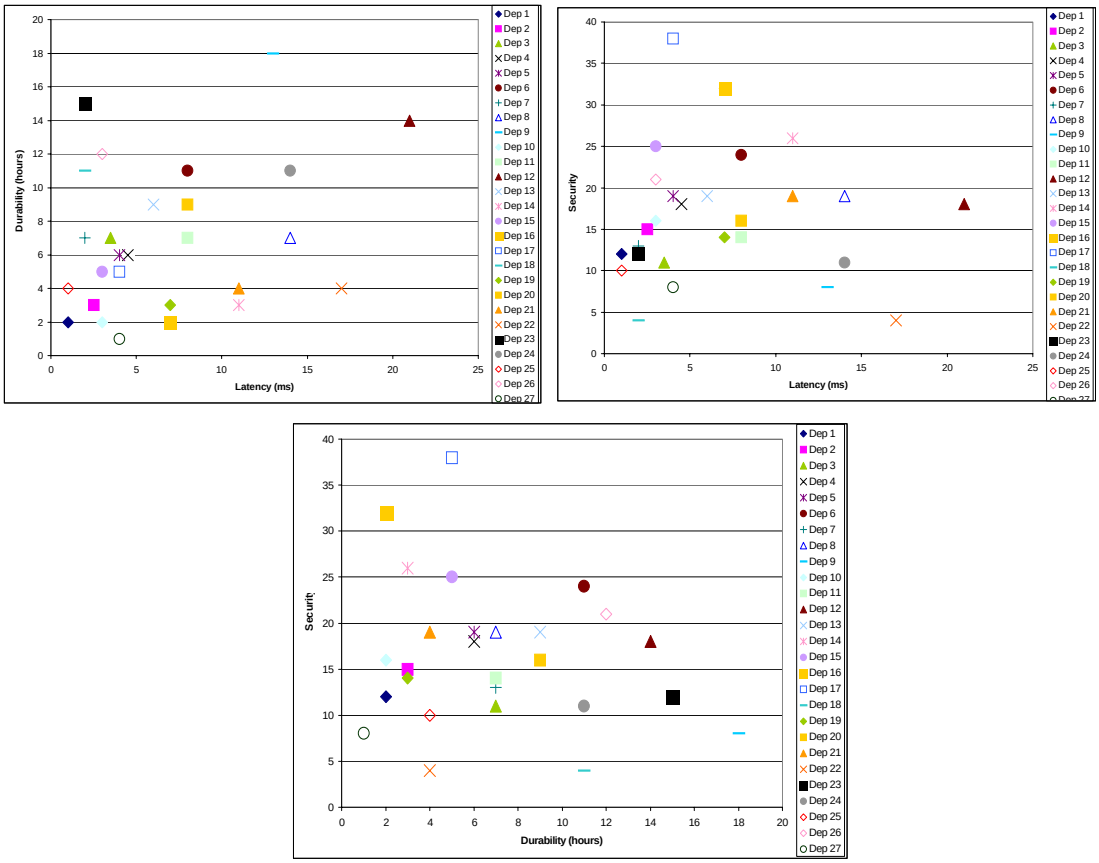


Figure 2-13: The QoS dimension trade-offs for the 27 possible deployment architectures.

CHAPTER 3: Related Work

The general area of software deployment has been studied extensively by the software engineering research community. However, only a small subset of these works have addressed the deployment issues from an architectural perspective. In this chapter, we provide an overview of the previous works that have addressed the challenges of modeling, assessing, and improving deployment of software architecture-based systems.

3.1 Deployment Modeling

UML [46] is the primary notation for the visual modeling of today's software systems. UML's deployment diagrams provide a standard notation for representing a system's software deployment architecture. Several recent approaches extend this notation via stereotypes [13,27]. However, using UML to visualize deployment architectures has several drawbacks: UML's deployment diagrams are static; they do not depict connections among hardware hosts; and they do not provide support for representing and visualizing the parameters that affect the key system properties (e.g., available network bandwidth). For these reasons, we opted not to use a UML-based notation in this research.

More applicable to generic purpose modeling of a system's deployment architecture is SysML [60], which is a modeling language standard for specifying system

engineering artifacts. Therefore, it is highly generic and extensible, such that different diagram types can be leveraged to model various aspects of a system. The most relevant SysML diagram for modeling software deployment is its *allocation* diagram, which allows arbitrary modeling elements to reference one another (e.g., allocation of behavioral elements to structural elements, or software elements to hardware elements). SysML does not enforce any modeling constraints on the construction or usage of the elements in its allocation diagrams. Therefore, it does not give the engineers feedback as they create or visualize deployment models of the system.

Proliferation of web-based service-oriented applications has resulted in the development of the Web Services Description Language (WSDL) [66], which is an XML format for describing network services as a set of endpoints operating on messages. While WSDL provides sophisticated language constructs for describing network protocol bindings and message formats required to interact with a web service, it supports neither modeling of a system's hardware architecture, nor does it support modeling software architectural connections and interdependencies among the software components.

Deployment architecture modeling in this research builds on the previous research in *Architecture Description Languages* (ADL) [35]. ADLs provide formal notations for describing and analyzing software systems at the software architecture level. While there are many examples of ADLs [35], none of them are capable of modeling a

system's deployment architecture. They support neither the modeling of hardware architectures nor the mapping of a system's software architecture to hardware platforms. Just like many ADLs (e.g., xADL [6]), DeSi, the deployment modeling tool produced by this dissertation research, provides an extensible approach to modeling a system's software architecture. However, unlike ADLs, DeSi also supports modeling of a system's hardware architecture.

3.2 Deployment Analysis

Several researchers have considered the impact of a system's deployment architecture on the non-functional properties of the system, including its QoS:

I5 [1], proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network, such that the overall remote communication is minimized. Solving the BIP model is exponentially complex in the number of software components, rendering I5 applicable only to systems with very small numbers of software components and target hosts. Furthermore, the approach is only applicable to the minimization of remote communication.

Coign [17] provides a framework for distributed partitioning of COM applications across the network. Coign monitors inter-component communication and then selects a distribution of the application that will minimize communication time, using the

lift-to-front minimum-cut graph cutting algorithm. However, Coign can only handle situations with two-machine client-server applications. Its authors recognize that the problem of distributing an application across three or more machines is NP hard and do not provide solutions for such cases.

Kichkaylo et al. [22] provide a model, called component placement problem (CPP), for describing a distributed system in terms of network and application properties and constraints, and an AI planning algorithm, called Sekitei, for solving the CPP model. The focus of CPP is to capture a number of different constraints that restrict the solution space of valid deployment architectures. At the same time, CPP does not provide facilities for specifying the goal, i.e., a criterion function that should be maximized or minimized. Therefore, Sekitei only searches for a valid deployment that satisfies the specified constraints, without considering the quality of the found deployment.

In our own prior work [29,40,42], we devised a set of algorithms for improving a software system's availability by finding an improved deployment architecture. The novelty of our approach was a set of approximative algorithms that scaled well to large distributed software systems with many components and hosts. However, our approach was limited to a predetermined set of system parameters, and a predetermined definition of availability.

None of the above approaches (including our own previous work) considers the system users and their QoS preferences. Furthermore, none of these approaches attempt to improve more than one QoS dimension of interest. Finally, no previous work has considered users' QoS preferences at the granularity of the application-level services. Instead, the entire distributed software system is treated as one service with one user, and a particular QoS dimension serves as the only QoS objective. Finally, the implementation and evaluation of all of the above solutions are done in an ad-hoc way, making it hard to adopt and reuse their results. This dissertation's tool-suite provides a common environment for developing solutions to this problem, which aids development, reuse, and cross evaluation of solutions to the many variations of this problem.

3.3 Software Deployment Support

A wide variety of technologies exist to support various aspects of the deployment process. Carzaniga et. al. [2] provide an extensive comparison of existing software deployment techniques. They identify three classes of software deployment technologies: Installers, Package Managers, and Application Management Systems. Examples of Installers are Microsoft Windows Installer [38] and InstallShield [19]. The primary focus of installers is to package a stand-alone software system into a self-installing archive that can be distributed via physical media or networks. Examples of Package Managers are Linux RedHat's RPM [10], and SUN Solaris's *pkg* commands [59]. These deployment technologies are based on the concept of

package, and on site repository that stores information representing the state of each installed package. A package is an archive that contains the files that constitute a system together with some meta-data describing the system. Examples of Application Management Systems are IBM Tivoli Composite Application Manager [18] and OpenView from HP [16]. Application Management Systems are usually composed of a centralized “producer”, which is a designated central administration site for all officially approved releases. Correspondingly, all the management and deployment activities are typically controlled by the central management station, which stores deployment meta-data of the whole system. Unlike all of the above deployment technologies, this dissertation’s framework supports system deployment in terms of its architectural components. This also facilitates dynamic modifications of the system, including redeployment of its components, at the architecture level.

3.4 Remote Deployment Monitoring

Before we can assess and improve a system’s deployment architecture, we need to understand the properties of the deployed system by monitoring it. Numerous previous works have focused on the problem of remote monitoring of a distributed system. In the context of our problem, they fall into two categories: (1) monitoring techniques that monitor at the granularity of software architectural constructs (e.g., components, connectors, their interfaces); and (2) monitoring systems that monitor at the granularity of system architectural constructs (e.g., hardware hosts, network links).

Prominent examples of the first category are MonDe [4], GAMMA[48], and COMPAS [44]. MonDe provides support for evaluating the performance of a new version of a component by (1) deploying it to remote sites, (2) running it in a controlled environment with the actual workloads being generated at that site, and (3) reporting the results back to the development engineers. GAMMA is a lightweight approach to monitoring deployed systems via software tomography, which uses probes that are optimally assigned to program instances in order to minimize the total overhead of monitoring. The information obtained from the probes is then aggregated into overall monitoring information. Similar to GAMMA, COMPAS provides a monitoring framework for adaptive instrumentation and diagnosis of the system. However, unlike GAMMA, COMPAS probes are automatically activated and deactivated based on runtime conditions.

Some prominent examples of the second category are JAMM [64] and Remos [8]. Both JAMM and Remos provide a monitoring infrastructure geared towards grid computing environments that can extract vital statistics such as CPU, network, and memory for the running nodes in the cluster. JAMM proposes an automated agent-based solution to monitoring system parameters. Remos presents a compositional approach to monitoring the system at an arbitrary level of detail and provides the grid applications a simple API for accessing the monitored data.

Our monitoring of a system's deployment architecture differs from previous works in this area, as we monitor both system properties and software architecture properties of the system. The aggregation of the monitored data then allows us to populate our deployment models with a complete view of the system, such that we can assess and analyze the properties of a system's deployment architecture.

3.5 Dynamic Reconfiguration of Deployment Architecture

(Re)Deployment is a process of installing, updating, and/or relocating a distributed software system. In a software architecture-based system, these activities fall under the larger category of *dynamic reconfiguration*, which encompasses run-time changes to a software system's architecture via addition and removal of components, connectors, or their interconnections. Oreizy et. al. [47] describe several aspects of dynamic reconfiguration, which determine the degree to which change can be reasoned about, specified, implemented, and governed. They also describe three causes of dynamic reconfiguration: (1) *corrective*, which is used to remove software faults, (2) *perfective*, used to enhance software functionality, and (3) *adaptive*, used to enact changes required for the software to execute in a new environment. Our approach is an instance of adaptive dynamic reconfiguration.

Garlan et. al. [12] propose a general purpose architecture-based adaptation framework, which monitors the software system and leverages ADLs in adapting and achieving architectural conformance. Their approach is different from our

framework, as they only model software architectural aspects of the system and not those of the hardware platforms, therefore, making it impossible to assess and reconfigure a system's deployment architecture.

Haas et. al. [14] provide a framework for autonomic service deployment in networks. The work resembles our approach in that the authors consider the scalability of their autonomic algorithms, which divide the network into partitions and perform a hierarchical deployment of network services. However, their approach is not applicable to application-level deployment.

Software Dock [15] is a system of loosely coupled, cooperating, distributed components. It supports software producers by providing a *Release Dock* and a *Field Dock*. The Release Dock acts as a repository of software system releases. The Field Dock supports a software consumer by providing an interface to the consumer's resources, configuration, and deployed software systems. Software Dock employs agents that travel from a Release Dock to a Field Dock in order to perform specific software deployment tasks. They perform four types of architectural reconfiguration: (1) component addition, (2) component removal, (3) component replacement, and (4) structural reconfiguration (i.e., recombination of existing functionality to modify overall system behavior). As we will see, the dynamic reconfiguration aspect of our approach closely resembles Software Dock's approach. However, to use Software Dock one needs to first install and configure it on each device, but in our approach we

leverage the same infrastructure used for implementation and deployment of components as we do for their reconfiguration at runtime, thus, providing a more consistent and efficient solution.

CHAPTER 4: Overview of the Framework

We have developed a methodology for improving the quality of a system's deployment architecture via (1) active system monitoring, (2) estimation of the improved deployment architecture, and (3) redeployment of (parts of) the system to effect the improved deployment architecture [30]. Based on this three-step methodology we have identified the high-level components of our deployment improvement framework [30], which are shown in Figure 4-1. In this section we describe the framework's components, the associated functionality of each component, and the dependency relationships that guide their interaction. We also describe the variation points for each component (shown in Figure 4-2).

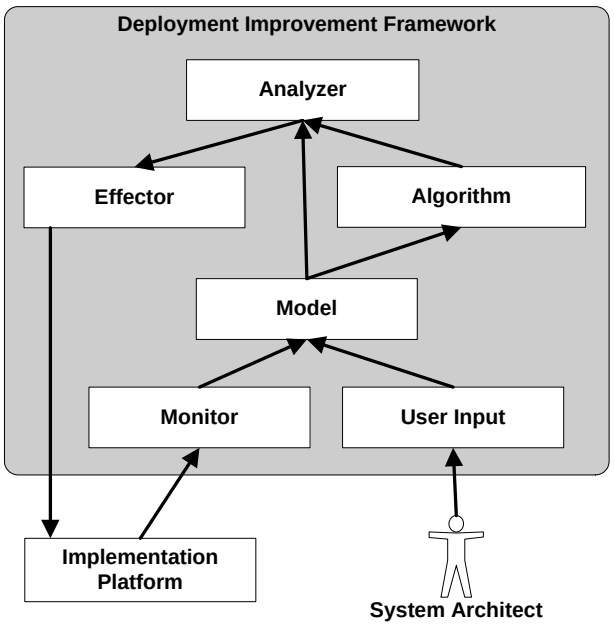


Figure 4-1: Deployment improvement framework.

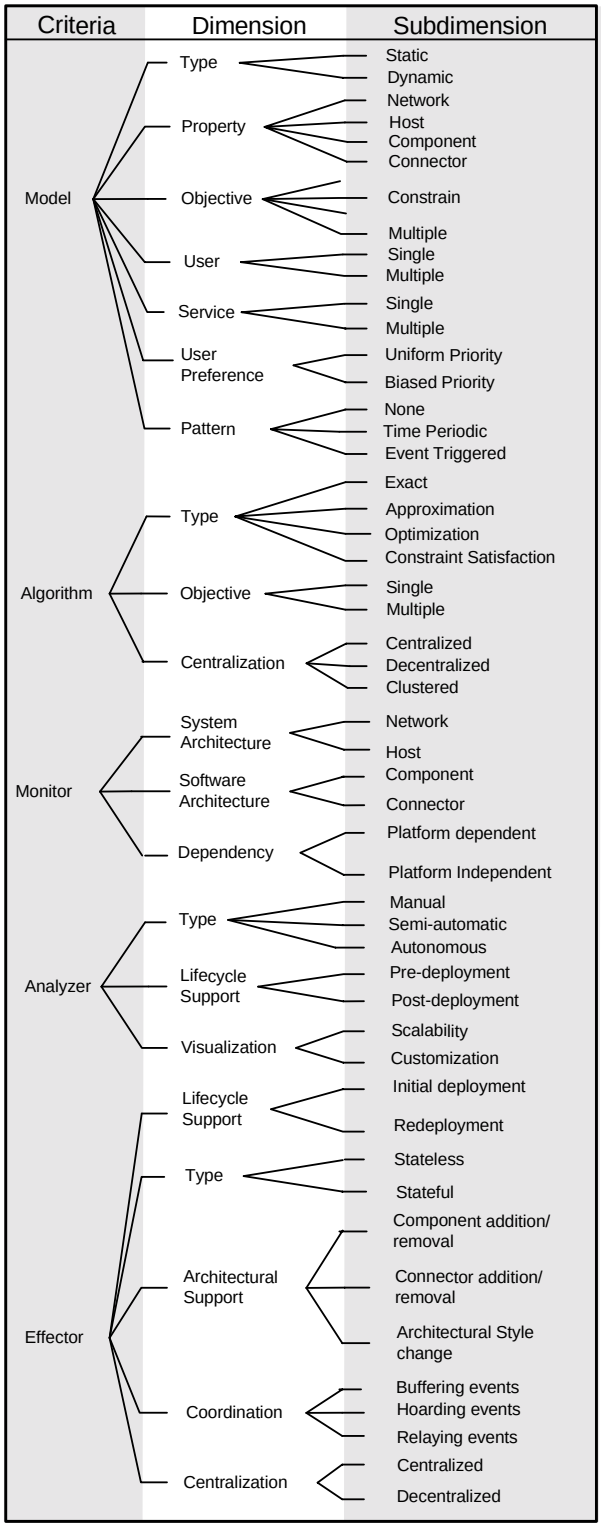


Figure 4-2: Framework variation points.

4.1 Model

Before the actual deployment of the system, engineers may need to create representative deployment models of the system. Deployment models typically incorporate or build on existing design or architectural models of the system. These models are critical as they determine the types of analysis that can be performed both at pre- and post-deployment stages. Therefore, they require the system architect to determine the kinds of analysis that are desirable to be performed in the future, and set up the model and its properties appropriately.

A deployment model in its most basic form is composed of four types of elements: hosts, components, physical links (network links) between hosts, and logical links (connectors) between components. Each of these element types could be associated with an arbitrary set of parameters. For example, each host can be characterized by the amount of available memory, processing speed, battery power (in case a mobile device is used), installed software, and so on. Therefore, the framework should allow for the specification of arbitrary system parameters and associate them with the modeling elements. The model should also support the specification of a set of objectives (QoS) based on system parameters that are defined. Therefore, the selection of a set of parameters to be modeled depends on the set of objectives that a system's deployment architecture should satisfy. For example, if minimizing latency is one of the objectives, the model should include parameters such as physical network link delays and bandwidth. However, if the objective is to improve a

distributed system's security, other parameters, such as security of each network link, need to be modeled.

While design-time deployment models of a system are typically static, runtime models are dynamic, and include the notion of time. Dynamic models make it possible to determine patterns of both system parameter fluctuation (e.g., periods of time that the bandwidth is low) and its usage (e.g., an event that when triggered results in a heavy load for a period of time). Some patterns may be known at design time. For example, in an aerospace satellite system, the engineers can predict the available bandwidth between ground station and the satellite based on its orbital position. Other patterns are not known at design time. For example, many commercial websites have a usage pattern where for a few hours during the day they receive more hits (heavier load) on their sites; these types of patterns are discovered at runtime. Both types of patterns are essential to assessing and improving a system's deployment architecture and this dissertation's framework should allow for their specification.

In systems where there are multiple users, the model should also include the ability to specify users and their unique preferences with respect to the defined objectives. For example, while one user may want to improve the system's availability, another user may want to improve its security. Some users' preferences may have priority over others; for example in a battlefield environment, the commander's preferences for improving a particular QoS may have higher priority over a soldier's preferences. The

framework should allow for specifying multiple users with different QoS preferences and different priorities.

Typically, a large distributed system provides many different services. The users of the system may have varying QoS preferences for each service. In order to determine the best deployment of the system, both the provisioned services and their importance in satisfying the users' QoS preferences should also be considered. Therefore, the model should provide the ability to specify a service and its mapping to a portion of the system model that provides that service (e.g., components and hosts that are involved).

4.2 Algorithm

The types of algorithms required depend on the application scenario and the objective properties of interest. An objective can be formally specified either as an optimization problem (e.g., maximize availability, minimize latency) or a constraint satisfaction problem (e.g., total memory of components deployed onto a host cannot exceed that host's available memory). Given an objective and the relevant subset of the system's model, an algorithm searches for a deployment architecture that satisfies the objective. An algorithm may also search for a deployment architecture that simultaneously satisfies multiple objectives (e.g., maximize availability while satisfying the memory constraints).

In terms of precision and computational complexity, there are four categories of algorithms for this problem: exact, approximation, optimization, and constraint satisfaction. Exact algorithms produce optimal results (e.g., deployments with minimal overall latency), but are exponentially complex, which limits their applicability to systems with very small numbers of components and hosts. On the other hand, both approximation and optimization algorithms in general produce sub-optimal solutions, but have polynomial time complexity, which makes them run much faster than exact algorithms. Unlike an optimization algorithm, an approximation algorithm can provide an error bound between its output and the optimal solution. Given the unbounded nature of most deployment improvement problems[1], most sub-optimal algorithmic solutions are of the optimization type.

In terms of centralization, there are three classes of algorithms: centralized, decentralized, and clustered. Centralized algorithms execute on a single host with the global knowledge of the system. Decentralized algorithms execute on multiple synchronized hosts with local knowledge of the system. Clustered algorithms leverage a centralized algorithm to determine the best deployment within each cluster of hosts (i.e., virtual hosts), and leverage a decentralized algorithm to determine the best deployment among the clusters.

1. Most deployment improvement problems are NP Hard. Therefore, finding the optimal solution is computationally very expensive.

As mentioned earlier, at the pre-deployment stage, the lack of knowledge about system parameters that fluctuate at runtime makes it impossible to determine the best deployment architecture for a system. However, there is still the possibility of improving the system's initial deployment based on the limited information that is available. Off-line algorithms are the types of algorithms that are executed at design time. Given that there are significantly more time and processing resources available for producing results at this stage, the engineers tend to use more complex and accurate algorithms to produce the best possible result given the limited amount of information. Furthermore, given that off-line algorithms are usually executed on design time models of the system available centrally on the engineer's production platform, they tend to be centralized. On the contrary, at post-deployment time, the algorithms have access to runtime knowledge of system parameters, but there are typically limited amounts of time and resources available. In fact, often it is preferable to have algorithms that produce a "good enough" solution that improves system's QoS significantly as opposed to highly complex algorithms that produce the optimal solution too late or at the expense of valuable resources. Therefore, on-line algorithms have to be more efficient than off-line algorithms.

4.3 Monitor

Many parameters that are essential to assessing and improving a system's deployment architecture are not known until the system is actually deployed in the field. There are two classes of parameters that need to be monitored: system parameters and software

architecture parameters. System parameters are either network or platform related properties. Software parameters are either component (e.g., memory or processing resources used) or connector (e.g., number of exchanged events) related properties. To determine the run-time values of the parameters in the model, a monitoring facility is associated with each monitored entity. Typically, monitoring consists of two parts: 1) *probe*, which is the platform-dependent part that "hooks" into the implementation platform and performs the actual low-level monitoring of the system; and 2) *gauge*, which is the platform-independent part that interprets and may look for patterns in the monitored data. For example, a gauge determines if the data is stable enough [42] to be passed on to the model.

4.4 Analyzer

Assessing non-functional properties of a large distributed system is a challenging task. In fact, even in the presence of algorithms that can be leveraged to improve one or more non-functional properties of the system, ensuring the solution produced by the algorithm satisfies the objective of the users is a challenging task. Assessment can be performed: 1) manually by one or more engineers, 2) semi-manually, where an analyzer agent is leveraged to aid the engineers, or 3) autonomously, where analyzer agents make decisions on behalf of the engineers.

Analyzers are meta-level algorithms that leverage the results obtained from the (re)deployment algorithm(s) and the monitoring data to determine a course of action

for satisfying the system's overall objective. In situations where several objective functions need to be satisfied, an analyzer resolves the results from the corresponding algorithms to determine the best deployment architecture. An autonomous analyzer may also have a reflective capability that would allow it to make changes to the deployment improvement framework itself. For example, once an analyzer determines that the system's parameters have changed significantly, it may choose to leverage a new algorithm that computes better results for the new operational scenario. Analyzers may also hold the history of the system's execution by logging fluctuations of the desired objectives and the parameters of interest. A system's execution profile allows the analyzer to fine-tune the framework's behavior by providing information such as the system's stability, workload patterns, and the results of previous redeployments.

The types of assessment done at pre-deployment are typically different from those of post-deployment. The lack of time and unpredictable fluctuations in system parameters make the autonomic type of analysis and assessment a more feasible solution during the post-deployment stage. On the contrary, the benefit of leveraging the engineer's domain knowledge makes (semi-)manual assessment of the system a more desirable solution at pre-deployment.

4.5 Effector

There are two high-level aspects of effecting a new deployment architecture: (re)deployment and dynamic reconfiguration. Below we discuss these two in more detail.

At a very high level software deployment is the process of installing a software system. Historically references to software deployment in the literature have implied the initial installation of a software system, which can be further broken down to subtasks such as software download, configuration, execution, etc. However, with the development of code mobility technologies over the last decade, the software deployment life-cycle does not end with the initial deployment of a system, as software may be redeployed. In terms of the underlying infrastructure that supports deployment of software components, there are significant differences between the initial versus subsequent deployments of a system. For example, during redeployment it may be necessary to migrate the execution state of the component, while during initial deployment it may be necessary to transfer the input parameters for the initialization of the component. Given the low-level nature of software deployment this aspect of the framework is typically platform-dependent and “hooks” into the platform to perform the redeployment of software components.

Dynamic reconfiguration in the context of our problem is the facility that manages a system’s redeployment and architectural adaptation process. The dynamic

reconfiguration facility leverages a framework's deployment support to migrate software components. It may leverage the underlying implementation platform to modify the system's software architecture, such that components and connectors can be added and removed at runtime. When changing a system's deployment architecture, the dynamic reconfiguration facility may ensure that the system's software architectural constraints are preserved. In some cases, it may be necessary to change the architectural style of a system as a result of redeploying its software components. For example, consider the scenario where the local architecture of a host adheres to the layered hierarchical architectural style (e.g., C2 [63]), where events are broadcasted from components in one layer to those in the next layer of the hierarchy. If we were to redeploy components from this host to multiple distributed devices, it may be necessary to also change the system's architectural style (e.g., to client-server or publish-subscribe) to avoid wasting bandwidth due to the broadcast of events over the network.

Our framework's dynamic reconfiguration facility may also need to ensure the system's correct functioning throughout the redeployment process. It may use techniques such as: 1) buffering events addressed to a component that is being redeployed, 2) hoarding events and servicing the requests before the component is redeployed, and 3) relaying events addressed to a component from its previously deployed location to its new location.

The dynamic reconfiguration facility is leveraged once a new deployment solution is selected. It is responsible for developing a plan that allows for the safe redeployment of the system. For example, the plan may indicate the order of deployment, the instantiation of “dummy” components for relaying the events, etc. In fact, the generated plans depend on the type of the system. In a centralized system, plans are generated and executed by a single agent. On the other hand, in a decentralized system, plans are generated and executed in collaboration by multiple agents.

4.6 User Input

For the engineers to be able to understand and assess a system’s deployment architecture, it is necessary to visualize the relevant aspects of the system’s deployment model. The framework should provide a highly customizable and extensible visualization environment that allows for displaying the many variation points and properties of the model. On top of this, the framework’s visualization should scale to large distributed systems, and allow the engineers to visually explore and assess the properties hidden within the underlying models of the system.

Some system parameters may not be easily monitored (e.g., security of a network link). Also, some parameters may be stable throughout the system’s execution (e.g., CPU speed on a given host). The values for such parameters are provided by the system’s architect at design time. We are assuming that the architect is able to provide a reasonable bound on the values of system parameters that cannot easily be

monitored. Furthermore, the architects typically provide constraints on the allowable deployment architectures. Examples of these types of constraints are location and collocation constraints. Location constraints specify a subset of hosts on which a given component may be legally deployed. Collocation constraints specify a subset of components that either must be or may not be deployed on the same host.

4.7 Framework Instantiation

Figure 4-3 shows the framework's instantiation for a centralized system. Centralized systems have a *Master Host* (i.e., central host) that has complete knowledge of the distributed system parameters. *Master Host* contains a *Centralized Model*, which

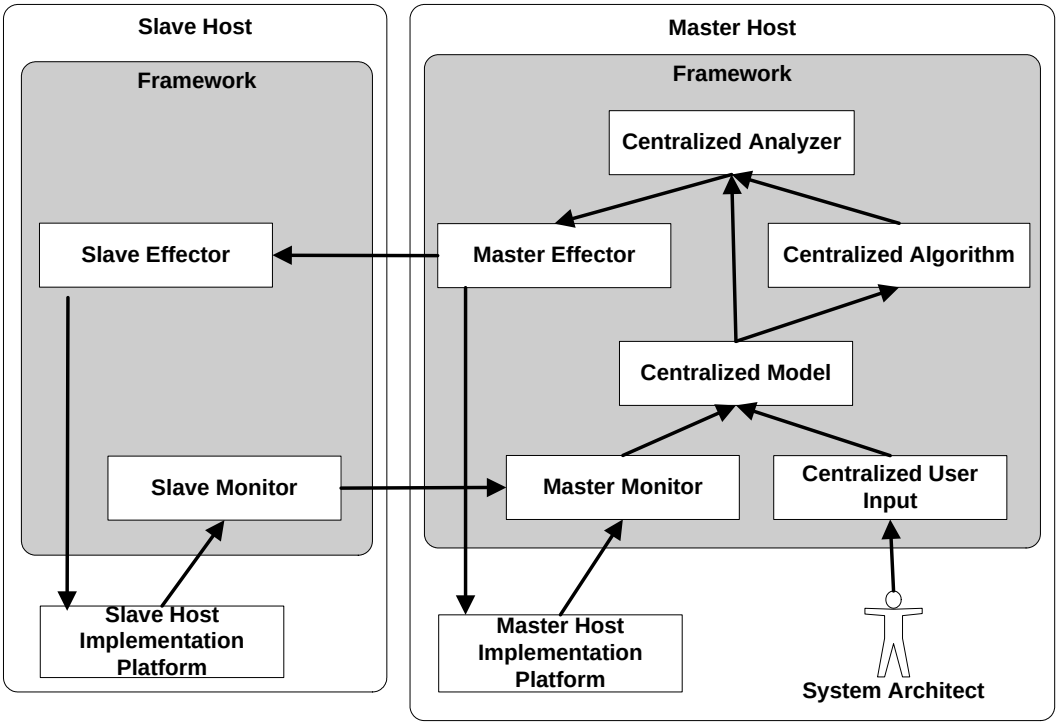


Figure 4-3: Framework's centralized instantiation.

maintains the global model of the distributed system. The *Centralized Model* is populated by the data it receives from *Master Monitor* and *Centralized User Input.* The *Master Monitor* receives all of the monitoring data from the *Slave Monitor*s on other hosts. Once all monitoring data from all *Slave Host*s is received, the *Master Monitor* forwards the monitoring data to the *Centralized Model*. Each *Slave Host* contains a *Slave Effector,* which receives redeployment instructions from the *Master Effector,* and a *Slave Monitor,* which monitors the *Slave Host*'s *Implementation Platform* and sends the monitoring data back to the *Master Monitor.* Finally, the *Master Effector* receives a sequence of command instructions from the *Centralized Analyzer* and distributes the redeployment commands to all the *Slave Effector*s.

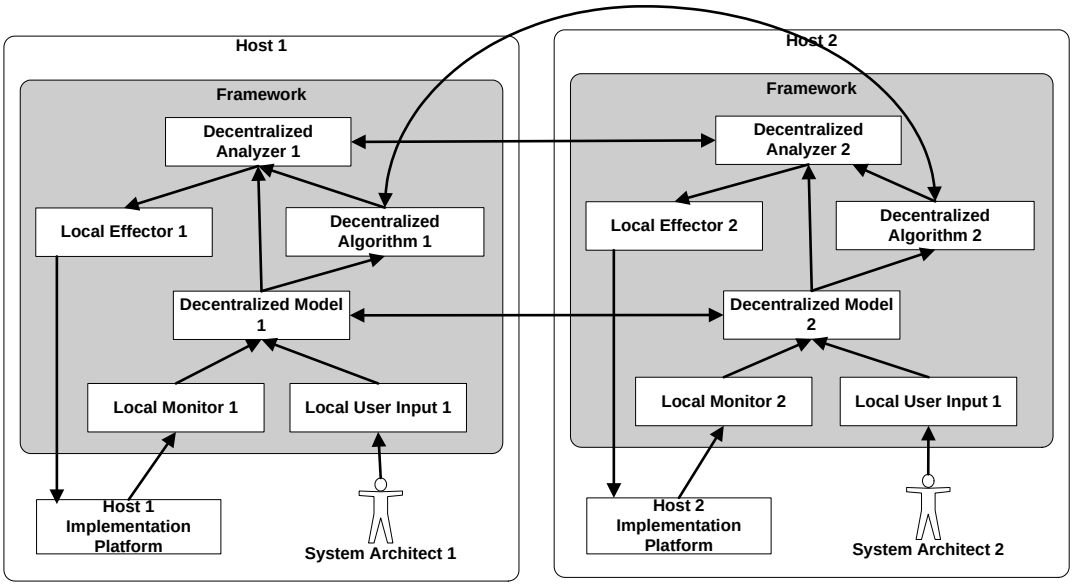


Figure 4-4: Framework's decentralized instantiation.

Figure 4-4 shows the framework's instantiation for a decentralized system. Unlike a centralized software system, a decentralized system does not have a single host with the global knowledge of system parameters. Each host has a *Local Monitor* and a *Local Effector,* which are only responsible for the monitoring and redeployment of the components that are located on that host. Each host has a *Decentralized Model* that contains some subset of the overall system's model, populated by the data received from the *Local Monitor* and the *Decentralized Model* of the hosts to which this host is connected. Therefore, if there are two hosts in the system that are not aware of (i.e., connected to) each other, then the respective models maintained by the two hosts do not contain each other's system parameters. Each host also has a *Decentralized Algorithm* that synchronizes with its remote counterparts to find a common solution. Finally, in a similar way, the *Decentralized Analyzer* on each host synchronizes with its remote counterparts to determine an improved deployment architecture and effect it.

CHAPTER 5: Framework Model

Before we can develop algorithms to improve system's deployment architecture, we need to be able to define the problem more precisely. In this chapter, we first present how an application scenario can be modeled using mathematical notation, and then leverage this foundation to formally define the deployment improvement problem.

5.1 Formal Problem Definition

As mentioned earlier, one of our primary objectives in the design of the framework is to make it practical: enable it to capture realistic distributed system scenarios and avoid making assumptions that will restrict its applicability. For example, we want to avoid prescribing a predefined number of system parameters, or particular definitions of QoS dimensions. We want the framework to provide the minimum skeleton structure required to model a distributed system's deployment and the system parameters that affect that deployment. Each skeleton element can be extended and arbitrarily refined by the system architect. Figure 5-1 shows the framework's formal model of a distributed software system's deployment architecture:

1. A set *H* of hardware nodes (hosts) with the associated parameters (e.g., available memory or CPU on a host), and a function *hParam* that maps each parameter to a value.

2. A set *C* of components with the associated parameters (e.g., required memory for

a component's execution or JVM version), and a function *cParam* that maps each parameter to a value.

3. A set *N* of physical network links with the associated parameters (e.g., available bandwidth, reliability of links), and a function *nParam* that maps each parameter to a value.

4. A set *I* of logical interaction links between software components in the distributed system, with the associated parameters (e.g., frequency of component interactions, average event size), and a function *iParam* that maps each parameter to a value.

5. A set *S* of services, and a function *sParam* that provides values for service-specific system parameters. An example service-specific system parameter is the number of component interactions resulting from an invocation of a particular service.

6. A set *DepSpace* of all possible deployment mappings.

7. A set *Q* of QoS dimensions, and a function *qValue* that quantifies a QoS dimension (e.g., security) for a given service (e.g., "find the best route to the disaster area") in the current deployment mapping. Also, a function *qType* that represents the minimization or maximization aspect of the QoS dimension.

8. A set *U* of users, and two complementary functions *qosRate* and *qosUtil* that denote a user's preference for a QoS dimension of a service. *qosRate* returns the

rate of change, while *qosUtil* returns the utility for that rate of change. For example, the user may denote a utility of 0.1 for a change of 0.2 (i.e., 20%) in a particular QoS dimension of a service. Relative importance of different users is determined by two threshold values: *MinRate* and *MaxUtil*. *MinRate* denotes the minimum rate of change a user is allowed to specify, while *MaxUtil* denotes the maximum utility. In general, a smaller value of *MinRate* and a larger value of *MaxUtil* indicates higher importance (or relative influence) of the user in the final solution.

9. A set *PC* of parameter constraints, and a function *pcSatisfied* that, given a constraint and a deployment architecture, returns 1 if the constraint is satisfied and 0 otherwise. For example, if the parameter constraint is “bandwidth satisfaction”, the corresponding constraint function may ensure that the total volume of data exchanged across any network link does not exceed that link’s bandwidth in a given deployment architecture.

10. Using the *loc* function, deployment of any component can be restricted to a subset of hosts, thus denoting a set of allowed hosts for that component. Using the *colloc* function, constraints on allowed collocations of components can be specified.

Note that some elements of the framework model are intentionally left “loosely defined” (e.g., system parameter sets, QoS set). These elements correspond to the many and varying factors that are found in different distributed application scenarios.

As we will see in Section 5.2, when the framework is instantiated, the system architect specifies these loosely defined elements.

For brevity we use the following shorthand notations in the remainder of this document: H_c is a host on which component c is deployed; $I_{c1,c2}$ is an interaction

A distributed system is modeled in terms of

1. a set H of hardware nodes, a set HP of host parameters, a function $hParam: H \times HP \rightarrow \Re$
2. a set C of components, a set CP of component parameters, a function $cParam: C \times CP \rightarrow \Re$
3. a set N of network links, a set NP of network link parameters, a function $nParam: N \times NP \rightarrow \Re$
4. a set I of logical links (interactions), a set IP of logical link parameters, a function $iParam: I \times IP \rightarrow \Re$
5. a set S of services, and a function $sParam: S \times \{H \cup C \cup N \cup I\} \times \{HP \cup CP \cup NP \cup IP\} \rightarrow \Re$ of values for service-specific system parameters
6. a set $DepSpace = \{d_1, d_2, \ldots\}$ of all possible deployment mappings, where $|DepSpace| = |H|^{|C|}$
7. a set Q of quality of services, a function $qValue: S \times Q \times DepSpace \rightarrow \Re$ that quantifies the achieved level of QoS, and
$$qType: Q \rightarrow \begin{cases} -1 & \text{if it is desirable to minimize this QoS} \\ 1 & \text{if it is desirable to maximize this QoS} \end{cases}$$
8. a set U of users, a function $qosRate: U \times S \times Q \rightarrow [MinRate, 1]$ representing the rate of change in a QoS, and a complementary function $qosUtil: U \times S \times Q \rightarrow [0, MaxUtil]$ representing the utility for that rate of change
9. a set PC of parameter constraints, and a function
$$pcSatisfied: PC \times DepSpace \rightarrow \begin{cases} 1 & \text{if } constr \text{ is satisfied} \\ 0 & \text{if } constr \text{ is not satisfied} \end{cases}$$
10. two functions that restrict locations of software components
$$loc: C \times H \rightarrow \begin{cases} 1 & \text{if } c \in C \text{ can be deployed onto } h \in H \\ 0 & \text{if } c \in C \text{ cannot be deployed onto } h \in H \end{cases}$$
$$colloc: C \times C \rightarrow \begin{cases} 1 & \text{if } c1 \in C \text{ has to be on the same host as } c2 \in C \\ -1 & \text{if } c1 \in C \text{ cannot be on the same host as } c2 \in C \\ 0 & \text{if there are no restrictions} \end{cases}$$

Figure 5-1: Framework model.

between components *c1* and *c2*; $N_{h1,h2}$ is a network link between hosts *h1* and *h2*; finally, C_s is a set of components that constitute service *s*.

Figure 5-2 shows the formal definition of the problem based on the framework model. The function *overallUtil* represents the overall satisfaction of the users with the QoS delivered by the services they use. The goal is to find a (new) deployment architecture that maximizes *overallUtil* and meets the constraints on location, collocation, and system parameters (items 9 and 10 in Figure 5-1).

5.2 Framework Instantiation

As the reader may recall, the formal specification of our problem (shown in Figure 5-1) was intentionally left loosely defined. However, to describe the algorithms' logic, we need to be able to precisely specify an application scenario, including its loosely

Given the current deployment of the system $d \in DepSpace$, find an improved deployment d' such that the users' overall utility defined as the function

$$overallUtil(d,d') = \sum_{u=1}^{|U|}\sum_{s=1}^{|S|}\sum_{q=1}^{|Q|}\left(\frac{\left(\frac{qValue(s,q,d') - qValue(s,q,d)}{qValue(s,q,d)}\right)}{qosRate(u,s,q)}\right) * qosUtil(u,s,q) * qType(q)$$

is maximized, and the following conditions are satisfied:

1. $\forall c \in C \qquad loc(c, H_c) = 1$
2. $\forall c1 \in C \quad \forall c2 \in C \qquad if \quad (colloc(c1,c2) = 1) \Rightarrow (H_{c1} = H_{c2})$
   $if \quad (colloc(c1,c2) = -1) \Rightarrow (H_{c1} \neq H_{c_i})$
3. $\forall constr \in PC \qquad pcSatisfied(constr, d) = 1$

In the most general case, the number of possible deployment architectures is $|DepSpace| = |H|^{|C|}$. However, note that some of these deployments may not satisfy one or more of the above three conditions.

Figure 5-2: Problem definition.

defined parts. Framework instantiation is the process of configuring the framework model for an application scenario. We illustrate this using four QoS dimensions: availability, latency, communication security, and energy consumption. Note that any arbitrary set of QoS dimensions can be used in our framework. Also note that the framework does not place any restrictions on the manner in which QoS dimensions are defined and quantified. This allows an engineer to tailor the framework to her specific needs. Below we give sample quantifications of these selected four QoS dimensions in the context of distributed systems.

The first step in instantiating the framework is to define the relevant system parameters. Item 1 of Figure 5-3 shows a list of parameters that we have identified to be of interest for specifying the four QoS dimensions. We should note that additional parameters may be found to be relevant as well. Those parameters can be similarly instantiated in our framework. Once the parameters of interest are specified, the parameter realization functions (e.g., *hParam*, *cParam* of Figure 5-1) need to be defined. These discrete functions can be defined in many ways: monitoring the system, relying on system engineer's knowledge, extracting from the architectural description, etc.

A software system's *availability* is commonly defined as the degree to which the system is operational when required for use [20]. Availability is the QoS dimension denoting whether a service is present or ready for immediate use. In the context of

distributed environments, where a most common failure is a network failure, we quantify availability as a function of successfully completed inter-component interactions in the system [40,42]. Item 2 of Figure 5-3 defines availability for a single service *s* in a given deployment *d*. A software service's *latency* is commonly defined as the time elapsed between making a request for service and receiving the

1. System parameters

$hostMem \in HP$ available memory on a host $hostEnrCons \in HP$ average energy consumption per opcode $compMem \in CP$ required memory for a component $opcodeSize \in CP$ average amount of computation per event $freq \in IP$ frequency of interaction between two components $evtSize \in IP$ average event size exchanged between two components $bw \in NP$ available bandwidth on a network link	$rel \in NP$ reliability of a network link $td \in NP$ transmission delay of a network link $enc \in NP$ encryption capability of a network link $commEnrCons \in NP$ energy consumption of transmitting data $availability, latency, security, energy \in Q$ four QoS dimensions $memConst \in PC$ constraint on host's available memory parameter

2. Availability: $qValue(s, availability, d) = \sum_{c1=1}^{C_s} \sum_{c2=1}^{C_s} sParam(s, I_{c1,c2}, freq) * nParam(N_{H_{c1},H_{c2}}, rel)$

3. Latency: $qValue(s, latency, d) = \sum_{c1=1}^{C_s} \sum_{c2=1}^{C_s} sParam(s, I_{c1,c2}, freq) * nParam(N_{H_{c1},H_{c2}}, td) + \frac{sParam(s, I_{c1,c2}, freq) * sParam(s, I_{c1,c2}, evtSize)}{nParam(N_{H_{c1},H_{c2}}, bw) * nParam(N_{H_{c1},H_{c2}}, rel)}$

4. Communication security:

$qValue(s, \text{security}, d) = \sum_{c1=1}^{C_s} \sum_{c2=1}^{C_s} sParam(s, I_{c1,c2}, freq) * sParam(s, I_{c1,c2}, evtSize) * nParam(N_{H_{c1},H_{c2}}, enc)$

5. Energy consumption:

$$qValue(s, energy\ d) = \sum_{c1=1}^{C_s} \sum_{c2=1}^{C_s} \left(\frac{sParam(s, I_{c1,c2}, freq) * sParam(s, I_{c1,c2}, evtSize)}{nParam(N_{H_{c1},H_{c2}}, commEnrCons)} + cParam(s, I_{c1,c2}, freq) * \left(\frac{cParam(c1, opcodeSize)}{hParam(H_{c1}, hostEnrCons)} + \frac{cParam(c2, opcodeSize)}{hParam(H_{c2}, hostEnrCons)} \right) \right)$$

6. Memory constraint:

if $\forall h \in H \left\{ \forall c \in C \quad H_c = h \;\middle|\; \sum cParam(c, compMem) \le hParam(h, hostMem) \right\}$,

then $pcSatisfied(memConst, d) = 1$

else $pcSatisfied(memConst,\ d)\ =\ 0$

Figure 5-3: Framework instantiation example.

response [20]. The most common causes of communication delay in a distributed system are the unreliability of network links, low bandwidth, and the network transmission delay. Item 3 of Figure 5-3 defines latency for a service *s*. Note that for simplicity in our specification of latency we did not consider the computational delay associated with each software component's execution; however, it should be evident that the framework does not prevent one from including a more elaborate and accurate quantification of latency. A major factor in the *security* of distributed systems is the level of encryption (e.g., 128-bit versus 40-bit encryption) capability provided in remote communication [58]. Item 4 of Figure 5-3 defines communication security for a service *s*. Finally, *energy consumption* (or battery usage) of each service is determined by the energy required for the transmission of data among hosts plus the energy required for the execution of application logic in each software component for the service. Item 5 of Figure 5-3 defines energy consumption of a service *s*. For ease of exposition in this document we have provided a simplified definition; a more sophisticated energy consumption model can be found in our recent work [55].

As mentioned above, the definitions of the four QoS dimensions in Figure 5-3 are intended to serve primarily as an example of how QoS dimensions are quantified and used in our framework. A more detailed explanation of these QoS dimensions is beyond the scope of this research. We do not argue that these are the only, "correct", or even most appropriate definitions for these four dimensions. In fact, our framework

can accommodate arbitrary definitions of these as well as other QoS dimensions, so long as they are quantifiable.

For illustrating parameter constraints we use the memory available on each host. The constraint in item 6 of Figure 5-3 specifies that the total size of the components deployed on each host may not be greater than the total available memory on that host. Other constraints, such as “bandwidth satisfaction”, are included in the same manner.

We also need to populate the set *S* with the services and the set *U* with the users of the system (recall Figure 5-1). Finally, the users’ preferences are determined by defining the two functions *qosRate* and *qosUtil*. The users define the values these two functions take based on their preferences.

As discussed in Chapter 2, the greatest difficulty in solving the above problem is that it is an exponentially complex problem. Therefore, finding the optimal solution is infeasible for many large systems. Given that typically there is a short amount of time available for finding an improved deployment architecture at runtime, it is often preferable to find a solution that comes close to the optimal solution in a fraction of the time. Furthermore, as will be described in more detail in Chapter 8, some solutions are more appropriate for solving a given class of application scenarios than others (e.g., centralized versus decentralized systems).

5.3 Decentralization Modeling Constructs

Figure 5-4 defines several additional model elements needed for describing a decentralized application scenario. The relation *dep* denotes the current deployment of the system's components on hosts. The function *aware* and the relation *dom* model the system's decentralized nature. Function *aware* denotes whether two hosts have access to each other's properties and the properties of components that reside on them. Relation *dom* denotes the "domain" of a host h_i, which is the set of all hosts of which h_i is aware. A host's domain corresponds to the host's extent of knowledge about the overall system's parameters. For example, in the centralized application scenarios discussed above, the assumption is that at least one host's domain is the entire set of hosts *H*.

1. A relation $dep: H \rightarrow P(C)$ where $c_k \in dep(h_i)$ iff c_k is deployed on h_i
2. A function $aware: H \times H \rightarrow \{0,1\}$

$$aware(h_i, h_j) = \begin{cases} 1 & \text{if } h_i \text{ and } h_j \text{ have complete knowledge of each other's} \\ & \text{system properties and constraints} \\ 0 & \text{if } h_i \text{ and } h_j \text{ have no information of each other} \end{cases}$$

3. A relation $dom: H \rightarrow P(H)$, where $h_k \in dom(h_i)$ iff $aware(h_k, h_i) = 1$

Figure 5-4: Decentralization modeling constructs.

CHAPTER 6: Framework Algorithms

As mentioned earlier, this dissertation's research problem is an instance of multi-dimensional optimization problems, characterized by many QoS dimensions, system users and user preferences, and constraints that influence the objective function. Our objective has been to devise reusable algorithms that provide highly accurate results regardless of the application scenario. An in-depth study of general strategies applicable to this problem resulted in five algorithms for solving the described problem, where each algorithm is suitable to a particular class of systems as will be further discussed in Chapter 8. Unlike previous works that depend on the knowledge of specific system parameters, we have developed a number of novel heuristics for improving the performance and accuracy of our algorithms independently of system parameters. Therefore, regardless of the specific application scenario the system architect simply executes the algorithm most suitable for the system (e.g., based on the size of the system, or stability of system parameters) without any modification.

Of the five general approaches we have adopted and adapted, two (Mixed-Integer Nonlinear and Linear Programming, a.k.a. MINLP and MIP [67]) are best characterized as generic techniques developed in operations research to deal with multi-dimensional optimization problems. These techniques are accompanied by widely used algorithms and solvers. We tailor these techniques to target them specifically at our problem as defined in Figures 5-1 and 5-2, and thereby improve their results. The

remaining three approaches (greedy, genetic, and market-based) can be characterized as generally applicable strategies, which we have employed in developing specific algorithms tailored to our problem. In this chapter, we provide a discussion of all five techniques, with an analysis of their algorithmic complexity.

6.1 Mixed-Integer Nonlinear Programming (MINLP)

A linear (or non-linear) programming problem consists of three parts: decision variables, constraint functions, and an objective function. A linear programming problem has a linear objective function, while a non-linear programming problem has a non-linear objective function. In a mixed-integer programming problem the decision variables can only take on integer values within a specified domain. We will discuss two types of optimization techniques: in this section we discuss the MINLP approach and in the next section the MIP approach [67]. As we will see, since our objective function is a non-linear function, our problem is by default a MINLP problem. While we cannot solve the MINLP representation of our problem optimally, off-the-shelf MINLP solvers can approximate a solution most of the time. In Chapter 8, we compare the solution of our three approximative algorithms (introduced below) against the solutions produced by these MINLP solvers. Furthermore, in the next section we demonstrate a technique for converting a MINLP problem into an MIP problem, which can be solved optimally.

The first step in representing our problem as a MINLP problem is defining the decision variables. We define decision variable $x_{c,h}$, which corresponds to the decision of whether component *c* is to be deployed on host *h* or not. Therefore, we need |*C*|*|*H*| binary decision variables, where $x_{c,h}=1$ if component *c* is deployed on host *h*, and $x_{c,h}=0$ if *c* is not deployed on *h*.

The next step is defining the constraints, which includes the representation of both parameter and locational constraints in MINLP. For example, the memory constraint for a host *h* (Item 6 of Figure 5-3) is specified in this way:

$$\sum_{c=1}^{|C|} x_{c,h} * cParam(c, compMem) \leq hParam(h, hostMem)$$

Other parameter constraints are represented similarly. The fact that a software component *c* can only be deployed on one host is another constraint that needs to be specified: $\sum_{h=1}^{|H|} x_{c,h} = 1$. Other locational constraints are represented similarly.

Finally, we need to define the objective function. Below the elided MINLP representation of the objective function for the scenario introduced in Section 5.2 is shown,

where for brevity we are only showing availability and its contribution to the objective function; other QoS dimensions are included very similarly.

$$availValue(s) = \sum_{h1}^{|H|}\sum_{h2}^{|H|}\sum_{c1}^{|C|}\sum_{c2}^{|C|} sParam(s, I_{c1,c2}, freq) * nParam(N_{h1,h2}, rel) * x_{c1,h1} * x_{c2,h2}$$

$$availUtil(u,s) = (\frac{availValue(s) - initAvail(s)}{initAvail(s)}) / qosRate(u,s,avail) * qosUtil(u,s,avail)$$

//other QoS dimensions Value and Util functions ...

$$Maximize \quad overallUtil = \sum_{u=1}^{|U|}\sum_{s=1}^{|S|}(availUtil(u,s) + latenUtil(u,s) + \sec urUtil(u,s) + energyUtil(u,s))$$

The function *availValue* corresponds to the *qValue* function defined in item 2 of Figure 5-3. The function *availUtil* calculates a user's utility for the availability of a service. *initAvail* is a constant which represents the availability of the service for the initial deployment of the system. Note that as a result of decision variable multiplication ($x_{c1,h1}$*$x_{c2,h2}$), the objective function is not linear.

As mentioned earlier, while there are approximative algorithms for estimating a solution to a MINLP problem, there is no known algorithm for solving it optimally. Furthermore, for problems with non-convex functions (such as ours), MINLP solvers are not guaranteed to find and converge to an improved approximate solution [67]. Finally, given the non-standard techniques for solving MINLP problems, it is hard to determine a complexity bound for these MINLP solvers.[2] We will provide an empirical evaluation and comparison of MINLP solvers with other algorithms in Chapter 8.

2. All state-of-the-art MINLP solvers are based on confidential algorithms. Most vendors claim that their solvers run in polynomial time [5].

6.2 Mixed-Integer Linear Programming (MIP)

While MIP problems can be solved optimally in principle, doing so is computationally expensive, even for small problems. However, by leveraging appropriate heuristics, it is possible to reduce the search space while maintaining the optimality criterion. Below we describe a technique for transforming our MINLP problem into an MIP problem and then present a heuristic we have developed that results in improving the resulting MIP algorithm's performance.

In order to transform the MINLP problem to MIP, we need to introduce $|C|^2*|H|^2$ new binary decision variables $t_{c1,h1,c2,h2}$ to the specification formula of each QoS. We want the variable $t_{c1,h1,c2,h2}=1$, if component *c1* is deployed on host *h1* and component *c2* is deployed on host *h2*, and $t_{c1,h1,c2,h2}=0$ otherwise. To ensure that the variable *t* satisfies the above relationship, we need to add the following three new constraints:

$$t_{c1,h1,c2,h2} \leq x_{c1,h1}$$

$$t_{c1,h1,c2,h2} \leq x_{c2,h2}$$

$$1 + t_{c1,h1,c2,h2} \geq x_{c1,h1} + x_{c2,h2}$$

Using the new variable *t* and the three new constraints, we can rewrite the *availValue* function (and other elided QoS functions) to arrive at an equivalent linear problem: $availValue(s) = \sum_{h1}^{|H|} \sum_{h2}^{|H|} \sum_{c1}^{|C|} \sum_{c2}^{|C|} sParam(s, I_{c1,c2}, freq) * nParam(N_{h1,h2}, rel) * t_{c1,h1,c2,h2}$

MIP solvers use branch-and-bound to solve the problem efficiently. Therefore, our problem has the upper bound of:

$$O(size\ of\ branch^{\text{height of tree}}) = O(2^{\text{number of t variables}} + 2^{\text{number of x variables}}) =$$
$$O(2^{|H|^2|C|^2} + 2^{|H||C|}) = O(2^{|H|^2|C|^2})$$

However, most MIP solvers support specification of the order in which the variables are to be branched by assigning different priorities to the decision variables. Thus, by assigning a higher priority to *x* variables and lower priority to *t* variables, we are able to reduce the complexity of the algorithm significantly, to $O(2^{|H||C|})$. This is because, after solving the problem for the *x* variables, the values of *t* variables trivially follow from the three constraints discussed in the previous paragraph. Finally, the constraint that each software component can be deployed on only one host (recall Section 6.1), allows for significant pruning of the branch-and-bound tree, thus reducing the complexity of our problem to $O(|C|^{|H|})$.

By fixing some components to selected hosts, the complexity of the exact algorithm reduces to $O(\prod_{c=1}^{|C|}\sum_{h=1}^{|H|}(loc(c,h))$. Similarly, specifying that a pair of components c_i and c_j have to be collocated on the same host further reduces the algorithm's complexity [40].

As we will see in Chapter 8, even after all this reduction, the MIP algorithm remains computationally very expensive. It may still be used in calculating optimal deploy-

ments for systems whose characteristics are stable for a very long time. In such cases, it may be beneficial to invest the time required for the MIP algorithm, in order to gain maximum possible overall QoS utility. However, note that even in such cases, running the algorithm may become infeasible very quickly, unless the number of allowed deployments is substantially reduced through location and collocation constraints.

6.3 Greedy Algorithm

The high complexity of MIP and MINLP solvers, and the fact that the MINLP solvers do not always find an improved solution, motivated us to devise additional domain-specific approximative algorithms that significantly reduce this complexity while exhibiting good precision. Our approach leverages several heuristics in finding solutions that come close to the optimal found by MIP (for smaller systems) and are on par with those found by state-of-the-art MINLP solvers (for much larger examples).

The greedy algorithm is an iterative algorithm that incrementally finds better solutions. Unlike the previous algorithms that need to finish executing before returning a solution, the greedy algorithm generates a valid and improved solution in each iteration. This is a desirable characteristic for systems where the parameters change frequently and the available time for calculating an improved deployment varies significantly: whenever the algorithm is terminated, it returns either the initial deployment or one that is better than it. In each step of the algorithm, we take a single component *aComp* and estimate the new deployment location for it (i.e., a host) such that

the objective function *overallUtil* is maximized. Our strategy is to improve the QoS dimensions of the "most important" services first. The most important service is the service that has the greatest total utility gain as a result of the smallest improvement in its QoS dimensions. The importance of service *s* is calculated via the following formula: $svcImp(s) = \sum_{u=0}^{|U|} \sum_{q=0}^{|Q|} qosUtil(u,s,q)/qosRate(u,s,q)$

Going in the decreasing order of service importance, the algorithm searches for the host *bestHost* that maximizes the total utility when *aComp* is deployed on it. *bestHost* is the host that has the maximum value of *hValue*, calculated via the following formula:

$$hValue(h, aComp) = \sum_{s}^{|S_{aComp}|} \sum_{u=1}^{|U|} \sum_{q=1}^{|Q|} \left(\frac{\left(\frac{qValue(s,q,d') - qValue(s,q,d)}{qValue(s,q,d)} \right)}{qosRate(u,s,q)} * qosUtil(u,s,q) * qType(q) \right)$$

where *d* is the initial deployment, *d*' is the new deployment when *aComp* is deployed on *h*, and S_{aComp} is a subset of services in whose provision *aComp* is involved.

If the host *bestHost* for *aComp* satisfies all the parameter constraints (e.g., host memory constraint), the solution is modified by mapping *aComp* to *bestHost*. Otherwise, the algorithm tries to find all "swappable" components *sComp* on *bestHost*, such that after swapping a given *sComp* with *aComp* (1) the parameter constraints associated with H_{aComp} and *bestHost* are satisfied, and (2) the overall users' utility is increased. Among the swappable components, we choose the component whose swapping

results in the maximum utility gain, calculated as follows:

$$sValue(aComp,sComp) =$$
$$\text{the utility gain of deploying } aComp \text{ on } bestHost \; - \; \text{the utility effect of deploying } sComp \text{ on } H_{aComp} =$$
$$(hValue(bestHost,aComp) - hValue(H_{aComp},aComp)) - (hValue(H_{aComp},sComp) - hValue(bestHost,sComp))$$

If no swappable components exist, the algorithm selects the host with the next highest *hValue* and repeats the above process.

The algorithm continues improving the overall utility by finding the best host for each component of each service, until it determines that a stable solution has been found. A solution becomes stable when during a single iteration of the algorithm all components remain on their respective hosts. Note that the heuristics implicitly disallow moves that decrease the quality of a solution and thus the algorithm is guaranteed to terminate when it stops improving the quality of the solution. The complexity of this algorithm in the worst case with *k* iterations is:[3]

$$O(\#\,iterations \times \#\,services \times \#\,comps \times \#\,hosts \times hValue\ calculation\ \times \#\,swap\ comps \times hValue\ calculation) =$$
$$O(k \times |S| \times |C| \times |H| \times (|S||U||Q|) \times |C| \times (|S||U||Q|) = O(k|H||S|^3 (|C||U||Q|)^2) = O(|S|^3 (|C||U||Q|)^2)$$

However, since typically only a small subset of components participates in a given service and swappable components are only a small subset of components deployed on the *bestHost*, the average complexity of this algorithm is typically much lower. Finally, similar to the analysis of the MIP algorithm, specification of locational con-

3. Our analysis is based on the assumption that the numbers of system parameters (e.g., sets HP and CP) are significantly smaller than the numbers of modeling elements (i.e., sets H, C, N, and I) they are associated with. In such cases, we can ignore system parameters in our complexity analysis.

straints decreases the number of times we calculate *hValue*, thus resulting in further complexity reduction.

An important heuristic we have introduced in this algorithm is the swapping of components, which significantly decreases the possibility of getting "stuck" in a bad local optimum. Further enhancements to the algorithm are possible that would result in improving the results at the cost of higher complexity. For example, simulated annealing [52] could be leveraged to explore several solutions and return the best one by conducting a series of additional iterations over our algorithm.

6.4 Genetic Algorithm

We present another approximative solution to our problem that is based on a well-known class of stochastic approaches called genetic algorithms [52]. An aspect of a genetic algorithm that sets it apart from the previous algorithms presented in this paper is the fact that it can be extended to execute in parallel on multiple processors with no additional overhead. Furthermore, in contrast with previous two approximative algorithms that eventually stop at "good" local optima (MINLP and greedy), a genetic algorithm continues to improve the solution until it is explicitly terminated by a triggering condition or the global optimal solution has been found. However, the performance and accuracy of the genetic algorithm significantly depends on its mechanism design (i.e., the representation of the problem and the heuristics leveraged in promoting the good properties of individuals). In fact, the genetic algorithm we devel-

oped initially using conventional heuristics typically suggested in literature significantly under-performed in comparison to the other three algorithms. Instead, we had to devise a novel mechanism specifically tailored at our problem, as discussed below.

In a genetic algorithm, an individual represents a solution to the problem. Each individual is composed of a sequence of genes that represent the structure of that solution. A population contains a pool of individuals. An individual for the next generation of the population is evolved in three steps: 1) two or more parent individuals are heuristically selected from the population; 2) a new individual is created via a cross-over between the parent individuals; and 3) the new individual is mutated via slight random modification of its genes.

In our problem, an individual is a string of size $|C|$ that corresponds to the deployment mapping of all software components to hosts. Figure 6-1a shows a simple representation of an individual for a problem of 10 components and 4 hosts. Each block of an individual represents a gene and the number in each block corresponds to the host that the component is deployed on. For example, component 1 of Individual 1 is deployed on host 4 (denoted by h4), as are components 4, 5, and 8. The problem with this representation is that the genetic properties of parents are not passed on to future generations as a result of cross-overs. This is because the components that constitute a service are dispersed in the gene sequence of an individual and a cross-over may result in a completely new deployment for the components of that service. For

instance, assume that in Figure 6-1a service 1 of Individual 1 and services 2 and 3 of Individual 2 have very good deployments (with respect to user utility); then as a result of a cross-over, we may create an individual that has an inferior deployment for all three services. For example, the components collaborating to provide service 2 are now distributed across hosts 1, 3, and 4, which is different from the deployment of service 2 in both Individuals 1 and 2.

Figure 6-1b shows a mechanism we have developed for the representation of an individual in response to this problem. In this representation, the components of each service are grouped together via a mapping function, represented by the Map sequence. Each block in the Map sequence tells us the location on the gene sequence of an indi-

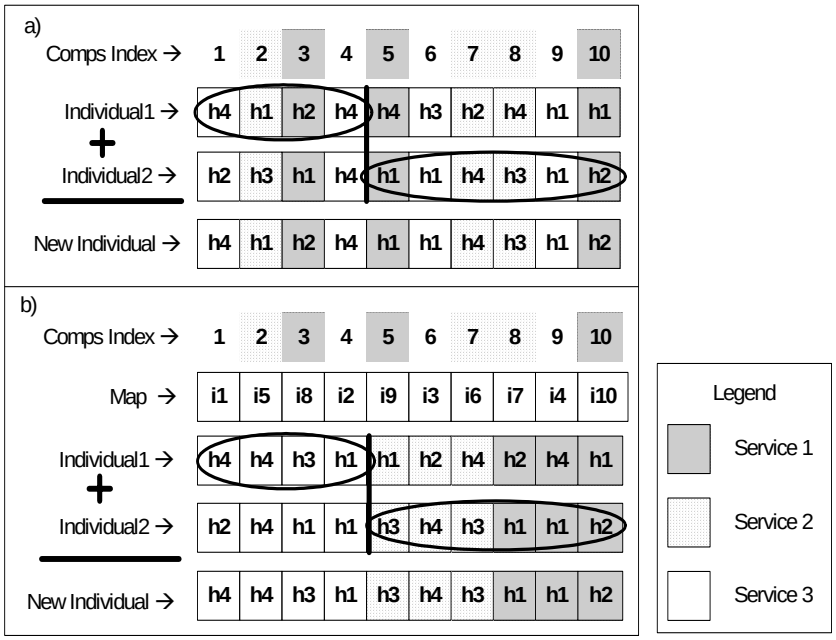


Figure 6-1: Application of the genetic algorithm for a problem of 10 components and 4 hosts: a) Simple representation, b) Representation based on services.

vidual to which a component is mapped. For example, component 2 is mapped to block 5 on the gene sequence (denoted by i5). Thus, block 5 of Individual 1 in Figure 6-1b corresponds to block 2 of Individual 1 in Figure 6-1a, and both denote the fact that component 2 is deployed on host 1. If the component participates in more than one service, the component is grouped with the components providing the service that is most important. Similarly to the greedy algorithm, the most important service results in the highest utility gain for the smallest improvement in its QoS dimensions, calculated via the *svcImp* function from the previous section. We only allow cross-overs that occur on the borders of services. For example, in Figure 6-1b, we may perform cross-overs at two locations: the line dividing blocks 4 and 5, or the line dividing blocks 7 and 8 of Individuals 1 and 2. Note that as a result of the cross-over in Figure 6-1b, we have created an individual that has inherited the deployment of service 1 from *Individual1* and the deployment of services 2 and 3 from *Individual2*.

After the cross-over, the new individual is mutated. In our problem, this corresponds to changing the deployment of a few components. To evolve populations of individuals, we need to define a fitness function that evaluates the quality of each new individual. The fitness function returns zero if the individual does not satisfy all the parameter and locational constraints, otherwise it returns the value of *overallUtil* for the deployment that corresponds to the individual. The algorithm improves the quality of a population in each evolutionary iteration by selecting parent individuals with

a probability that is directly proportional to their fitness values. Thus, individuals with a high fitness value have a greater chance of getting selected, and increase the chance of passing on their genetic properties to the future generations of the population. Furthermore, we directly copy the individual with the highest fitness value (i.e., perform no cross-over or mutation on it) from each population to the next generation, thus keeping the best individual found in the entire evolutionary search.

The complexity of this algorithm in the worst case is:

$$O(\#\,populations \times \#\,evolutions \times \#\,individuals \times fitness\ \ function\ \ calculation\ ) =$$
$$O(\#\,populations \times \#\,evolutions \times \#\,individuals \times |S|\,|U|\,|Q|\,)$$

We can improve the results of the algorithm by instantiating several populations and evolving each of them independently. For further improvement, these populations may be allowed to keep a history of their best individuals and share them with other populations at pre-specified time intervals.

6.5 Market-based Algorithm

In this section, we present *DecAp++*, a decentralized, collaborative algorithm for improving system's deployment architecture. This algorithm is an adaptation of an algorithm, called *DecAp* [29], which we developed previously for finding a deployment architecture that improves the system's availability. Each host that runs DecAp++ contains a single autonomous agent. These agents collaborate to improve the overall utility of the system. Each agent has access to the monitoring data within

its domain of awareness (recall Figure 5-4). An agent exchanges messages with other agents that are members of its host domain.

The auctioned items in DecAp++ are software components. For a component to be ready for auctioning, its relevant parameters must be stable [42]. An agent plays two roles during the redeployment process: (1) auctioneer, in which the agent conducts the auction of its local components, and (2) bidder, in which the agent bids on components auctioned by a remote agent. DecAp++ extends the classic auction algorithm in two ways: (1) an auctioneer is allowed to participate in auctions it conducts, by setting the minimum bid for the auctioned component; and (2) the auctioneer may adjust the received bids.

To participate in an auction conducted on host h_a, a bidder agent has to reside on one of the hosts that are members of h_a's domain. Each agent can be in one of the following three states: *auctioning*, *bidding*, or *free*. The auctioning process for a single component is as follows. First, the auctioneer announces an auction of a local component c_a. It then receives all the bids from bidders within its domain. Finally, the auctioneer determines the "winner", i.e., the location for c_a within $dom(h_a)$ that results in highest availability. To ensure that the winner is correctly determined, agents participating in this auction cannot participate in other auctions at the same time.

As a result of a single auction, a component can move only to one of the hosts that are inside the domain of the component's auctioneer host. For this reason, multiple auctions of a single component may be required before the "sweet spot" for that component in the given distributed system is found. A component's sweet spot is its deployment location that does not change as a result of future auctions for that component. This is known as the Nash Equilibrium State in market-based literature [23].

DecAp++'s auctioneer and bidder algorithms use a function that denotes the contribution of a deployment decision to the overall system utility. The *contribution* of component c_x to the overall utility of the domain of host h_x when c_x is deployed on h_x, is defined as follows:

$$contribution(c_x, h_x) = \sum_{s}^{|S_{C_x}|} \sum_{u=1}^{|U|} \sum_{q=1}^{|Q|} \left(\frac{\left(\frac{qValue(s,q,d'_{h_x}) - qValue(s,q,d_{h_x})}{qValue(s,q,d)} \right)}{qosRate(u,s,q)} * qosUtil(u,s,q) * qType(q) \right)$$

where d_{h_x} is the initial deployment of the components in the *dom(h_x)*, d'_{h_x} is the new deployment when c_x is deployed on h_x, and S_{c_x} is a subset of services in whose provision c_x is involved. Below we describe both the auctioneer's and the bidder's algorithms and how they are coordinated.

6.5.1 Auctioneer's Algorithm

The auctioneer's algorithm, performed on auctioneer's host h_a for one of its software components c_a (i.e., $c_a \in dep(h_a)$), consists of the following eight steps. These steps are repeated for each component on h_a:

1. If c_a is ready to be auctioned, calculate the minimum bid for c_a as follows: $minBid(c_a) = contribution(c_a, h_a)$

2. If h_a's *state* is *free*, change it to *auctioning*, send the AUCTION INTENT message to all hosts in $dom(h_a)$, and proceed to step 3. Otherwise, wait for a given time interval and repeat step 2.

3. If all hosts in $dom(h_a)$ respond with an AUCTION ACCEPT message before the specified time-out, continue to step 4. Otherwise, send AUCTION CANCEL message to all hosts in $dom(h_a)$, set h_a's *state* to *free*, wait for a random time interval, and go back to step 2.

4. Broadcast an AUCTION START message to every host in $dom(h_a)$. Include the *minBid* in the message. The *minBid* sets up a threshold for an acceptable bid. It is used by the bidders to determine whether they qualify to participate in the auction or not.

5. When the bids from all the hosts in $dom(h_a)$ are received, or a time-out occurs, adjust the bids from the hosts that do not satisfy the system constraints (e.g., suffi-

cient memory) for the auctioned component. When a bidding host does not satisfy some of the constraints for component c_a, it needs to trade c_a with one of its local components. As will be detailed in Section 6.5.2, each host h_b that does not satisfy a constraint, in addition to the bid, sends a set of "tradable" components' identifiers $T \subseteq dep(h_b)$ and their contributions (i.e., $\forall c_x \in T \,|\, contribution(c_x, h_b)$). For each host h_b, the auctioneer determines the best candidate component for trade c_t, as a component whose migration from h_b to h_a will have the smallest negative impact on the overall utility, as follows:

$$c_t = \min\langle \forall c_x \in T \,|\, contribution(c_x, h_b) - contribution(c_x, h_a) \rangle$$

Then, the auctioneer recalculates the bid from host h_b to adjust for the effect of the trade, as follows:

$$bid(c_a, h_b) = bid(c_a, h_b) - (contribution(c_t, h_b) - contribution(c_t, h_a))$$

When adjusting the bids for all the hosts that do not satisfy the system constraints is complete, go to step 6.

6. Find the winner host h_w by selecting the highest bidder. If $bid(c_a,h_w) > minBid$, continue to step 7. Otherwise, c_a remains deployed on h_a; skip to step 8.

7. If h_w satisfies all the system constraints, migrate c_a to h_w. Otherwise, perform the trade by migrating c_a to h_w and migrating c_t to h_a.

8. Broadcast an AUCTION TERMINATION message to every host in $dom(h_a)$ to denote the completion of this auction. Set h_a's *state* to *free*.

6.5.2 Bidder's Algorithm

The bidder's algorithm, where $h_b \in dom(h_a)$ is the bidder host, consists of the following eight steps:

1. When an AUCTION INTENT message arrives, if h_b's *state* is *free*, send the AUCTION ACCEPT message to h_a, set the *state* to *bidding*, and continue to step 2. Otherwise, send the AUCTION REJECT message to h_a.

2. If an AUCTION CANCEL message arrives, set the *state* to *free*, and go back to step 1. If the AUCTION START message arrives from h_a, calculate the bid for c_a as the contribution of c_a to the overall utility of $dom(h_b)$ if c_a were to be deployed on h_b: $bid(c_a, h_b) = contribution(c_a, h_b)$

3. If $bid(c_a,h_b) < minBid$, h_b does not qualify to place a bid on c_a, skip to step 8. Otherwise create the bid message by including the $bid(c_a,h_b)$. Proceed to step 4.

4. If h_b satisfies all the system constraints for c_a, proceed to step 7.

5. Since h_b does not satisfy some system constraint(s) for the deployment of c_a, find the set $T \subseteq dep(h_b)$ of "tradable" components. A component is tradable when as a result of trading it with c_a all the system constraints are satisfied.

6. If T is not empty, append to the bid message both the identifiers of all components $c_x \in T$ and their contributions, $contribution(c_x,h_b)$, and proceed to step 7. Other-

wise, when T is empty, a tradable component does not exist and component c_a cannot be deployed onto h_b; skip to step 8.

7. Place the bid by sending the bid reply message to h_a.

8. Upon arrival of the AUCTION TERMINATION message, set h_b's state to *free*.

6.5.3 Analysis of the Two Algorithms

To ensure that an agent participates in a single auction at a time, we employed a distributed locking mechanism using the *state* variable for each agent as described in steps 2, 3, and 8 of the auctioneer's algorithm, and steps 1, 2, and 8 of the bidder's algorithm. To avoid deadlocks and starvation, each auctioneer waits a random interval of time before the next attempt at starting an auction.

The worst-case time complexity analysis for each of the two algorithms is given below. Note that the analysis of agent synchronization time complexity is not provided, since we adopted a well-known distributed locking technique, whose complexity analysis is provided in [62].

O(auctioneer) = O(step 1) + O(step 5) + O(step 6) = O(|C|*|S||U||Q|) + O(|C|*|S||U||Q|) + O(|H|) = O(|C||S||U||Q|)

O(bidder) = O(step 2) + O(step 5) = O(|S||U||Q|) + O(|C|) = O(|S||U|Q|)

Finally, the auctioneer's algorithm will be executed several times for each software component. Some of these auctions may occur simultaneously within the entire system, depending on the number of components on each host and the number of hosts within each host's domain. In the worst case (e.g., the domain of each host is the entire set of hosts H), the auctioneer's algorithm executes in a sequential manner for each component, resulting in the total complexity of DecAp++ to be $|C|*O(auctioneer) = O(|C|^2|S||U|Q|)$.

Algorithm's Convergence. DecAp++ performs a redeployment of components only if the redeployment results in the overall utility increase. For this reason, each auction guarantees that the system's utility will either increase or remain the same (if the auctioned component remains on the auctioneer host). As will be illustrated in Chapter 8, the algorithm typically converges after only a few auctions for each component, i.e., subsequent auctions do not change the deployment architecture of the system. As soon as the given host becomes the "sweet spot" for all of its components, the auctioneer algorithm on that host assumes the algorithm's convergence with a certain degree of confidence, and extends the period of time before attempting a new auction (i.e., the host's dormant time). If during subsequent auctions the host remains the "sweet spot" for its components, its degree of confidence, and thus the period of dormancy, increases.

Algorithm's Sensitivity to the Level of Awareness. DecAp++ provides a flexible approach for capturing the level of awareness present at each node, through careful definition of the *aware* function and *dom* relation in our model. DecAp++'s model does not make any assumptions about what constitutes awareness among two hosts (i.e., when $aware(h_i,h_j)=1$). We simply set a given host's domain (i.e., the *dom* relation) to the set of all the hosts of which it is *aware*. The model can then be instantiated with an implementation-level definition of awareness. Some commonly used policies in determining aware hosts are: directly connected hosts, proximity of hosts, number of node hops, bandwidth or signal strength, and reliability of links. Figure 6-2 illustrates the effect of using different policies for determining host awareness. While our algorithm is independent of the policy that constitutes host awareness, the performance of the algorithm is significantly affected by the level of awareness present at each host. We will demonstrate the sensitivity of our algorithm to the level of awareness in Chapter 8.

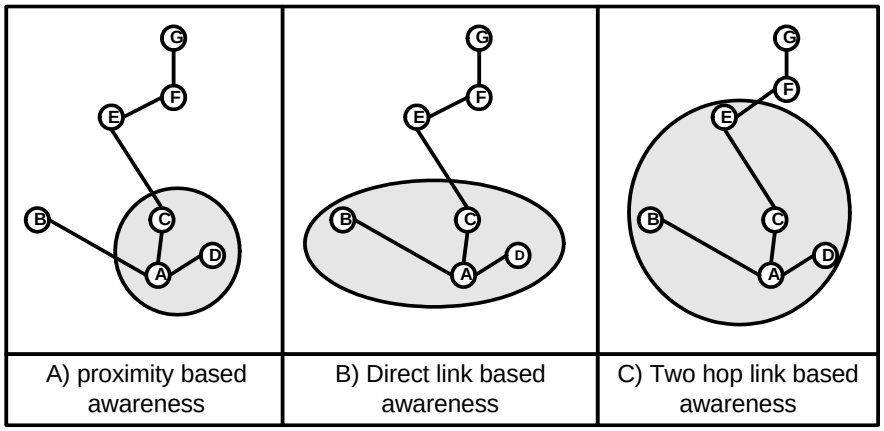


Figure 6-2: Domain of host A with different policies for determining host awareness.

Location Constraints. In Section 5.1 we discussed how the *loc* and *colloc* functions can be leveraged to capture constraints other than memory. For clarity in presenting DecAp++ we did not explicitly describe how the location constraints remain satisfied throughout the algorithm's execution. The constraint imposed by the *loc* function is enforced by inviting the hosts to participate in an auction only if they satisfy the *loc* constraint. The constraint imposed by the *colloc* function is enforced as follows: (1) when a component cannot be on the same host as the auctioned component, the auctioneer simply does not invite the host that contains that component to the auction, and (2) when two component have to be on the same host, the components are merged into a single virtual component and therefore always auctioned at the same time. Also note that through the use of *loc* and *colloc*, the complexity of the algorithm is reduced proportionally to the extent of the constraints imposed by the two functions in the given system [40].

CHAPTER 7: Tool-Support

The framework has been realized on top of an integrated tool suite, which allows the engineer to model an application scenario, use one of the provided algorithms for improving its architecture, and finally effect the results of the analysis by redeploying the software components. In this chapter we describe the tool suite, which is composed of a customizable deployment modeling and analysis environment (called DeSi [43]) and an extensible architectural middleware platform (called Prism-MW [28]). DeSi and Prism-MW provide an integrated tool suite that can be leveraged by software engineers to develop reusable solutions to this problem.

7.1 Architectural Middleware

This section describes Prism-MW [28], a middleware that supports architecture-based software development. Prism-MW forms the foundation for the proposed research, as it provides support for monitoring, (re)deployment, and dynamic reconfiguration at the architectural level.

7.1.1 Middleware Design

Prism-MW supports architectural abstractions by providing classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. These abstractions enable a direct mapping between an architecture and

its implementation. Furthermore, Prism-MW employs a well-defined extensibility mechanism for addressing emerging development concerns.

Figure 7-1 shows the class design view of Prism-MW. The shaded classes constitute the middleware core, which represents a minimal subset of Prism-MW required for implementing and executing an architecture. Note that the core does not include support for key Prism concerns such as architectural styles, mobility, awareness, and so on. Those concerns are addressed by Prism-MW's extensions, which we discuss below. Only the five dark gray classes of Prism-MW's core are directly relevant to the application developer. Our goal was to keep the core compact, reflected in the fact that it contains only twelve classes (four of which are abstract) and four interfaces. Furthermore, we tried to keep the design of the core (and the entire middleware) highly modular, by limiting direct dependencies among the classes via abstract classes, interfaces, and inheritance as discussed below.

Brick is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection, possibly at system runtime. A distributed application is implemented as a set of interacting *Architecture* objects.

*Event*s are used to capture communication in an architecture. An event consists of a name and payload. An event's payload includes a set of typed parameters for carrying data and meta-level information (e.g., sender, type, and so on). Two base event types are *request* for a recipient component to perform an operation and *reply* that a sender component has performed an operation.

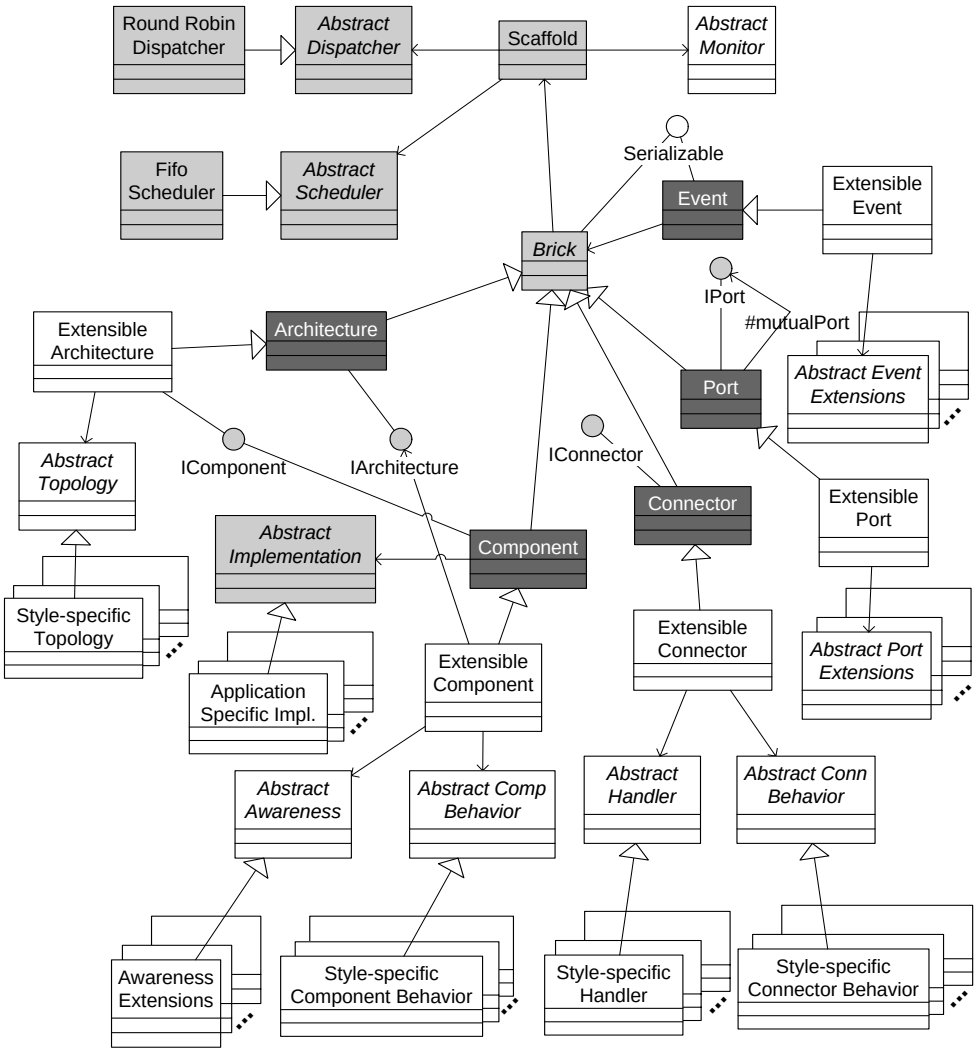


Figure 7-1: UML class design view of Prism-MW. Middleware core classes are highlighted.

*Port*s are the loci of interaction in an architecture. A *link* between two ports is made by *weld*ing them together; the link acts as a bidirectional communication channel between the ports. A port can be welded to at most one other port. Each *Port* has a type, which is either *request* or *reply.* An event placed on one port is forwarded to the port linked to it in the manner shown in Figure 7-2: request events are forwarded from request ports to reply ports, while reply events are forwarded in the opposite direction.

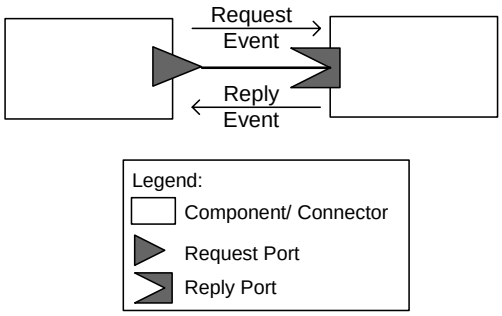


Figure 7-2: Link between two Ports in Prism-MW.

*Component*s perform computations in an architecture and may maintain their own internal state. A component is dynamically associated with its application-specific functionality via a reference to the *AbstractImplementation* class. This allows us to perform dynamic changes to a component's application-specific behavior without having to replace the entire component. Each component can have an arbitrary number of attached ports. Components interact with each other by exchanging events via their ports. When a component generates an event, it places copies of that event on each of its ports whose type corresponds to the generated event type.

Components may interact either directly (through ports) or via connectors. *Connector*s are used to control the routing of events among the attached components. Like components, each connector can have an arbitrary number of attached ports. A component attaches to a connector by creating a link between one of its ports and a single connector port. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast). In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime. This property, coupled with event-based interaction, has proven to be highly effective for addressing system reconfigurability, further discussed below.

Each subclass of the *Brick* class has an associated interface. The *IArchitecture* interface exposes a *weld* method for attaching two ports together. The *IComponent* interface exposes *send* and *handle* methods used for exchanging events. *Component* provides the default implementation of *IComponent*'s *send* method: generated request events are placed asynchronously on all of the request ports attached to the component, while generated reply events are placed asynchronously on all of the attached reply ports. The *IConnector* interface provides a *handle* method for routing events. The *Connector* class provides the default implementation of *IConnector*'s *handle* method, which forwards all request events to the connector's attached request ports and all reply events to the attached reply ports. We also provide implementations of different routing policies, including unidirectional broadcast, bidirectional broadcast,

and multicast. The *IPort* interface provides the *setMutualPort* method for creating a one-to-one association between two ports.

Finally, Prism-MW's core associates the *Scaffold* class with every *Brick*. *Scaffold* is used to schedule and queue events for delivery (via the *AbstractScheduler* class) and pool execution threads used for event dispatching (via the *AbstractDispatcher* class) in a decoupled manner. Prism-MW's core provides default implementations of *AbstractScheduler* and *AbstractDispatcher*: *FIFOScheduler* and *RoundRobinDispatcher*, respectively. The novel aspect of our design is that this separation of concerns allows us to independently select the most suitable event scheduling, queueing, and dispatching policies for a given application. Furthermore, it allows us to independently assign different scheduling, queueing, and dispatching policies to each architectural element, and possibly even change these policies at runtime. For example, a separate event queue can be assigned to each component; alternatively, a single event queue can be shared by a number of (collocated) components. Additionally, dispatching and scheduling are decoupled from the *Architecture*, allowing one to easily compose many sub-architectures (each with its own scheduling and dispatching policies) in a single application. *Scaffold* also directly aids architectural awareness (also referred to as reflection) by allowing probing of the runtime behavior of a *Brick* via different implementations of the *AbstractMonitor* class, which will be discussed in more detail in Section 7.1.10.

7.1.2 Developer's View of Prism-MW

Prism-MW's core provides the necessary support for developing arbitrarily complex applications, so long as they rely on the default facilities (e.g., event scheduling, dispatching, and routing). Prism-MW's extensions, which will be discussed below, are used in a similar manner. The first step a developer takes is to create the application-specific portion of each component, by subclassing from the *AbstractImplementation* class and providing the component's "business logic" inside its *handle* and *start* methods. The developer then instantiates the Prism-MW *Component*s and associates each *Component* with its implementation. Next, the developer instantiates the *Architecture* class and adds the *Component*s (and *Connector*s if they are used in a given architecture) to the *Architecture*. The developer then creates instances of request and reply *Port*s for components (and connectors), and associates them with their container *Component*s (or *Connector*s), using the *addPort* method. Attaching component and connector *Port*s into a configuration is achieved by using the *weld* method of the *Architecture* class. Finally, *Architecture*'s *start* method performs initialization of its constituent elements by invoking their individual *start* methods.

For illustration, Figure 7-3 shows two alternative usage scenarios in the Java version of Prism-MW. The code fragments correspond to two different implementations of a subset of the TDS application introduced in Chapter 7, where the instantiation and interaction of three components (*RenderingAgent, DeploymentAdvisor,* and *ResourceMonitor*) is highlighted. In the first usage scenario the components are communicat-

ing directly, while in the second scenario the components are communicating through a connector. The alternative implementations of the *TDSDemo* class's *main* method instantiate components, ports (and, in the second case, connectors) and compose

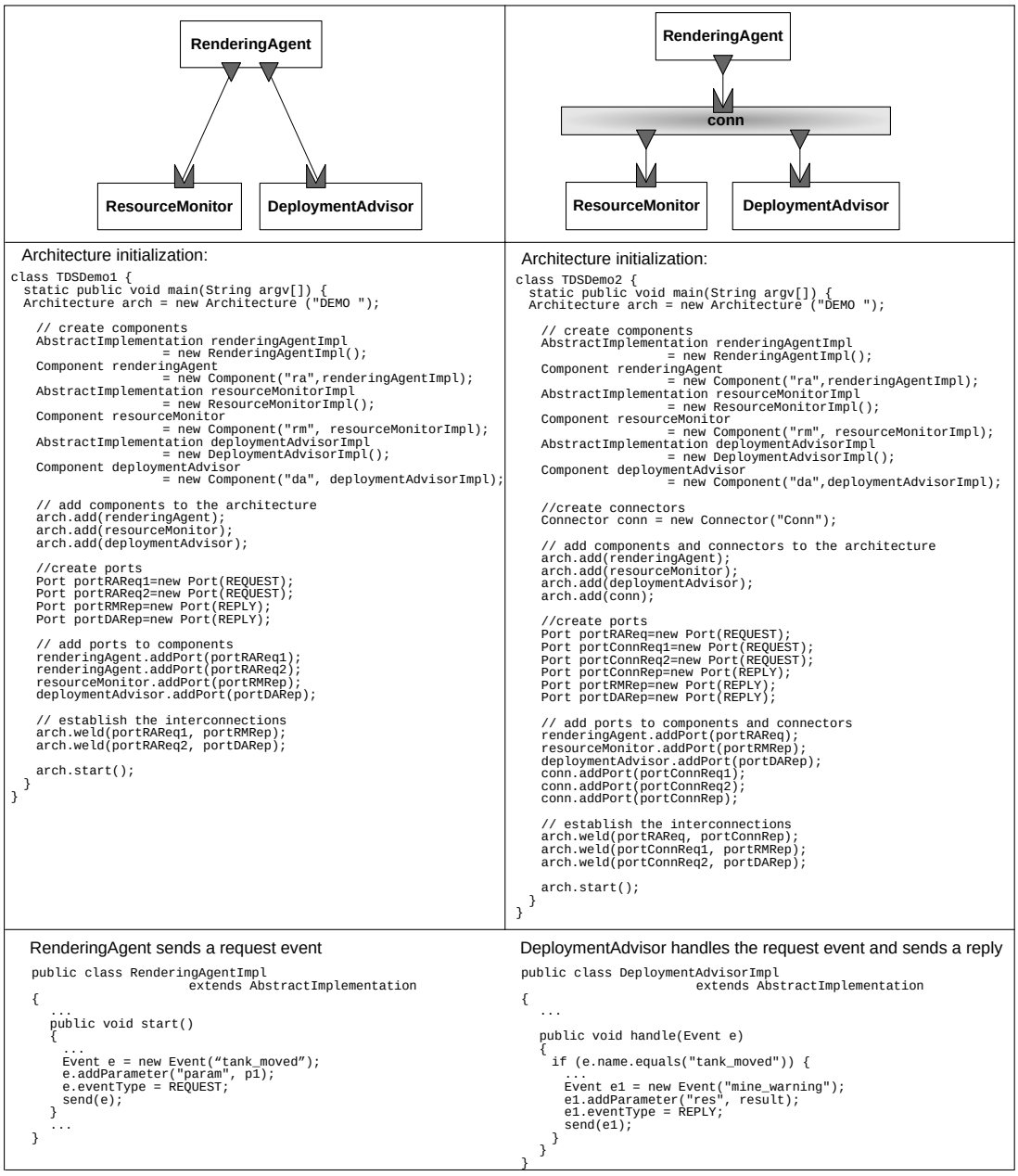


Figure 7-3: Prism-MW application implementation fragments.

(*weld*) them into a configuration. Figure 7-3 also demonstrates event-based communication between the two components (as implemented in *RenderingAgentImpl* and *DeploymentAdvisorImpl* classes). Component *RenderingAgent* creates and sends a request event, in response to which Component *DeploymentAdvisor* sends a reply event in the manner detailed in Section 7.1.3.

Prism-MW's core has been implemented in Java JVM and C++. Large subsets of the described functionality have also been implemented in Java KVM, Embedded Visual C++ (EVC++), Python, and Brew; they have been used in example applications and in evaluating Prism-MW. The implementation of the middleware core is relatively small (under 900 SLOC), which can aid Prism-MW's understandability and ease of use.

7.1.3 Prism-MW's Semantics

A distributed system implemented in Prism-MW consists of a number of *Architecture* objects, each of which serves as a container for a single subsystem and delimits an address space. *Component*s within and across the different *Architecture* objects interact by exchanging *Event*s. The default implementation of Prism-MW uses a fixed-sized, circular array for storing all events in a single address space. This allowed us to optimize event processing by introducing a pool of shepherd threads (implemented in Prism-MW's *RoundRobinDispatcher* class) to handle events sent by any component in a given address space. The size of the thread pool is parameterized and, hence,

adjustable. Since the event queue is of a fixed size (determined at system construction-time) we also use a producer-consumer algorithm to keep event production under control, and supply shepherd threads with a constant stream of events to process.

Figure 7-4 shows event processing for two alternative usage scenarios of Prism-MW introduced in Section 7.1.2. This figure shows the base case of event processing within a single address space. At the same time, this technique proved quite flexible and allowed us to support distributed event processing in a very similar manner, as will be discussed in Section 7.1.5.

By default, Prism-MW processes events asynchronously. A shepherd thread removes the event from the head of the queue. In the first scenario (Figure 7-4a), the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread. If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. The second usage scenario (shown in Figure 7-4b) uses direct connections between component ports and allows separate threads to be used for dispatching an event from the queue to each intended recipient component (steps 2-3 a and b in Figure 7-4b). This increases parallelism, but also resource consumption in the architecture. This solution represents an adaptation of an existing worker thread pool technique [54] that results in several unique benefits:

1. By leveraging explicit architectural topology, an event can be routed to multiple destinations. This minimizes resource consumption, since events need not be tagged with their recipients, nor do the recipients need to explicitly subscribe to events.

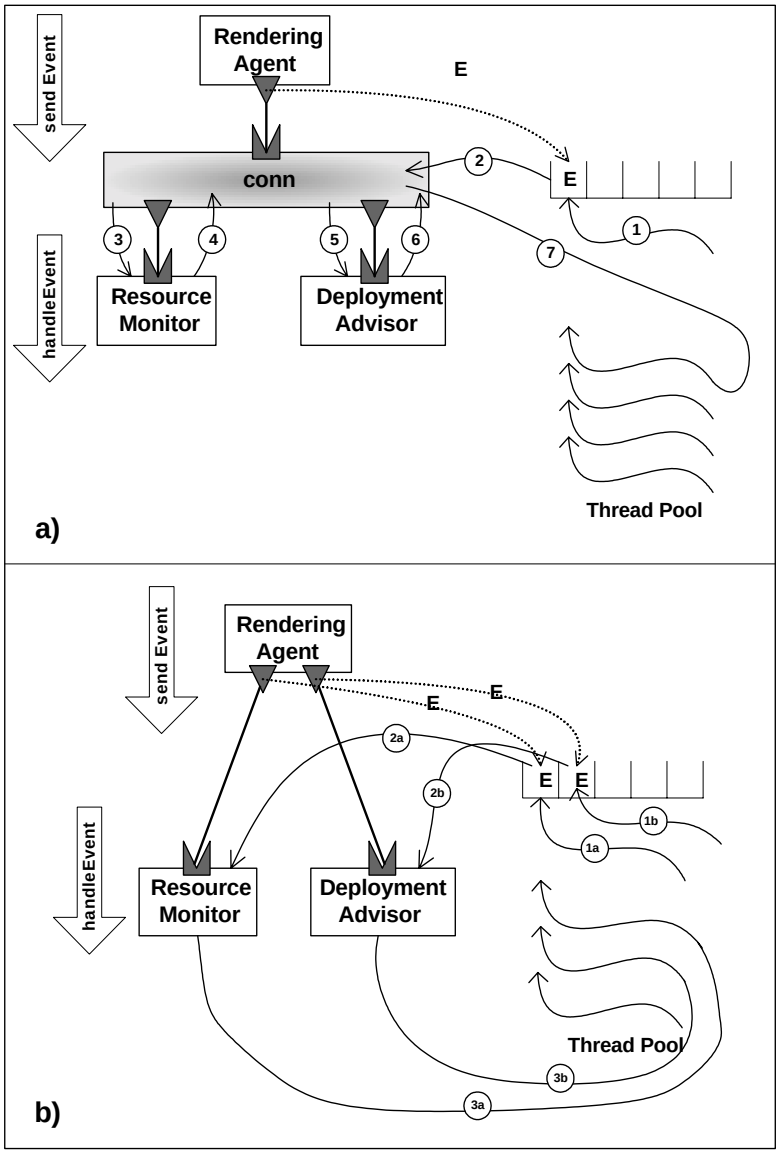


Figure 7-4: Event dispatching in Prism-MW for a single address space. a) Steps (1)-(7) are performed by a single shepherd thread. b) steps 1-3 are performed by two shepherd threads, assuming the RenderingAgent is sending event E to both recipient components.

2. We further optimize resource consumption by using a single event queue for storing both locally and remotely generated events (further discussed in Section 7.1.5and depicted in Figure 7-8).
3. Since Prism-MW processes local and remote events uniformly, and all routing is accomplished via the multiple and explicit ports and/or connectors, Prism-MW allows for seamless redeployment and redistribution of existing applications onto different hardware topologies.

7.1.4 Extensibility Mechanism

One of Prism-MW's key objectives is extensibility. The design of Prism-MW's core is intended to support this objective by providing extensive separation of concerns via explicit architectural constructs and use of abstract classes and interfaces. To date, we have built several specific extensions to support architectural awareness, real-time requirements, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility. Furthermore, we have been able to support directly multiple architectural styles, even within a single application. Our experience with the extensions we have built to date indicates that others can be easily added to the middleware in the manner presented here.

Our support for extensibility is built around our intent to keep Prism-MW's core unchanged. To that end, the core constructs (*Component*, *Connector*, *Port*, *Event*, and *Architecture*) are sub-classed via specialized classes (*ExtensibleComponent*, *ExtensibleConnector*, *ExtensiblePort*, *ExtensibleEvent*, and *ExtensibleArchitecture*), each of which has a reference to a number of abstract classes (*AbstractExtension*s in Figure 7-5). Each *AbstractExtension* class can have multiple implementations (*Extension i,j* in Figure 7-5), thus enabling the selection of the desired functionality inside each instance of a given *Extensible* class. If a reference to an *AbstractExtension* class is instantiated in a given *Extensible* class instance, that instance will exhibit the behavior realized inside the implementation of that abstract class. Multiple references to abstract classes may be instantiated in a single *Extensible* class instance. In that case, the instance will exhibit the combined behavior of the installed abstract class implementations.[4]

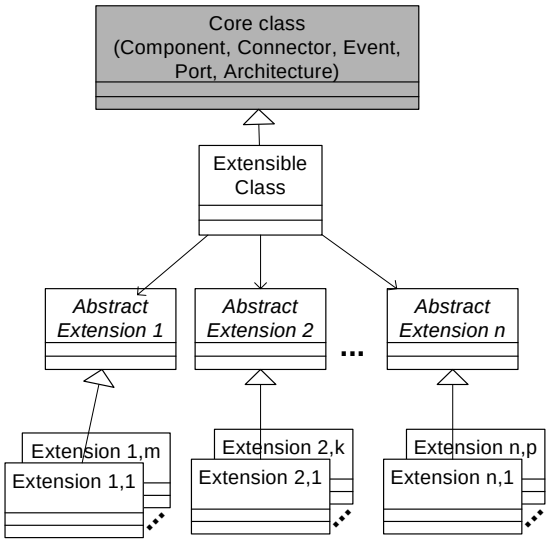


Figure 7-5: Prism-MW's extensibility mechanism

4. Since Prism-MW's objective is to be arbitrarily extensible in principle, the middleware places no restrictions on, and is agnostic as to the semantics of, such combined behaviors. It is the developers' responsibility to ensure that such behaviors make sense.

7.1.5 Distribution

In order to address different aspects of interaction, the *ExtensiblePort* class has references to a number of abstract classes that support various interaction services. In turn, each abstract class can have multiple implementations. Figure 7-6 shows five different port extensions we have implemented thus far. Given the importance of distribution to the Prism domain, in this section we focus solely on the distribution extensions. In Section 7.1.6 we will discuss other aspects of interaction.

The *AbstractDistribution* class has been implemented by two concrete classes, one supporting socket-based and the other infrared port-based inter-process communication (IPC). We refer to an *ExtensiblePort* with an instantiated *AbstractDistribution*

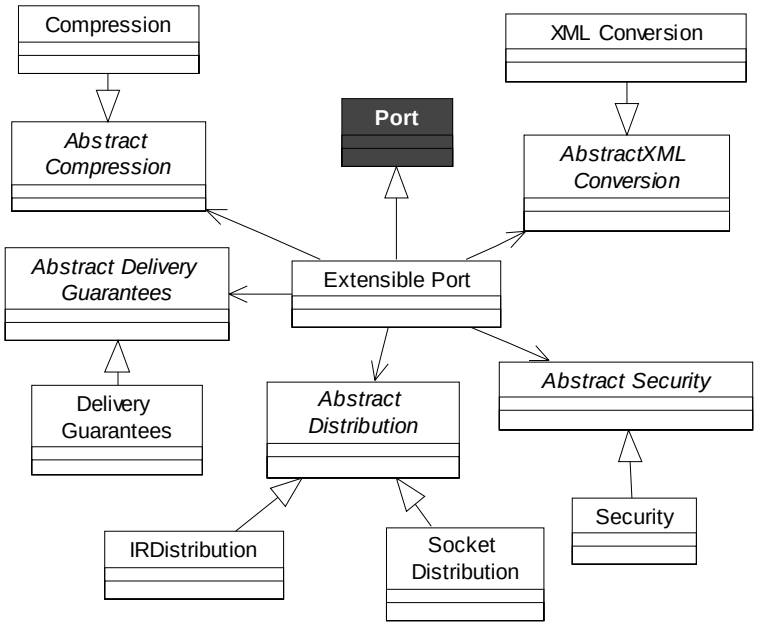


Figure 7-6: Port extensions.

reference as a *DistributionEnabledPort*. A *DistributionEnabledPort* can be instantiated in two modes of operation: server or client. A *DistributionEnabledPort* operating in the server mode has a listening thread (e.g., socket server) that is waiting for incoming connection requests on a specified network port. A *DistributionEnabledPort* operating in the client mode does not have a listening thread and is only capable of making connection requests to other *DistributionEnabledPort*s.

Our implementation of *AbstractDistribution* allows a *DistributionEnabledPort* to have an arbitrary number of network connections to other remote *DistributionEnabledPort*s (i.e., one-to-many association between ports). When a *DistributionEnabledPort* receives an event, it broadcasts the event on all its network connections. Note that the one-to-many association between *DistributionEnabledPort*s is a deviation from the one-to-one semantics of a basic port. As will be further discussed in Chapter 8, this deviation is introduced for efficiency.

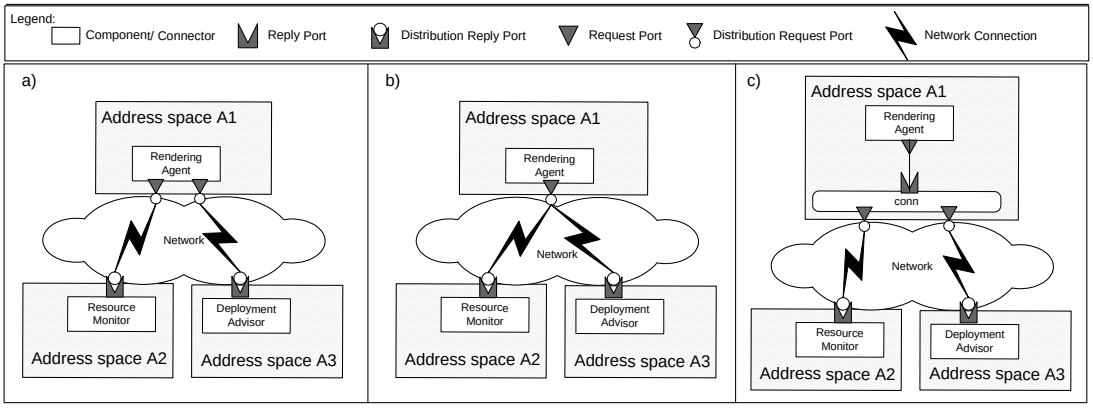


Figure 7-7: *DistributionEnabledPort* usage scenario

Figure 7-7 shows three different usage scenarios of Prism-MW's *DistributionEnabledPort*s. The architectures in Figures 7-7a and 7-7b are semantically identical (in both cases, a request from *RenderingAgent* will be sent to both recipient components), although the architecture of Figure 7-7b is more efficient as it uses one less *DistributionEnabledPort*. On the other hand, the architecture in Figure 7-7c allows the connector in address space A1 to route an outgoing event either to *ResourceMonitor* in A2 or *DeploymentAdvisor* in A3, or both.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: a sending component or connector places its outgoing event on an attached port. However, instead of depositing the event to the local event queue, in this case the *DistributionEnabledPort* deposits the event on the network, as shown in Figure 7-8. When the event is propagated across the network, the (server) *DistributionEnabledPort* on the recipient device uses its internal thread to retrieve the incoming event and place it on its local event queue. This supports distribution transparency and allows a component to be migrated between hosts with minimal impact on the system. For example, the impact of moving *ResourceMonitor* from A2 to A1 in Figure 7-8 would be limited to replacing two *DistributionEnabledPort*s (one attached to the *conn* connector and the other to *ResourceMonitor*) with local *Port*s.

7.1.6 Communication Properties

As depicted in Figures 7-6 and 7-9, we have implemented several port and event extensions, respectively, which enable different aspects of communication beyond distribution. Below we provide a brief description of each.

Security. The *AbstractSecurity* class (shown in Figure 7-6) is a port extension that has several implementations performing combinations of authentication, authorization, encryption, and event integrity. These services have been implemented using three major cryptographic algorithms: symmetric key, asymmetric key, and event digest function [50]. Asymmetric key (a.k.a. public key) algorithms are generally more computationally complex and therefore much slower than symmetric key (a.k.a.

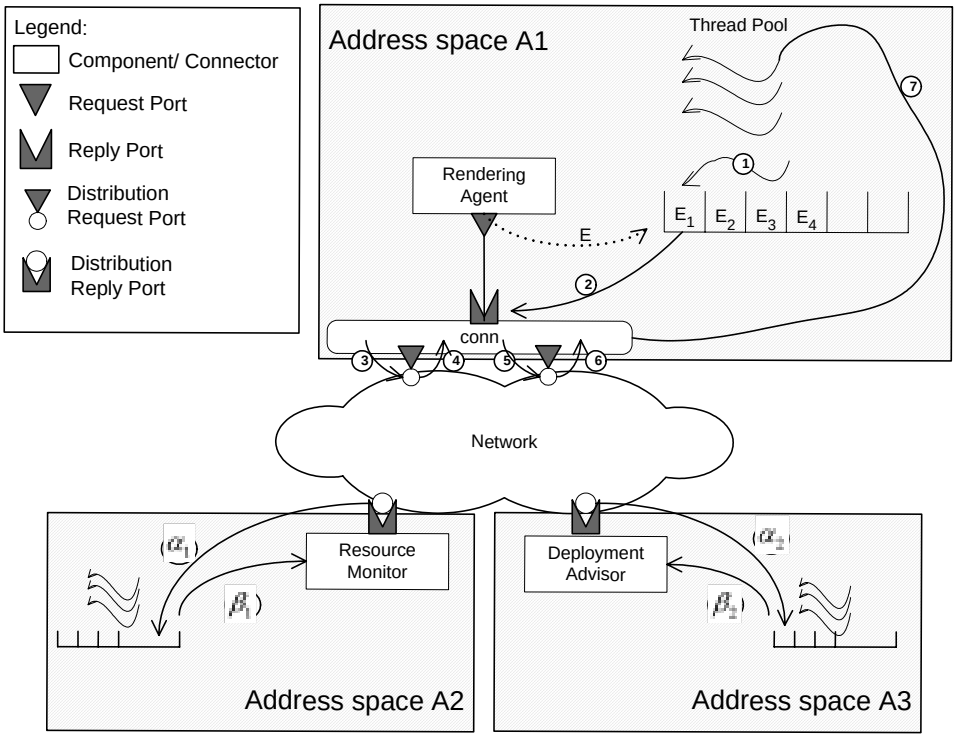


Figure 7-8: Event dispatching in Prism-MW for the remote scenario

secret key) algorithms. However, asymmetric key algorithms do not require secret channels to distribute the key. Therefore, we chose the RSA asymmetric key algorithm for establishing a connection between each two new users (i.e., their *DistributionEnabledPort*s) and for transmitting the secret (session) key. The same session key and the DES symmetric key algorithm is used for all subsequent transactions (i.e., exchanging events). In order to prevent request tampering, a message digest function is used in combination with RSA to generate the message signature. A message digest is a kind of cryptographic checksum over a message, used to verify data integrity.

Delivery Guarantees. *AbstractDeliveryGuarantees* and *AbstractDeliveryGuaranteesEvt* are port and event extensions, respectively, that support event delivery guarantees. We have implemented support for at most once, at least once, exactly once, and best effort delivery semantics. Each Prism-MW event is tagged with the delivery policy, with best effort being the default. Communicating *DistributionEnabledPort*s with the appropriate delivery guarantee extensions installed on them implement a "handshaking" protocol to ensure proper event delivery across address spaces. In order to maximize the efficiency of

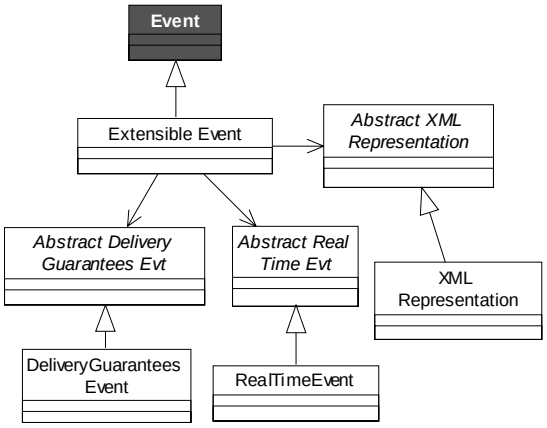


Figure 7-9: Event extensions.

the delivery guarantee support in the same address space, we make use of programing language (PL) exceptions (i.e., we assume that if no exception is raised, the event has been delivered). This optimization is possible since Prism-MW's event passing capability in a single address space is implemented on top of the underlying PL's (synchronous) method calls.

Real-Time Delivery. The *AbstractRealTimeEvent* class (shown in Figure 7-9) is used to assign a real-time deadline to an event. We have implemented this class to support both aperiodic and periodic real-time events. In support of real-time event delivery we have provided two additional implementations of the *AbstractScheduler* class (see Figure 7-12). *EDFScheduler* implements scheduling of aperiodic events based on the earliest-deadline-first algorithm, while *RateMonotonicScheduler* implements scheduling of periodic events via rate monotonic scheduling [26]. Coupled with this, we have provided *PriorityDispatcher*, which is a variant implementation of *AbstractDispatcher* that supports threads with varying priorities. These extensions are shown in Figure 7-12. Furthermore, in order to optimize the event-processing time, Prism-MW connectors have been specialized to demultiplex and dispatch incoming events using request and reply filtering: each event is routed only to the components that have subscribed for it at startup time.

Data Conversion. In order to support communication across PLs, we have provided the *AbstractXMLConversion* and *AbstractXMLRepresentation* extensions for port and event classes, respectively. Prism-MW events with the installed *XMLRepresentation* (our default implementation of *AbstractXMLRepresentation*) are encoded/decoded via ports with *XMLConversion* (our default implementation of *AbstractXMLConversion*).

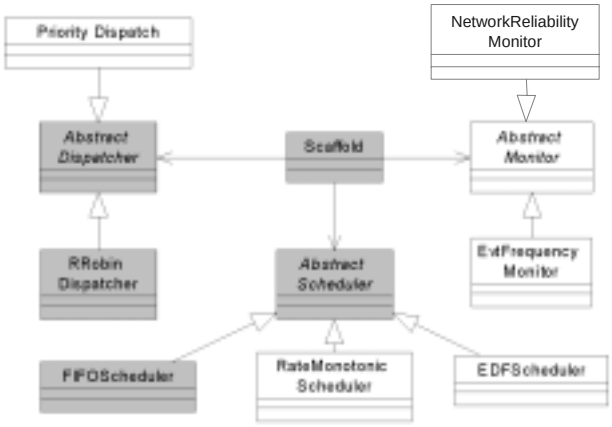


Figure 7-10: Scaffold extensions.

Data Compression. Finally, we have provided the *AbstractCompression* extension for port with the goal of minimizing the required network bandwidth for event dispatching. To this end, we have implemented the Huffman coding technique [53] inside the *Compression* class.

The extensions discussed here represent some of the frequently needed services in the Prism setting. We have developed them over the past several years as the need for them has arisen. Adding new extensions to Prism-MW is relatively straightforward. For example, addition of a new extension to *ExtensiblePort* requires adding a refer-

ence to the appropriate abstract class and invoking its methods inside *ExtensiblePort*'s *handle* method. Such a change to an *Extensible* class is minimal, averaging three new lines of code for each new extension.[5] The overhead introduced by this solution is that an *ExtensiblePort* instance may have many *null* references, corresponding to the extension classes that have not been instantiated. The values of these references will be checked each time *ExtensiblePort*'s *handle* method is invoked. An alternative solution, which would trade-off the extensibility for efficiency, is to subclass the *Port* class directly and to have the references only to the desired extensions.

7.1.7 Dynamism

Prism-MW supports dynamic reconfiguration of the system's architecture and its properties via six runtime capabilities:

- *Addition and removal of components and connectors.* An existing component/connector can be removed from the architecture via *IArchitecture*'s *remove* method. Similarly, an instantiated component/connector can be added to the architecture via *IArchitecture*'s *add* method.
- *Attachment of ports to components and connectors.* A Prism-MW connector or component does not contain any connection points (ports) at declaration time. Instead, ports are added and removed at runtime.

5. Again, it is important that developers ensure the right ordering of method calls to achieve the desired behavior. For example, when combining *AbstractSecurity* and *AbstractXMLConversion* extensions, *AbstractXMLConversion*'s *convert* method is invoked before *AbstractSecurity*'s *encrypt* method when sending the event; on the receiving end, the *AbstractSecurity*'s *decrypt* method is invoked before *AbstractXMLConversion*'s *reconstitute* method.

- *(Dis)association of two ports via the (un)weld method.* Two ports can be associated with one another at runtime via the architecture's *weld* and disassociated via *unweld* method, resulting in a restructured architecture.
- *Modification of communication.* As discussed above, the localization of communication properties to *port* (and *ExtensiblePort*) objects makes it trivial to modify the communication at the component or connector level by attaching a new port at runtime.
- *Modification of style.* Prism-MW provides the ability to modify the system's architectural style, possibly at runtime, via installation of new extensions [28].
- *Modifying the size of queue or thread pool*. Increasing the size of the queue is done simply by allocating more memory to the queue, while decreasing the size of the queue deallocates the reserved memory as long as there is unused memory in

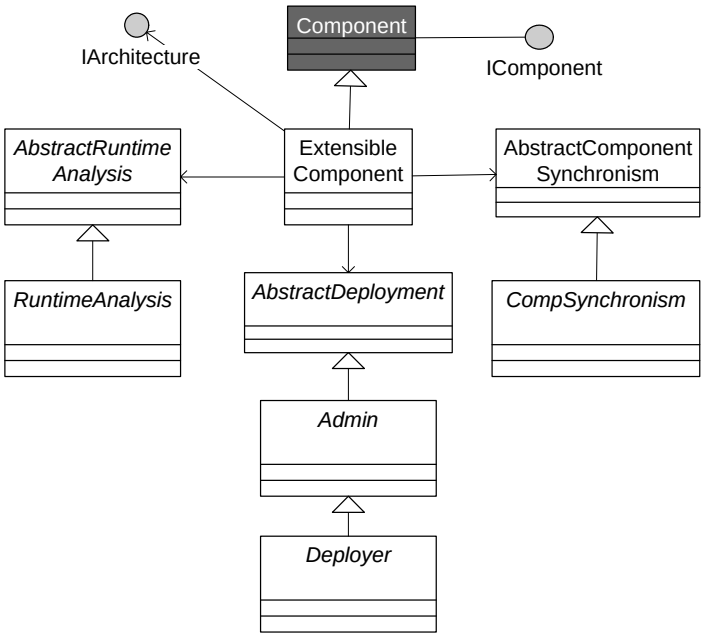


Figure 7-11: Component extensions.

the queue. Each shepherd thread in Prism-MW is encapsulated in a Java class that is associated with the queue at runtime. Thus, adding new threads is done by instantiating more shepherd thread objects, while removing threads can be done only when the thread is inactive (i.e., waiting) by halting its execution.

7.1.8 Awareness

To support various aspects of awareness (i.e., reflection), Prism-MW supports meta-level components. Typically, a meta-level component is implemented as an *ExtensibleComponent*, which contains a reference to the *Architecture* object via the *IArchitecture* interface and allows the component's instances to effect runtime changes on the system's local (sub)architecture. The *ExtensibleComponent* class can also have references to abstract classes that provide specific (meta-level) functionality (see Figure 7-11). The role of components at the meta-level is to observe and/or facilitate different aspects of the execution of application-level components. At any point, the developer may add meta-level components to a (running) application. Meta-level components may be welded to specific application-level connectors to exercise control over a particular portion of the architecture. Alternatively, a meta-level component may remain unwelded and may instead exercise control over the entire architecture via its pointer to the *Architecture* object. The structural and interaction characteristics of meta-level components are identical to those of application-level components, eliminating the need for their separate treatment in the middleware.

To date, we have augmented *ExtensibleComponent* with three extensions. In the next section we discuss in detail the *AbstractDeployment* class, which is used for performing component deployment and mobility. A detailed description of other extensions is provided in [28].

7.1.9 Deployment and Mobility

Our support for mobility exploits Prism-MW's explicit software connectors, event-based interaction, support for dynamism, and awareness. Below we discuss both stateless and stateful component mobility.

7.1.9.1 Stateless Mobility

Prism-MW components communicate by exchanging application-level events. Prism-MW also allows components to exchange *ExtensibleEvents*, which may contain architectural elements (components and connectors) as opposed to data. Additionally, *ExtensibleEvents* implement the *Serializable* interface (recall Figure 7-1), thus allowing their dispatching across address spaces.

In order to migrate the desired set of architectural elements onto a set of target hosts, we assume that a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW's *Architecture* object that contains an *Admin* with a *DistributionEnabledPort* attached to it. An *Admin* is an *ExtensibleComponent* with

the *Admin* implementation of *AbstractDeployment* installed on it (shown in Figure 7-11).

Since the *Admin* on each device contains a pointer to its *Architecture* object, it is able to effect runtime changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. *Admin*s are able to send and receive from any device to which they are connected the *ExtensibleEvent*s that contain application components and connectors (referred to as migrant elements below).

The process of stateless migration can be described as follows. The sending *Admin* packages the migrant element into an *ExtensibleEvent*: one parameter in the event is the compiled image of the migrant element itself (e.g., a collection of Java class files); another parameter denotes the intended location of the migrant element in the destination subsystem's configuration. The *Admin* then sends this event to its *DistributionEnabledPort*, which forwards the event to the attached remote *DistributionEnabledPort*s. Each receiving *DistributionEnabledPort* delivers the event to its attached *Admin*, which reconstitutes functional modules (i.e., components and connectors) from the event, and invokes the *IArchitecture*'s *add* and *weld* methods to insert the modules into the local configuration.

7.1.9.2 Stateful Mobility

The technique described above provides the ability to transfer code between a set of hosts. As such, the stateless technique is useful for performing initial deployment of a set of components and connectors onto target hosts. In cases when runtime migration of architectural elements is required, the migrant element's state needs to be transferred along with the compiled image of that element. Additionally, the migrant element may need to be disconnected and deleted from the source host (if the element's replication is not desired or allowed). We provide two complementary techniques for stateful mobility: serialization-based and event stimulus-based.

The serialization-based technique relies on the existence of Java-like serialization mechanisms in the underlying PL. Instead of sending a set of compiled images, the local *Admin* possibly disconnects and removes the (active) migrant elements from its local subsystem (using the *IArchitecture*'s *unweld* and *remove* methods), serializes each migrant element, and packages them into a set of *ExtensibleEvent*s, which are then forwarded by the *DistributionEnabledPort*. *Admin*s on each receiving host reconstitute the architectural elements from these events and attach them to the appropriate locations in their local subsystems.

In cases where the serialization-like mechanism is not available (e.g., Java KVM), we use the event stimulus-based technique: the compiled image of the architectural element(s) to be migrated is sent across a network using the stateless technique. In addi-

tion, each event containing a migrant element is accompanied by a set of application-level events needed to bring the state of the migrant element to a desired point in its execution (see [41] for details of how such events are captured and recorded). Once the migrant architectural element is received at its destination, it is loaded into memory and added to the architecture, but is not attached to the running subsystem. Instead, the migrant element is stimulated by the application-level events sent with it. Any events the migrant element issues in response are not propagated, since the element is detached from the rest of the architecture. Only after the migrant architectural element is brought to the desired state is it welded and enabled to exchange events with the rest of the architecture. While less efficient than the serialization-based migration scheme, this is a simpler technique, it is PL-independent, and it is natively supported in Prism-MW.

Note that our approach works for a component that is independent of any low-level local resources. A component that uses particular local resources, which cannot be serialized and migrated to a different host, will not work with our approach. However, a component that relies on its local interaction with other Prism-MW *Component*s via basic ports will still be capable of communicating with those components by installing the appropriate distribution extensions on its ports.

7.1.10 Monitoring

In support of monitoring at the architecture level, Prism-MW provides the *AbstractMonitor* class associated through the *Scaffold* with every *Brick* (shown in Figure 7-12). This allows for autonomous, active monitoring of a *Brick*'s runtime behavior. We have provided several implementations of the *AbstractMonitor* class. For example, *EvtFrequencyMonitor* records the frequencies of different events the associated *Brick* sends, while *NetworkReliabilityMonitor* records the reliability of connectivity between its associated *DistributionEnabledPort* and other, remote *DistributionEnabledPort*s using a common "pinging" technique. We have also leveraged the work presented in [55] in the development of measurement and monitoring facilities that affect system's energy consumption (recall Section 5.2).

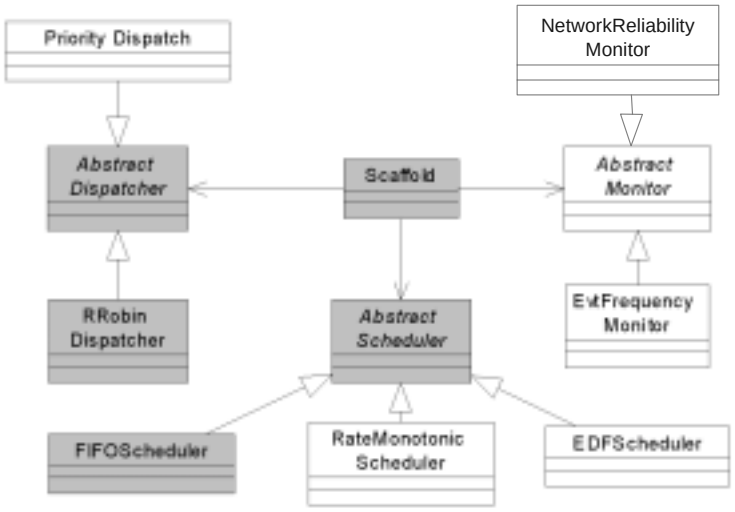


Figure 7-12: Scaffold extensions.

7.1.11 Support for Architectural Styles

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems' requirements [49,56]. Therefore, a middleware platform used to implement such architectures would need to support multiple styles. Prism-MW's design can be leveraged to support a number of distributed systems styles [11], which are likely to be useful in the Prism setting [36]. In this section we describe how Prism-MW can be configured to support different architectural styles using the mechanism introduced in Section 7.1.4 and leveraged in the extensions described earlier.

In order to effectively support architectural styles, Prism-MW should be configured to provide the following:

1. the ability to distinguish among different architectural elements of a given style (e.g, distinguishing `Clients` from `Servers` in the client-server style);
2. the ability to specify the architectural elements' stylistic behaviors (e.g., `Clients` block after sending a request while `C2Components` send requests asynchronously);
3. the ability to specify the rules and constraints that govern the architectural elements' valid configurations (e.g., disallowing `Clients` from connecting to each other in the client-server style, or allowing a `Filter` to connect only to a `Pipe` in the pipe-and-filter style); and

4. the ability to use multiple architectural styles within a single application.

We have leveraged Prism-MW's extensibility to support the above requirements. The following extensibility properties of Prism-MW have been used to satisfy the requirements:

- *Brick* has an attribute that identifies its style-specific type. The value of this variable corresponds to a given architectural style element, e.g., `Client`, `Server`, `Pipe`, `Filter`, and so on. The default value of this variable is *Null*, corresponding to the "null" style supported by Prism-MW's core. The association of *Brick* with its style-specific type satisfies our first requirement by enabling identification of different architectural elements.
- *ExtensibleConnector* has an associated implementation of the *AbstractHandler* class to support style-specific event routing policies (see Figure 7-13a). For example, `Pipe` forwards data unidirectionally, while a `C2Connector` uses bidirectional event broadcast. This partially satisfies the second requirement by allowing tailoring of a connector's style-specific behavior.
- *ExtensibleComponent* has an associated implementation of the *AbstractComponentSynchronism* class to provide synchronous component interaction (see Figure 7-13b). The default, asynchronous interaction is provided by Prism-MW's core. This partially satisfies the second requirement by allowing one to tailor a component's style-specific behavior (e.g., a `Client` blocks after it sends a request to a `Server` and unblocks when it receives a response).

- *ExtensiblePort* has an associated implementation of the *AbstractDistribution* class to support inter-process communication (see Figure 7-13c). This partially satisfies the second requirement by supporting architectural styles that require distribution (e.g., a `Server` may serve many distributed `Clients`).
- *ExtensibleArchitecture* has an associated implementation of the *AbstractTopology* class to ensure the topological constraints of a given style (see Figure 7-13d). For example, in the client-server style, `Client`s can connect to `Server`s, but two `Client`s cannot be connected to one another. Each time an *ExtensibleArchitec-*

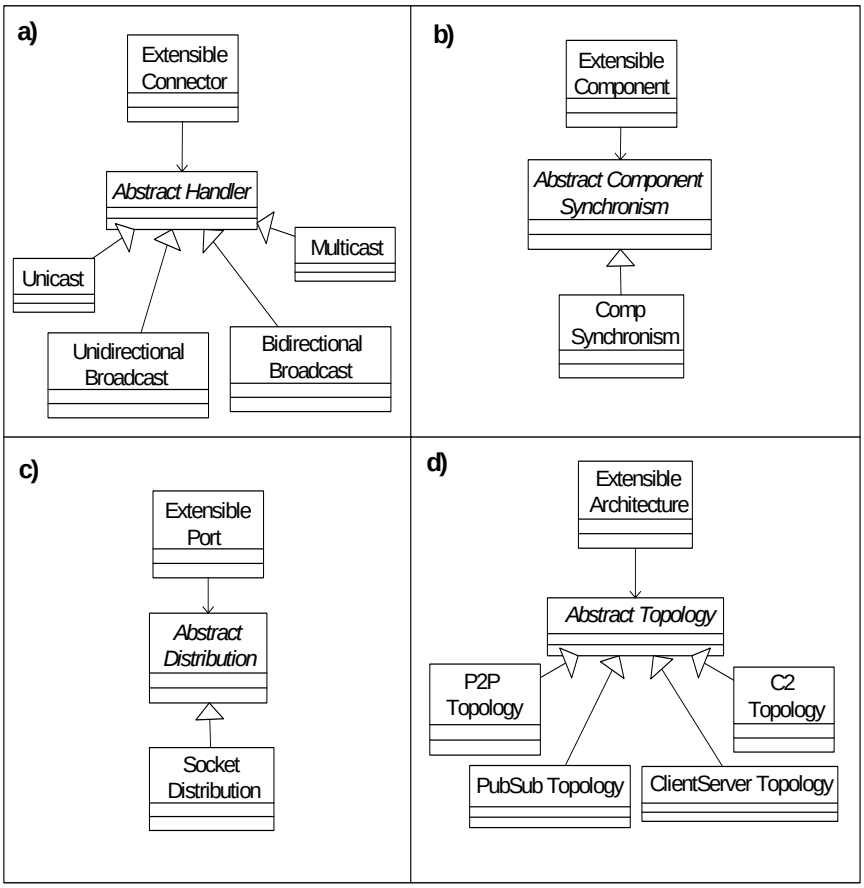


Figure 7-13: Prism-MW's support for architectural styles.

ture's *weld* (or *unweld*) method is invoked, the appropriate implementation of the *AbstractTopology* ensures that the topological rules of a given style are preserved. As a result, it either performs the *weld* (or *unweld*) operation or raises an exception. This satisfies the third requirement stated at the beginning of this section by allowing for the specification and modification of valid configurations of architectural elements. Note that while invoking *IArchitecture*'s *add* method results in the addition of a component/connector to the *Architecture* (or *ExtensibleArchitecture*) object, it does not affect the system's architectural style until it is *weld*ed. Similarly, the component/connector cannot be removed before *AbstractTopology*'s *unweld* method is called, which ensures that the removal will not undermine the system's architectural style.

- *ExtensibleArchitecture* implements the *IComponent* interface, thereby allowing hierarchical composition of components (see Figure 7-1). Each hierarchical component is internally composed of subarchitectures that can adhere to different architectural styles. This satisfies the fourth requirement by allowing combinations of different styles in a single system.

To produce a style-specific architectural element, the developer instantiates the corresponding *Extensible* class (recall Figure 7-13) and sets the desired stylistic behavior by installing the appropriate extensions on it. To simplify this task, we have provided a *StyleFactory* utility class (shown partially in Figure 7-14) that can automatically generate style-specific architectural elements.

For illustration we will discuss how we have developed support for the client-server style. In this style, a client is a triggering process; a server is a reactive process. Clients make requests that trigger reactions from servers. Thus, a client initiates activity at times of its choosing, and then blocks until its request has been serviced. On the other hand, a server waits for requests to be made and then reacts to them.

Both `Client` and `Server` in Prism-MW are represented using an *ExtensibleComponent*. However, `Client` uses an implementation of *AbstractComponentSynchronism* which overrides the default non-blocking behavior of a component. `Client`s make synchronous requests by blocking until the corresponding acknowledgement reply comes back. An acknowledgement reply indicates the completion of the requested operation on the `Server`. A `Client` can have one or more request ports through which it sends request events to the `Server`s, but cannot have any reply ports. A `Server` component can have one or more reply ports through which it sends reply events back to the requesting `Client`s. Prism-MW supports client-server applications that reside in one or more address spaces.

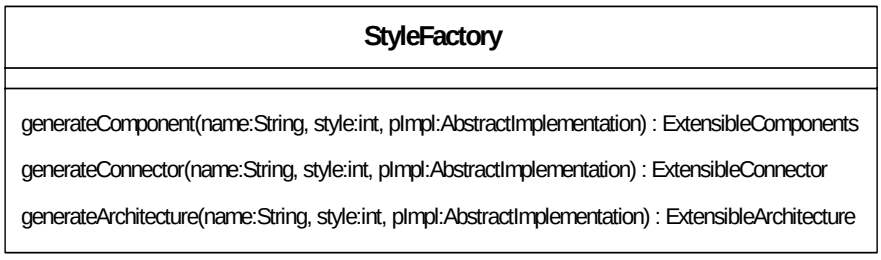
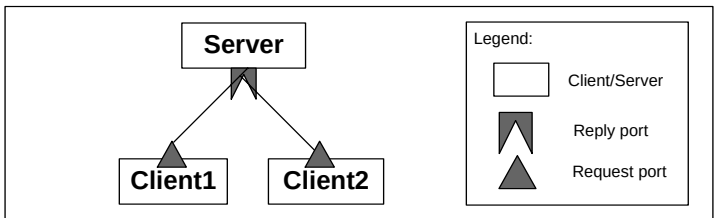


Figure 7-14: Partial API of StyleFactory.

Figure 7-15 shows a simple client-server style architecture, and the corresponding code in Prism-MW. A client-server architecture is composed of an *ExtensibleArchitecture* with the *ClientServerTopology* implementation of the *AbstractTopology. ClientServerTopology* enables welding of `Client`s and `Server`s while enforcing the topological rules (e.g., disallowing the welding of two `Client`s).

We have implemented a number of additional styles (see Figure 7-13) in a similar manner. Each style required, on average, the addition of 80 new SLOC to Prism-MW. Changes to Prism-MW were localized to new implementations of *AbstractHandler* and *AbstractTopology* classes. On average, the described extensions for each style required less than 1 person-hour of effort, including testing. As the number of sup-




```
class ClientServer {
  static public void main(String argv[]) {
    ExtensibleArchitecture arch =
    StyleFactory.generateArchitecture("arch", CLIENT_SERVER_ARCH);

    // create components here
    AbstractImplementation client1Impl = new Client1();
    ExtensibleComponent client1 =
    StyleFactory.generateComponent("client1", CLIENT, client1Impl);
    AbstractImplementation client2Impl = new Client2();
    ExtensibleComponent client2 =
    StyleFactory.generateComponent("client2", CLIENT, client2Impl);
    AbstractImplementation serverImpl = new Server();
    ExtensibleComponent server =
    StyleFactory.generateComponent("server", SERVER, serverImpl);

    // add components to the architecture
    arch.add(client1);
    arch.add(client2);
    arch.add(server);

    // establish the interconnections
    arch.weld(client1,server);
    arch.weld(client2,server);

    arch.start(); // start the architecture

  }
}
```

Figure 7-15: Client-Server style example.

ported styles in Prism-MW grows, we expect that implementing a new style would require even less effort since existing style implementations (e.g., different connector routing policies) may be reused.

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems' requirements. Prism-MW supports the use of multiple architectural styles in a single application by leveraging hierarchical composition. *Extensible-Architecture* implements the *IComponent* interface (recall Figure 7-1) and therefore allows its instances to be used as hierarchical components. The application architecture that contains multiple styles is then composed as a configuration of several hierarchical components (with their own internal architectures), each of which may adhere to a different architectural style.

7.2 Deployment Modeling and Analysis Environment

DeSi is a visual deployment exploration environment that supports specification, manipulation, and visualization of deployment architectures for large-scale, highly distributed systems. By leveraging DeSi, an architect is able to enter desired system parameters into the model, and also to manipulate those parameters and study their effects (shown in Figure 7-17). For example, the architect is able to use a graphical environment to specify new architectural constructs (e.g., components, hosts), parameters (e.g., network bandwidth, host memory), and values for the parameters (e.g., available memory on a host is 1MB). The architect may also specify constraints.

Example constraints are the maximum and minimum available resources, the location constraint that denotes the hosts that a component cannot be deployed on, and the collocation constraint that denotes a subset of components that should not be deployed on the same host. DeSi also provides a visualization environment for graphically displaying the system's monitored data, deployment architecture, and the results of analysis (shown in Figure 7-18).

Figure 7-16 shows the high-level architecture of DeSi. The centerpiece of the architecture is a rich and extensible *Model*, which in turn allows extensions to the *View* (used for model visualization) and *Controller* (used for model manipulation) subsystems.

Model. DeSi's *Model* subsystem is reactive and accessible to the *Controller* via a simple API. The *Model* captures different system aspects in its four components: *SystemData*, UserPrefData, *GraphViewData*, and *AlgoResultData*. *SystemData* is the key part of the *Model* and represents the software system itself in terms of the architectural constructs and parameters: numbers of components and hosts, distribution of components across hosts, software and hardware topologies, and so on. *UserPrefData* contains the data for representing the users of the system, their QoS preferences in terms of utility, and the provisioned user-level services. *GraphViewData* captures the information needed for visualizing a system's deployment architecture: graphical (e.g., color, shape, border thickness) and layout (e.g., juxtaposition, movability, con-

tainment) properties of the depicted components, hosts, and their links. Finally, *AlgoResultData* provides a set of facilities for capturing the outcomes of the different deployment estimation algorithms: estimated deployment architectures (in terms of component-host pairs), achieved utility, the impact on various QoS dimensions, algorithm's running time, estimated time to effect a redeployment, and so on.

View. DeSi's *View* subsystem exports an API for visualizing the *Model*. The current architecture of the *View* subsystem contains three components—*GraphView, UserPrefView,* and *TableView. GraphView* is used to depict the information provided by the *Model*'s SystemData and *GraphViewData* components. Similarly, *UserPrefView* is used to depict the information provided by *UserPrefData* component. *TableView* is intended to support a detailed layout of system parameters and deployment estimation

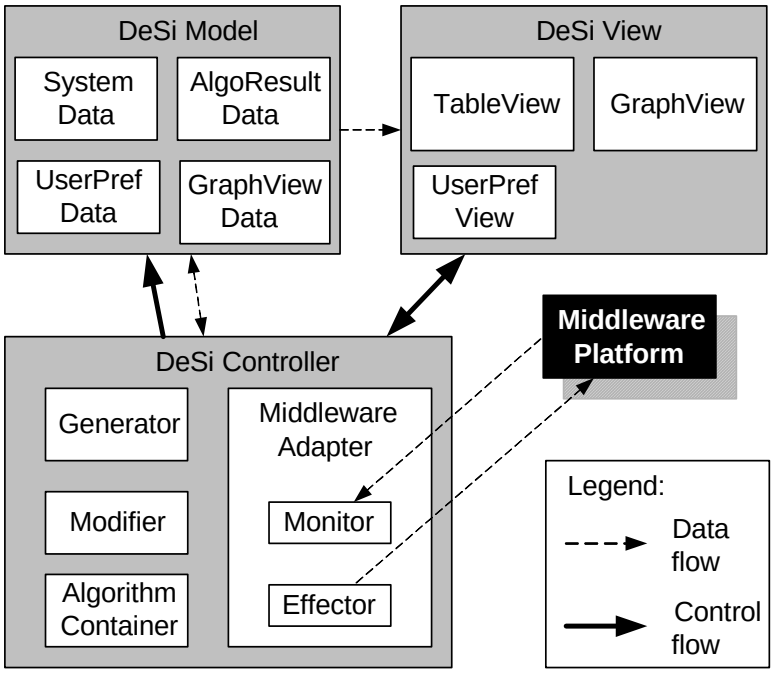


Figure 7-16: DeSi's architecture.

algorithms captured in the *Model*'s *SystemData* and *AlgoResultData* components. The decoupling of the *Model*'s and corresponding *View*'s components allows one to be modified independently of the other. For example, it allows us to add new visualizations of the same models, or to use the same visualizations on new, unrelated models, as long as the component interfaces remain stable.

Controller. DeSi's *Controller* subsystem comprises four components. The *Generator*, *Modifier*, and *AlgorithmContainer* manage different aspects of DeSi's *Model* and *View* subsystems, while the *MiddlewareAdapter* component provides an interface to a, possibly third-party, system implementation, deployment, and execution platform (depicted as a "black box" in Figure 7-16). The *Generator* component takes as its input the desired number of hardware hosts, software components, and a set of ranges for system parameters (e.g., minimum and maximum network reliability, component interaction frequency, available memory, and so on). Based on this information, *Generator* creates a specific deployment architecture that satisfies the given input and stores it in *Model* subsystem's *SystemData* component. The *Modifier* component allows fine-grain tuning of the generated deployment architecture (e.g., by altering a single network link's reliability, a single component's required memory, and so on). Finally, the *AlgorithmContainer* component invokes the selected redeployment algorithms (recall algorithms presented in Chapter 6) and updates the *Model*'s *AlgoResultData*. In each case, the three components also inform the *View* subsystem that the

Model has been modified; in turn, the *View* pulls the modified data from the *Model* and updates the display.

The above components allow DeSi to be used to automatically generate and manipulate large numbers of hypothetical deployment architectures. The *MiddlewareAdapter* component, on the other hand, provides DeSi with the same information from a running, real system. *MiddlewareAdapter*'s *Monitor* subcomponent captures the run-time data from the external *MiddlewarePlatform* and stores it inside the *Model*'s *SystemData* component. *MiddlewareAdapter*'s *Effector* subcomponent is informed by the *Controller*'s *AlgorithmContainer* component of the calculated (improved) deployment architecture; in turn, the *Effector* issues a set of commands to the *MiddlewarePlatform* to modify the running system's deployment architecture. The details of this process are further illuminated below.

7.2.1 DeSi's Implementation

DeSi has been implemented within the Eclipse platform using Java 1.4. DeSi's implementation adheres to its MVC architectural style. In this section, we discuss (1) the implementation of DeSi's extensible model, (2) the visualizations currently supported by DeSi, and (3) its capabilities for generating deployment scenarios, assessing a given deployment, manipulating system parameters and observing their effects, and estimating redeployments that result in improved system properties.

7.2.1.1 Model Implementation

The implementations of the *SystemData*, *UserPrefData*, and *AlgoResultData* components of DeSi's *Model* are simple: each one of them is implemented as a set of classes, with APIs for accessing and modifying the stored data. For example, *AlgoResultData*'s implementation provides an API for accessing and modifying the array containing the estimated deployment architecture, and a set of variables representing the achieved QoS properties, algorithm's running time, and estimated time to change a given system from its current to its new deployment.

The *GraphViewData* component contains all the persistent data necessary for maintaining the graphical visualization of a given deployment architecture. It keeps track of information such as figure shapes, colors, placements, and labels that correspond to hardware hosts, software components, and software and hardware links. In our implementation of *GraphViewData*, each element of a distributed system (host, component, or link) is implemented via a corresponding *Figure* class that maintains this information (e.g., each host is represented as a single instance of the *HostFigure* class). Components and hosts have unique identifiers, while a link is uniquely identified via its two end-point hosts or components.

The *GraphViewData* component's classes provide a rich API for retrieving and modifying properties of individual components, hosts, and links. This allows easy run-time modification of virtually any element of the visualization (e.g., changing the line

thickness of links). Furthermore, the information captured inside *GraphViewData* is not directly tied to the properties of the model captured inside *SystemData*. For example, the *color* property of *HostFigure* can be set to correspond to the amount of available memory or to the average reliability with other hosts in the system.

7.2.1.2 View Implementation

The *TableView* component of DeSi's *View* subsystem displays the *Deployment Control Window*, shown in Figure 7-17. This window consists of five sections, identified by panel names: *Input*, *Constraints*, *Algorithms*, *Results*, and *Tables of Parameters*.

The *Input* section allows the user to specify different input parameters, for example: numbers of components and hosts; ranges for component memory, frequency and event size; and ranges for host memory, reliability and bandwidth. For centralized deployment scenarios we provide a set of text fields for specifying the properties of the central host. The *Generate* button on the bottom of this panel results in the (random) generation of a single deployment architecture that satisfies the above input. Once the parameter values are generated, they are displayed in the *Tables of Parameters* section, which will be discussed in more detail below.

The *Constraints* section allows specification of different conditions for component (co-)location: (1) components that must be deployed on the same host, (2) components that may not be deployed on the same host, and (3) components that have to be on specific host(s).

The *Algorithms* section allows the user to run different redeployment estimation algorithms by clicking the corresponding algorithm's button. There are three provided options for running the algorithms: (1) *Just run*, which runs the algorithm and displays the result in the *Results* panel, (2) *Run and preview*, which runs the algorithm displays the results both in the *Results* panel and in the graphical view (discussed further below), and (3) *Run and effect*, which, in addition to actions described in option 2, also updates *SystemData* to set the current deployment to the output of the selected algorithm. The latter action can be reversed by clicking on the *Revert to previous deployment* button.

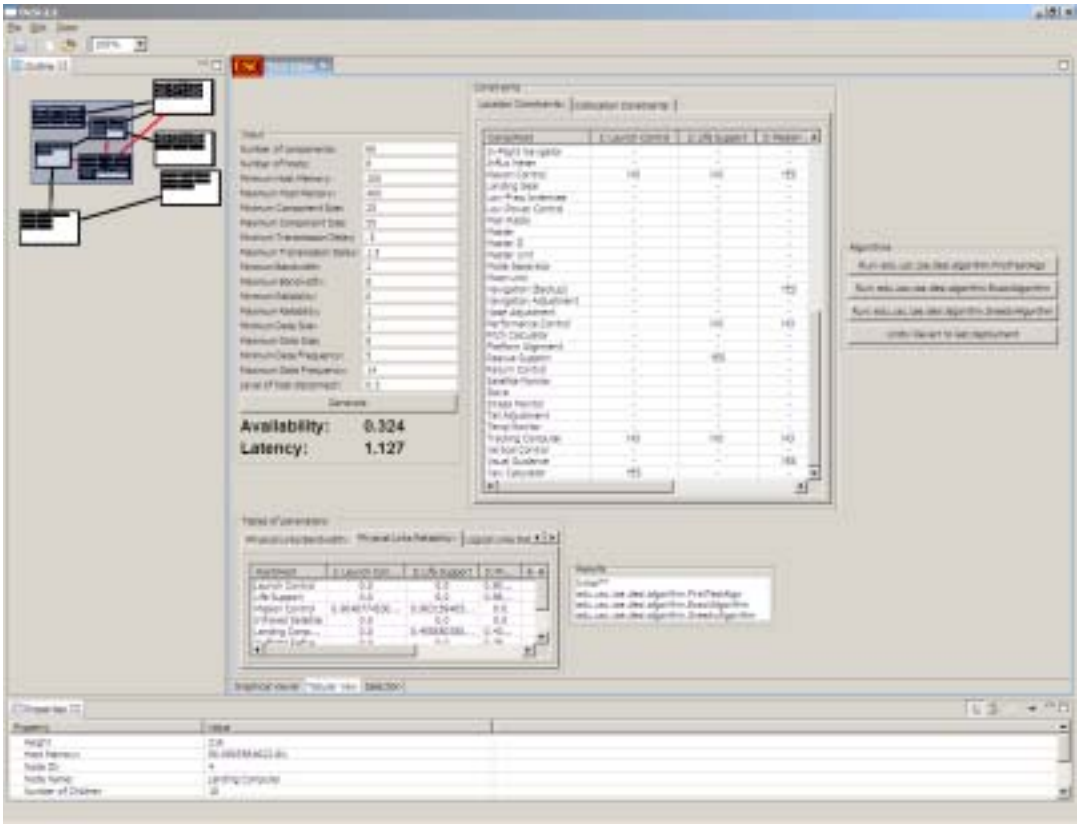
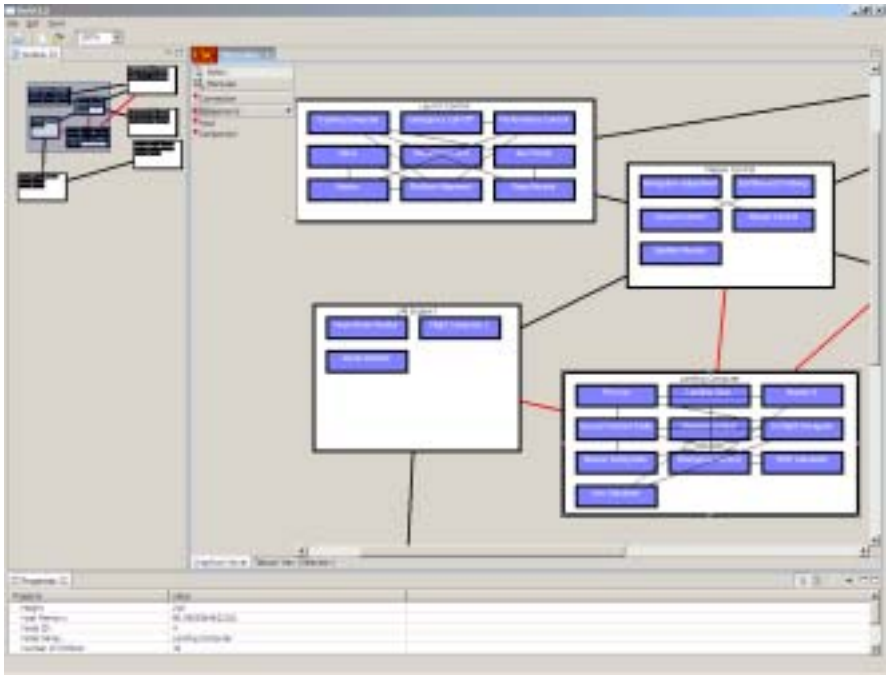


Figure 7-17: DeSi's editable tabular view of the system's deployment architecture.

a)



b)

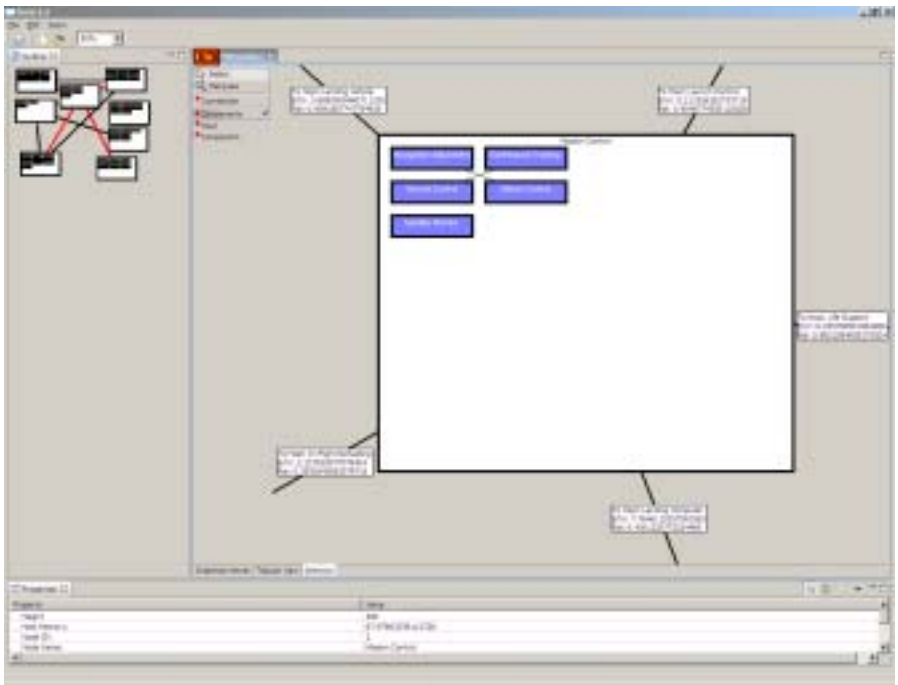


Figure 7-18: DeSi's graphical view of a system's deployment architecture: (a) zoomed out vie showing multiple hosts; (b) zoomed in view of the same architecture.

We have provided a benchmarking capability to compare the performance of various algorithms. The user can specify the number of times the algorithms should be invoked. Then, the user triggers via the *Benchmark* button a sequence of automatic random generations of new deployment architectures and executions of all algorithms.

The *Results* section displays the outcomes of different algorithms. For each algorithm, the output consists of (1) a new mapping of components to hosts, (2) the system's achieved QoS properties, (3) the running time of the algorithm, and (4) the estimated time to effect the new deployment.

Finally, the *Table of Parameters* section provides an editable set of tables that display different system parameters. The user can view and edit the desired table by selecting

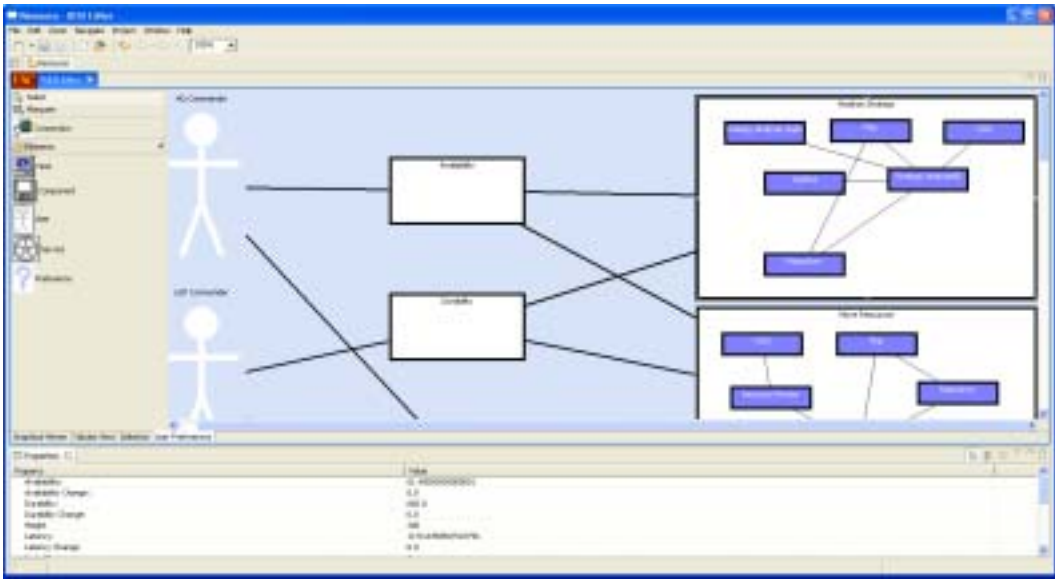


Figure 7-19: DeSi's graphical view of system users' QoS preferences.

the appropriate tab from the tab list. The editable tables support fine-tuning of system parameters.

The goals of the *GraphView* and *UserPrefView* component of DeSi's *View* subsystem were to (1) allow the architect to quickly examine the complete deployment architecture of a given system as well as the various users' QoS preferences, (2) provide scalable and efficient displays of deployment architectures with large numbers of components and hosts, and (3) be platform independent. To this end we used the Eclipse environment's GEF plug-in, which consists of a library for displaying different shapes, lines, and labels and a facility for run-time editing of the displayed images.

GraphView and *UserPrefView* provide a simple API for displaying their elements, such as hosts, components, users, and links. For example, displaying two connected hosts requires two consecutive calls of the *createHost* method, followed by the *createH2HLink* method. Figure 7-18 illustrates a sample deployment architecture of 100 components and 8 hosts. Network connections between hosts are depicted as thick solid lines, while the interactions between software components are represented as thin solid lines. Figure 7-19 illustrates DeSi's graphical view of users' QoS preferences. Users' QoS preferences are visualized in three tiers: on the left side are the users of the system, in the middle are the various QoS preferences, and finally on the right side are the provisioned services and their mapping to the underlying software

architecture. Users mapping to QoS preferences, and QoS preferences mapping to services are depicted via the solid lines connecting them.

For systems with large numbers of hosts and components, visualizing the system and its connectivity becomes a challenge. For this reason, *GraphView* supports zooming in and out (see Figure 7-18), and provides the ability to "drag" hosts and components on-screen, in which case all relevant links will follow them. Alternatively, one can configure DeSi to not display connections between components residing on different hosts. In this scheme, if remote connections exist, the components will have thicker borders. Consequently, a thin border on a component denotes that it does not communicate with remote components.

GraphView and *UserPrefView* have several other features that allow users to easily visualize and reason about a given deployment architecture. A user can get at-a-glance information on any of the elements in the system. Selection of a single graphical object displays its information in the properties sheet at the bottom of the window (see Figure 7-18 and Figure 7-19). The displayed information can easily be changed or extended through simple modifications to *GraphView* to include any (combination) of the information captured in the *SystemData* component. Detailed information about any of the modeling elements (e.g. host, component) can be displayed by double-clicking on the corresponding graphical object. The *DetailWindow* for a host, shown in Figure 7-18b, displays the host's properties in the property sheet, the com-

ponents deployed on the host, the host's connections to other hosts, and the reliabilities and bandwidths of those connections. Similarly, the property sheet for a selected service, shown in Figure 7-19, displays the service's properties (e.g., availability, latency) and the components that participate in provisioning it.

7.2.1.3 Controller Implementation

The implementation of DeSi *Controller*'s *Generator* component provides methods for (1) generating random deployment problems, (2) producing a specific (initial) deployment that satisfies the parameters and constraints of a generated problem, and (3) updating DeSi *Model*'s *SystemData* and *UserPrefData* classes accordingly. This capability allows us to rapidly compare the different deployment algorithms discussed in Chapter 6. *Generator* is complemented by the class implementing the *Controller*'s *Modifier* component. *Modifier* provides facilities for fine-tuning system parameters (also recall the discussion of editable tables in Section 7.2.1.2), allowing one to assess the sensitivity of a deployment algorithm to specific parameters in a given system.

Each deployment algorithm in DeSi *Controller*'s *AlgorithmContainer* component is encapsulated in its own class, which extends the *AbstractAlgorithm* class. *AbstractAlgorithm* captures the common attributes and methods needed by all algorithms (e.g., *calculateQoSDimension*, *estimateRedeploymentTime*, and so on). Each algorithm class needs to implement the abstract method *execute*, which returns an object of type

AlgorithmResult. A bootstrap class called *AlgorithmInvoker* provides a set of static methods that instantiate an algorithm object and call its *execute* method. The localization of all algorithm invocations to one class aids DeSi's separation of concerns and enables easy addition of new (kinds of) algorithms.

Finally, DeSi provides the *Monitor* and *Effector* components which can interface with a middleware platform to capture and display the monitoring data from a running distributed system, and invoke the middleware's services to enact a new deployment architecture. This facility is independent of any particular middleware platform and only requires that the middleware be able to provide monitoring data about a distributed system and an API for modifying the system's architecture. DeSi does not require a particular format of the monitoring information or system modification API; instead, the goal is to employ different wrappers around the *Monitor* and *Effector* components for each desired middleware platform. In the next section we discuss in detail an adapter we have developed for the integration of DeSi with Prism-MW.

7.3 Tool Integration

To integrate DeSi with Prism-MW, we have wrapped *Monitor* and *Effector* components of DeSi (shown in the *Middleware Adapter* of Figure 7-16) as Prism-MW components that are capable of receiving *Event*s containing the monitoring data from Prism-MW's *Admin*, and issuing events to the *Admin* to enact a new deployment architecture. Once the monitoring data is received, DeSi updates its own system

model. This results in the visualization of an actual system, which can now be analyzed and its deployment improved by employing different algorithms. Once the outcome of an algorithm is selected by the *Analyzer*, DeSi issues a series of events to Prism-MW's *Admin* to update the system's deployment architecture.

Figure 7-20 depicts an application running on top of Prism-MW with the monitoring and deployment facilities instantiated and associated with the appropriate architectural constructs. A meta-level *Admin* (recall Section 7.1.9) on any device is capable of accessing the monitoring data of its local components via its reference to *Architecture*. In order to minimize the time required to monitor the system, monitoring is performed in short intervals of adjustable duration. Once the monitored data is stable (i.e., the difference in the data across a desired number consecutive intervals is less

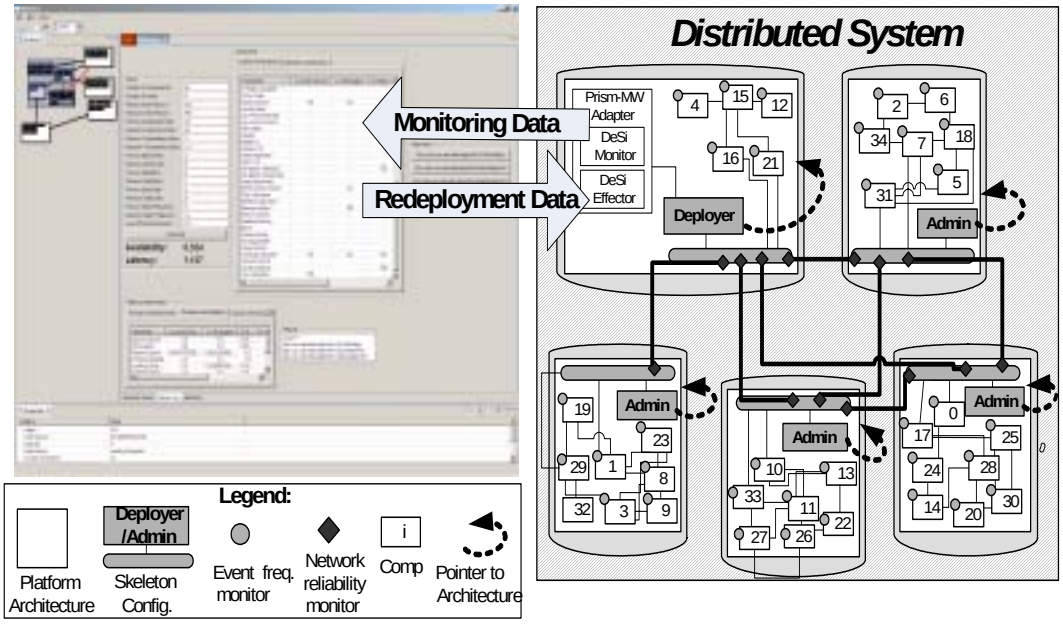


Figure 7-20: An example of a distributed system running on top of Prism-MW that is monitored and (re)deployed in collaboration with DeSi.

than an adjustable value *e*), the *Admin* sends the description of its local deployment architecture and the monitored data (e.g., event frequency, network reliability, etc.) in the form of serialized Prism-MW *Event*s to a central *Admin*, called a *Deployer*. Our assessment of Prism-MW's monitoring support suggests that monitoring on each host may induce as little as 0.1% and no greater than 10% in memory and computation overheads. Note that Prism-MW's extensible design allows for addition of new monitoring capabilities via new implementations of *IMonitor* interface.

Once a new deployment architecture is selected by one of DeSi's algorithms based on the monitoring data supplied by Prism-MW, DeSi informs the *Deployer* of the desired deployment architecture, which now needs to be effected. The effecting process requires coordination among different hosts (e.g., ensuring architectural consistency, synchronization, etc.). Prism-MW's support for coordination is implemented in its *Admin* and *Deployer Component*s:

- The *Deployer* sends events to inform *Admin*s of their new local configurations, and of the remote locations of software components required for performing changes to each local configuration.
- Each *Admin* determines the difference between its current and new configurations, and issues a series of events to remote *Admin*s requesting the components that are to be deployed locally. If devices that need to exchange components are

not directly connected, the relevant request events are sent to the *Deployer*, which then mediates their interaction.

- Each *Admin* that receives an event requesting its local component(s) to be deployed remotely, detaches the required component(s) from its local configuration, serializes them, and sends them as a series of events via its local *DistributionConnector* to the requesting device.
- The recipient *Admins* reconstitute the migrant components from the received events and invoke the appropriate methods on its *Architecture* object to attach the received components to the local configuration.

Other coordination techniques have also been incorporated into Prism-MW in a similar manner via different implementations of the *Deployer* and *Admin*. In fact, the coordination policy and the order in which components are deployed/started may be application specific. For example, we have developed an alternative redeployment mechanism, where DeSi determines a complete set of commands required for redeploying and reconfiguring the system based on the model of system's deployment architecture. A command corresponds to a Prism-MW event type, and may contain a number of parameters that determine the type of task that should be performed remotely by *Deployer/Admin*. DeSi also determines the order in which the commands should be executed remotely to ensure architectural consistency. Commands are sent to the appropriate *Deployer/Admin*, which sends an acknowledgement back to DeSi once the command is executed successfully. This mechanism ensures a safe and synchronized (re)deployment of the system.

CHAPTER 8: Evaluation

In this chapter we provide an evaluation of the framework with respect to the objectives and hypotheses that have guided this dissertation research. First, we present the results of benchmarking the algorithms, and compare them in terms of their accuracy and performance in solving our problem. Since the framework is geared towards embedded, resource-constrained, and large-scale applications, it is important for the implementation support to be highly efficient and scalable. We therefore provide evaluation of Prism-MW in terms of these two properties. Furthermore, since one of the motivations behind the development of the framework has been to promote cross-evaluation, and reuse across different application scenarios, we will also evaluate the tool-support's tailorability, extensibility, and ability to explore the large space of deployments. Finally, we provide an overview of our experience with applying the framework to two real distributed applications that we have developed in collaboration with external development organizations.

8.1 Algorithms

In this section we evaluate the results of the algorithms discussed in Chapter 6 for several instances of the scenario introduced in Section 5.2. Recall that in this scenario we tailored the framework with four QoS dimensions. The evaluation of the algorithms is focused on the five hypotheses that have motivated this research.

8.1.1. Evaluation Setup

We leveraged DeSi's hypothetical deployment generation capability to create the scenario instances. In the generation of deployment scenarios all system parameters are populated with randomly generated data within a specified range, and an initial deployment of the system that satisfies all the constraints is provided. DeSi thereby enabled us to evaluate our approach on a large number of generated examples.

Figure 8-1 shows the input into DeSi for the generation of example scenarios and benchmarks. The values in Figure 8-1 represent the allowable ranges for each system parameter. The numbers of hosts, components, services, and users vary across the benchmark tests and are specified in the description of each test. Note that both the framework and DeSi are independent of the unit of data used for each system parameter. For example, in the case of transmission delay, neither the framework nor DeSi depend on the unit of time (*s*, *ms*, etc.). It is up to the system architect to ensure that the right units and appropriate ranges for the data are supplied to DeSi. After the deployment scenario is generated, DeSi simulates users' preferences by generating hypothetical desired rates of change (*qosRate*) and desired utilities (*qosUtil*) for the QoS dimensions of each service. While users

$hostMem \in [10,30]$, hostEnrCons $\in [1,20]$,
$compMem \in [2,8]$, opcodeSize $\in [5,500]$,
$freq \in [1,10]$, $evtSize \in [10,100]$,
$bw \in [30,400]$, enc $\in [1,512]$, $rel \in [0,1]$,
$td \in [5,100]$, $commEnrCons \in [50,200]$
prob. a comp. is used by a service : 0.5
prob. a service is used by a user : 1
prob. a user has QoS pref. for a service : 1
$MinRate = 0.01$ and $MaxUtil = 1$

Figure 8-1: Input for DeSi's deployment scenario generation.

may only use and specify QoS preferences for a subset of services, we evaluate our algorithms in the most constrained (and challenging) case, where each user uses all the services and specifies a QoS preference for each service. Unless otherwise specified, the genetic algorithm used in the evaluation was executed with a single population of one hundred individuals, which were evolved one hundred times.

As mentioned in the hypotheses, we evaluate the accuracy of algorithms in large problems by comparing their results against the statistical average (or "most likely") deployment of a system. This algorithm, which we call *unbiased average*, generates different deployments by randomly assigning each component to a single host from a set of component's allowable hosts. If the generated deployment satisfies all the constraints, the utility of the produced deployment architecture is calculated. This process repeats a given number of times and the average utility of all valid deployments is calculated (*unbiased average*). The complexity of this algorithm is $O(n^2)$. In [40] we have experimentally shown that *unbiased average* does not significantly deviate from the actual average and thus signifies the quality of system's "most likely" deployment.

8.1.2 Improving Conflicting QoS Dimensions

Table 8-1 shows the result of running our algorithms on an example application scenario generated for the input of Figure 8-1 (with 12 components, 5 hosts, 8 services, and 8 users). The values in the first eight rows correspond to the percentage of

improvement over the initial deployment of each service. The ninth row shows the average improvement for each QoS dimension of all the services. Finally, the last row shows the final value of our objective function (*overallUtil).* The results demonstrate that, given a highly constrained system with conflicting QoS dimensions, the algorithms are capable of significantly improving QoS dimensions of each service. As discussed in Section 6.2, the MIP algorithm found the optimal deployment (with the objective value of 64 in this case).[6] The other algorithms also found good approximative solutions, which are within 20% of the optimal (recall hypothesis 1). Our experience with other generated scenario have been similar. For example, a benchmark of 20 randomly generated application scenarios showed average improvements of 73% for availability, 61% for latency, 51% for security, and 66% for energy consumption. The results indicate that the algorithms are capable of significantly improving multiple conflicting QoS dimensions.

Table 8-1: Results of an example scenario with 12C, 5H, 8S, and 8U.

	MIP				MINLP				Greedy				Genetic			
QoS	Avail.	Latency	Comm. Security	Energy Cons.	Avail.	Latency	Comm. Security	Energy Cons.	Avail.	Latency	Comm. Security	Energy Cons.	Avail.	Latency	Comm. Security	Energy Cons.
service 1	56%	-8%	18%	-8%	33%	2%	-5%	14%	24%	-8%	4%	-4%	16%	-2%	18%	-8%
service 2	93%	94%	97%	24%	91%	41%	32%	24%	83%	91%	62%	15%	93%	84%	35%	18%
service 3	39%	30%	22%	49%	32%	38%	11%	69%	39%	30%	22%	49%	19%	30%	22%	49%
service 4	215%	97%	302%	7%	215%	97%	302%	7%	165%	50%	220%	12%	180%	91%	150%	10%
service 5	59%	7%	25%	26%	23%	5%	39%	21%	43%	7%	19%	18%	29%	5%	35%	33%
service 6	99%	55%	37%	44%	83%	35%	45%	32%	99%	55%	37%	44%	99%	55%	37%	44%
service 7	91%	57%	20%	47%	97%	29%	44%	25%	91%	37%	14%	23%	91%	43%	4%	49%
service 8	43%	22%	7%	56%	41%	11%	-5%	72%	32%	21%	-10%	58%	13%	51%	7%	72%
Average	86%	44%	66%	30%	76%	32%	57%	33%	72%	35%	46%	26%	67%	44%	38%	33%
overallUtil	64				57				55				52			

6. An objective value has no absolute meaning, but is relative to the objective values of other deployments within the same application scenario.

8.1.3 Sensitivity to Users' Preferences and Service's Importance

Recall from Section 5.1 that the importance of a QoS dimension to a user (which we call *QoS importance*) is determined by the ratio of the user's *qosUtil* to *qosRate* for that dimension. Thus, QoS dimensions of services that on average have higher importance to the users typically show a greater degree of improvement (recall hypothesis). For example, in the scenario of Table 8-1, the users have placed a great degree of importance on service 4's availability and security. This is reflected in the results, which show a greater improvement of these dimensions for service 4 than their average improvement (e.g., in MIP's solution, availability of service 4 is improved by 215% and security by 302%; the average respective improvement of these two dimensions for all services was 86% and 66%). Note that, for this same reason, a few QoS dimensions of some services have degraded in quality, as reflected in the negative percentage numbers. These were not very important to the users and had to be degraded for improving other, more important QoS dimensions.

For validating hypotheses 2 and 3 we also performed other experiments with the objective of determining the sensitivity of the results produced by the algorithms to a user's preferences and a service's criticality. Note that the criticality of a service is defined as the amount of utility specified for that service. We ran the greedy algorithm on a slight variation of the scenario shown in Table 8-1, in which one user had specified more utility for the latency of service 1. The result showed a positive improvement in the latency property of service 1 of 19%. At the same time, the user

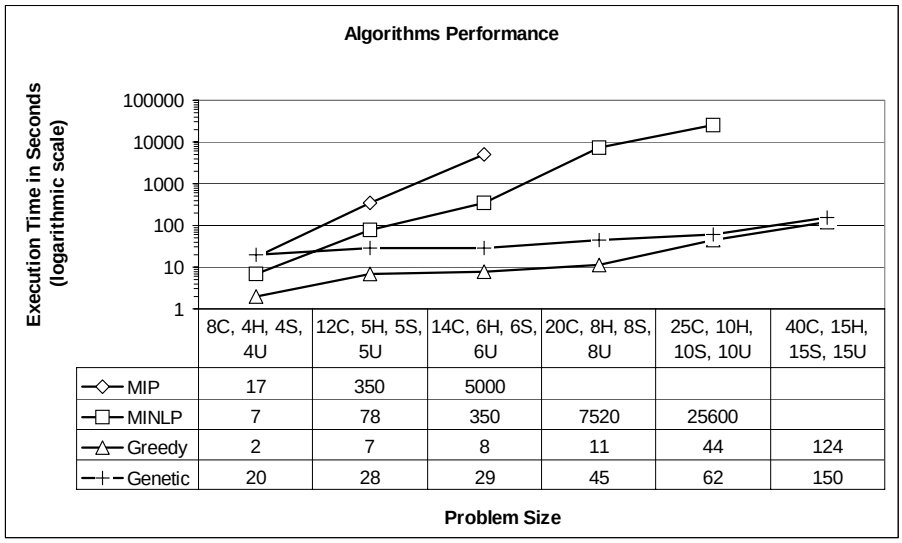
also showed an increase in the amount of utility accrued. We also performed over 50 similar experiments (i.e., running the greedy algorithm on two application scenarios that only differ with respect to a user's utility for a QoS dimension of a service), which showed that 1) the amount of improvement in the QoS dimension were either the same or more, 2) the amount of utility accrued by the user is the same or more. The consistent results could be attributed to the greedy algorithm's deterministic and step-wise maximization of the objective function.

We select as another illustration a benchmark of 20 randomly generated application scenarios, which showed average QoS improvements of 89% for services for which the users specified a QoS importance of 1 or more, and 34% for services for which the user specified a QoS importance of less than 1. Since the benchmark scenarios were generated for the input of Figure 8-1, QoS importance could have taken values in the range of 0 to 100. However, a value of 1 represents the expected (statistical average) value that QoS importance could have taken, and divides the data distribution into two halves.[7] In this comparison we have compared the half with higher QoS importance values against the half with lower QoS importance values, which verifies that improvements in QoS are based on users' QoS preferences (recall hypothesis 2).

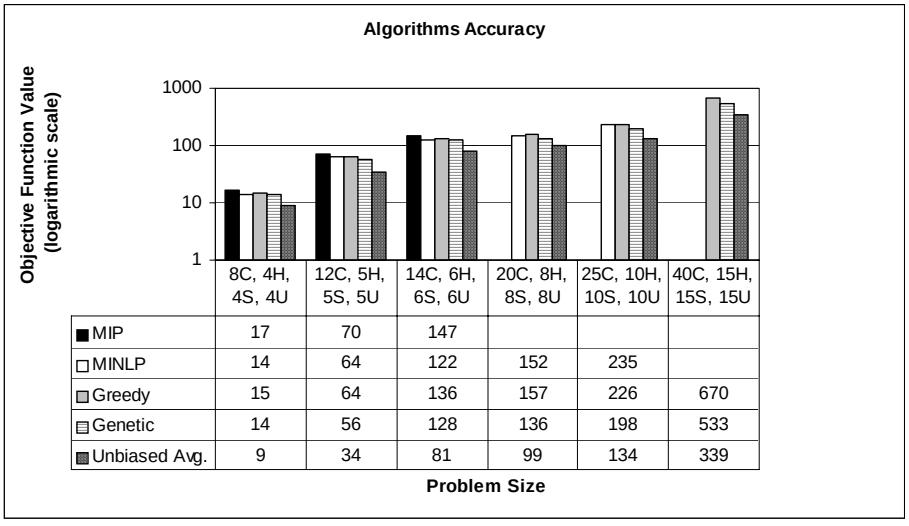
7. Recall from Figure 8-1 that minRate=0.01 and maxUtil=1. Based on the definitions of Figure 5.1, qosRate is randomly selected from 0.01 to 1, and qosUtil is randomly selected from 0 to 1. Thus, the expected average value for these two variables can be estimated to be 0.5. The expected average value for QoS importance is then equal to qosUtil/qosRate=0.5./0.5=1.

8.1.4 Performance and Accuracy

Figure 8-2 shows the comparison of the four algorithms in terms of performance (execution time) and accuracy (value of the objective function *overallUtil*). Note that the vertical axis is plotted logarithmically, thus the slope of lines may be deceiving. For each data point (shown on the horizontal axis with the number of components, hosts, services, and users), we created ten representative problems and ran the four



	8C, 4H, 4S, 4U	12C, 5H, 5S, 5U	14C, 6H, 6S, 6U	20C, 8H, 8S, 8U	25C, 10H, 10S, 10U	40C, 15H, 15S, 15U
MIP	17	350	5000			
MINLP	7	78	350	7520	25600	
Greedy	2	7	8	11	44	124
Genetic	20	28	29	45	62	150



	8C, 4H, 4S, 4U	12C, 5H, 5S, 5U	14C, 6H, 6S, 6U	20C, 8H, 8S, 8U	25C, 10H, 10S, 10U	40C, 15H, 15S, 15U
MIP	17	70	147			
MINLP	14	64	122	152	235	
Greedy	15	64	136	157	226	670
Genetic	14	56	128	136	198	533
Unbiased Avg.	9	34	81	99	134	339

Figure 8-2: Comparison of the four algorithms' performance and accuracy.

algorithms on them. The results correspond to the average values attained from these benchmarks. As shown, the high complexity of MIP and MINLP solvers made it infeasible to solve the larger problems. Comparing results of MINLP, greedy, and genetic algorithms against the optimal solution found by the MIP algorithm shows that all three approximative algorithms come within at least 20 percent of the optimal solution, which validates the first part of hypothesis 1. Furthermore, the solutions found by our optimization algorithms are at least more than 30% better than the solutions found by the unbiased average, which validates the second part of hypothesis 1. The results also corroborate that the greedy and genetic algorithms are capable of finding solutions that are on par with those found by state-of-the-art MINLP solvers. On the other hand, our greedy and genetic algorithms demonstrate much better performance than both MIP and MINLP solvers, and are scalable to very large problems. The MINLP solvers were unable to find solutions for approximately 20% of larger problems (beyond 20 components and 10 hosts). However, for a meaningful comparison of the benchmark results we are not including problems that could not be solved by the MINLP solvers in Figure 8-2.

8.1.5 Sensitivity to QoS Dimensions

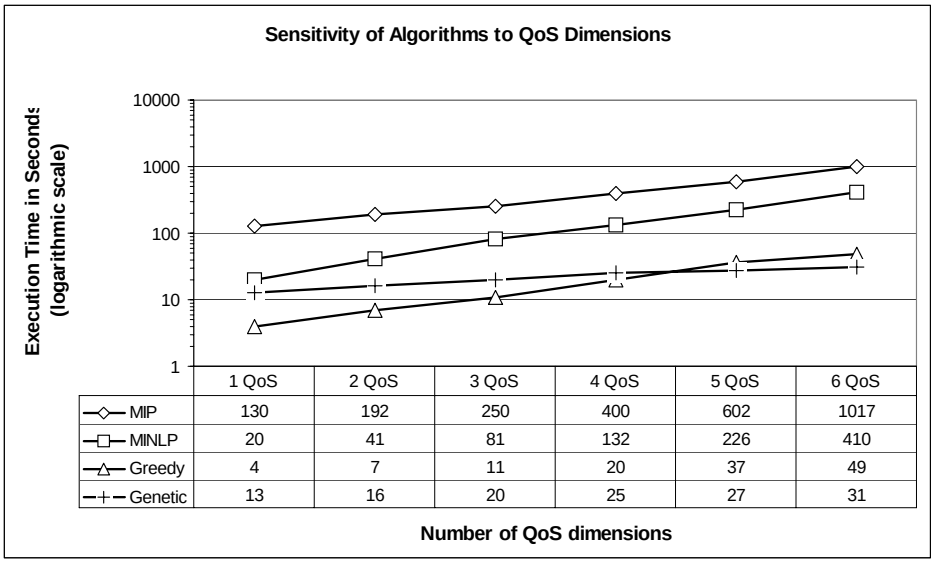
Figure 8-3 shows the sensitivity of each algorithm's performance to the number of QoS dimensions. We executed a deployment architecture of 12 components, 5 hosts, 5 services, and 5 users for varying numbers of QoS dimensions.[8] As expected, the

8. Recall from Section 5.2 that our framework allows arbitrarily specified QoS dimensions. This allowed us to simply introduce "dummy" QoS dimensions as needed in the course of our evaluation.

performance of all four algorithms is affected by the addition of new dimensions. However, the algorithms show different levels of sensitivity to the addition of new QoS dimensions. The genetic algorithm shows the least amount of degradation in performance. This is expected, since the analysis in Section 6.4, suggests that the complexity of the genetic algorithm increases linearly in the number of QoS dimensions, while the complexity of the greedy algorithm increases polynomialy. Even though we do not have access to the proprietary algorithms used by MIP and MINLP solvers, we can see that their performance also depends significantly on the addition of new QoS dimensions

8.1.6 Sensitivity to Heuristics

In this section we evaluate the heuristics we have introduced in the development of our algorithmic solutions. Figure 8-4 shows the effect of variable ordering on the per-



	1 QoS	2 QoS	3 QoS	4 QoS	5 QoS	6 QoS
MIP	130	192	250	400	602	1017
MINLP	20	41	81	132	226	410
Greedy	4	7	11	20	37	49
Genetic	13	16	20	25	27	31

Figure 8-3: Sensitivity of performance to QoS dimensions.

formance of the MIP algorithm. As discussed in Section 6.2 and shown in the results of Figure 8-4, specifying priorities for the order in which variables are branched can improve the performance of MIP significantly (in some instances, by an order of magnitude).

Figure 8-5 compares the greedy algorithm against a version that does not swap components when the parameter constraints on the *bestHost* are not satisfied. As was discussed in Section 6.3, by swapping components we decrease the possibility of getting "stuck" in a bad local optimum. The results of Figure 8-5 corroborate the importance of this heuristic on the accuracy of the greedy algorithm: the heuristic has improved the algorithm's accuracy by up to 50% in a large number of evaluation scenarios.

Finally, Figure 8-6 compares three variations of the genetic algorithm. The first two variations were discussed in Section 6.4, where one uses the Map sequence to group

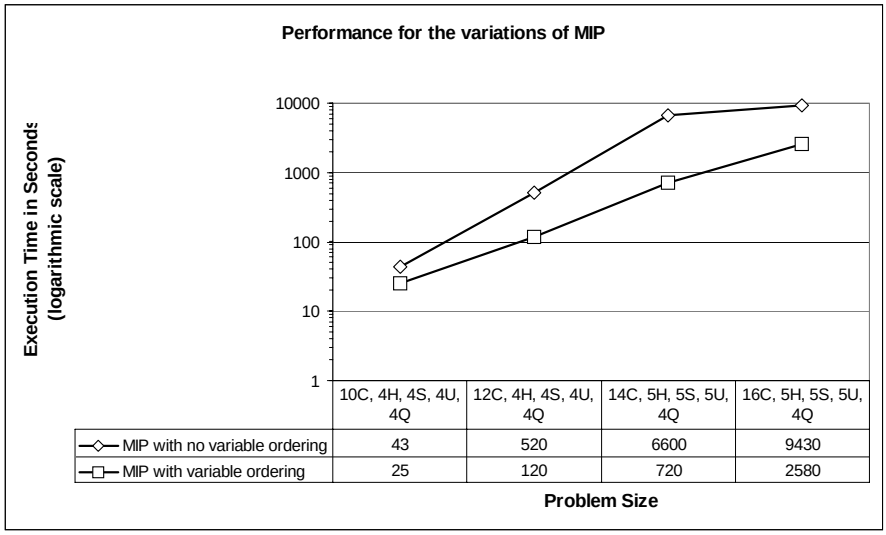


Figure 8-4: Impact of variable ordering on MIP's performance.

components based on service and the other does not. As expected, the results show a significant improvement in accuracy when components are grouped based on services, by up to a factor of 3. The last variation corresponds to the distributed and parallel execution of the genetic algorithm. In this variation we evolved three populations of one hundred individuals in parallel, where the populations shared their top ten individuals after every twenty evolutionary iterations. The results show a small improvement in accuracy (along with the expected significant time savings) over the simple scenario where only one population of individuals was used.

8.1.7 Market-based Algorithm

In order to quickly assess the performance of DecAp++ on large numbers of redeployment problems, involving large numbers of software components and hardware hosts, we implemented a simulated version of DecAp++ that runs on a single physical host. The distribution aspect of DecAp++ is simulated through the use of multiple,

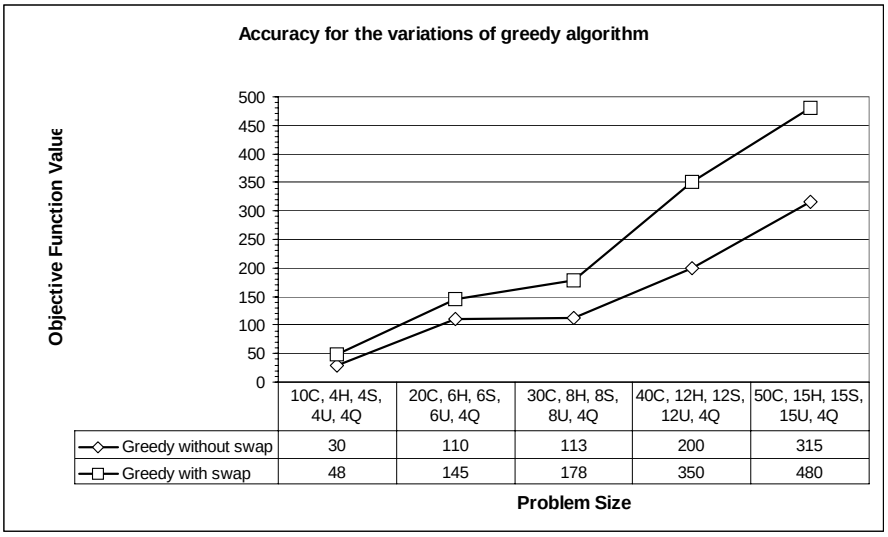


Figure 8-5: Impact of swapping on the accuracy of the greedy algorithm.

autonomous agents. We simulated the decentralization aspect of DecAp++ through the use of multiple threads and limited visibility among agents. DecAp++ was implemented in Java and integrated with DeSi. When DeSi's user interface invokes DecAp++, a bootstrap thread instantiates an agent object for each host. Each agent class is composed of two inner classes: auctioneer class and bidder class. Both auctioneer and bidder classes have their own threads of execution, which are started once the corresponding agent class is instantiated. Agents in the same domain are given access to each other's class variables. In our implementation of DecAp++, we used direct links to denote the awareness level of 1 (recall Figure 6-2B). Subsequent levels of awareness correspond to the number of intermediate hosts between a pair of hosts (recall Figure 6-2C). Auctioneer and bidder threads synchronize their interactions

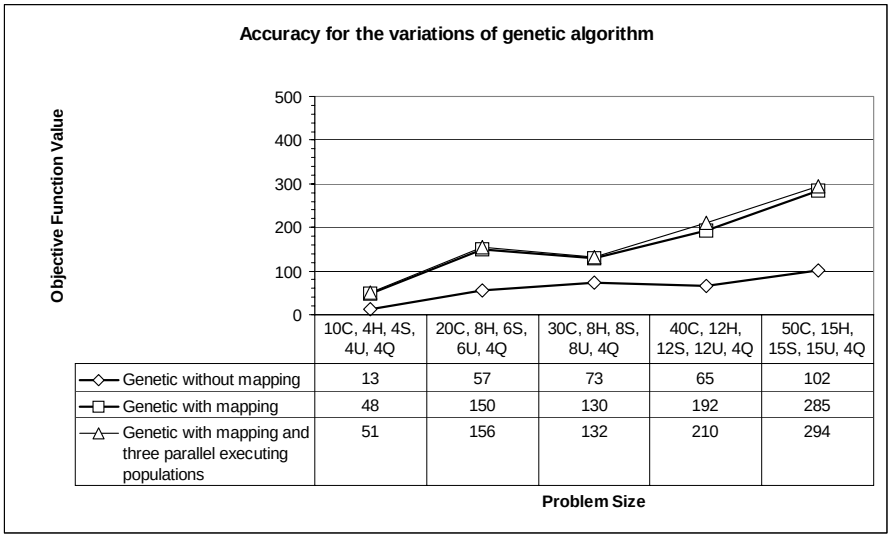


Figure 8-6: Impact of mapping and parallel execution on the accuracy of the genetic algorithm.

through message passing. A shared data structure that holds the current deployment of the system is updated as a result of each auction. DeSi's bootstrap class calculates the overall utility of the current deployment (i.e., the shared data structure) in pre-specified time intervals. The algorithm terminates when the utilities at two consecutive time intervals are the same, which indicates that the algorithm has converged to a solution.

Table 8-2 provides the comparison of DecAp++ with the centralized algorithms, in cases where the graph of hosts is fully connected (possibly via unreliable links). The last two columns show the results of running the algorithms for 10 different redeployment problems and averaging the results using the benchmarking option of DeSi. DecAp++ provided at least 40% improvement over the system's "most likely" deployment. On average, DecAp++ produced results that are on par with the centralized algorithms' results. However, in certain situations the performance of DecAp++

Table 8-2: Comparison of DecAp++'s accuracy in deployment architectures characterized by fully connected graphs of hosts.

		Scenario				
		10 C, 4 H, 5S, 5U 1 problem	20 C, 10H 8S, 8U 1 problem	40C, 15H 10S, 10U 1 problem	10 C, 4H 5S, 5U 10 problems	40C, 15H 10S, 10U 10 problems
Calculated Utility	MIP	32	infeasible	infeasible	25	infeasible
	MINLP	28	infeasible	infeasible	20	infeasible
	Greedy	29	86	513	22	610
	Genetic	28	89	490	22	634
	Unbiased average	20	57	320	15	428
	DecAp++ Awareness level = 1	*29*	*84*	*502*	*21*	*612*
% improvement over the unbiased average[a]		45	47	56	40	42

a. calculated as 100% * (DecAp++ – unbiased average) / unbiased average

could suffer, due to its reliance on the initial deployment. For example, in situations where some of the "best" hosts (recall the above description of the greedy algorithm) in the system do not have any components initially deployed on them, they may not ever be selected as the winners of any of the auctions.

Table 8-3 provides another comparison of DecAp++ with centralized algorithms in cases where the graph of hosts is not fully connected (each column is labelled with the percentage of missing host-to-host links). For each problem, DecAp++ was executed three times with different levels of awareness. In the last two rows we show the accuracy comparison of the level-1 DecAp++ with the Greedy and Genetic algorithms. As the table indicates, the algorithm's accuracy is negatively affected by the decrease in host inter-connectivity. However, as long as the graph of hosts is connected, increasing the level of awareness improves DecAp++'s performance significantly. Column 6 shows a scenario where as a result of a very high percentage of missing links, "islands" of hosts (i.e. subsets of hosts that are not connected to each

Table 8-3: Comparison of DecAp++'s performance in deployment architectures with varying levels of disconnected links among hosts.

		Scenario					
		25 C, 8 H 6 U, 6 S 20% of links missing	25 C, 8 H 6 U, 6 S 50% of links missing	25 C, 8 H 6 U, 6S 80% of links missing	50 C, 15 H 12 U, 12 S 30% of links missing	50 C, 15 H 12 U, 12 S 60% of links missing	50 C, 15 H 12 U, 12 S 90% of links missing
Calculated Utility	Greedy	101	157	131	280	310	281
	Genetic	103	169	119	282	289	388
	DecAp++ Awareness level = 1	93	138	84	285	252	50
	DecAp++ Awareness level = 2	94	150	99	286	265	62
	DecAp++ Awareness level = 3	94	154	109	286	267	63
% difference between DecAp++ awareness level 1 and Greedy		-7	-12	-35	2	-18	-82
% difference between DecAp++ awareness level 1 and Genetic		-9	-18	-29	1	-13	-87

other) are created, significantly impacting the accuracy of DecAp++. These results validate hypothesis 4: 1) For problems with a modest lack of knowledge (i.e., 20% to 30% disconnection) the solutions found by DecAp++ are within 10% of those found by the Greedy and Genetic; 2) As the lack of knowledge increases, and there are no islands of hosts, the solution accuracy degrades gracefully (e.g., in the problem of column 3, 80% disconnection has resulted in a solution that is only 35% worse than the solution found by Greedy).

Table 8-4 shows DecAp++'s convergence to a solution. Each iteration corresponds to the resulting overall utility (objective function) after auctioning each one of the components exactly once. Note that the largest gain is achieved in the first iteration of the algorithm, which shows that by just auctioning each component once, we can get a solution that is at least 66% of the final solution. Also note that after the first iteration of the algorithm, most components have found a "sweet spot", which results in no further redeployment of those components. This contributes to the quick convergence

Table 8-4: Demonstration of DecAp++'s convergence.

Iteration Number	10 C, 4 H 5 U, 5 S 20% of links missing 1 level of awareness	25 C, 10 H 8 U, 8 S 50% of links missing 1 level of awareness	40 C, 15 H 15 U, 15 S 70% of links missing 1 level of awareness	40 C, 15 H 15 U, 15 S 80% of links missing 2 levels of awareness
Initial Utility	0	0	0	0
1	50	79	190	146
2	67	91	267	183
3	72	93	281	193
4	74	94	285	199
5	74	94	286	206
6	74	94	286	207
7	74	94	286	207
% first iteration / final solution	67%	84%	66%	71%

of the algorithm, typically around the fifth or sixth iteration. For the largest problem (shown in the last column of Table 8-4), DecAp++'s execution time was 8.4s with the maximum auctioneer thread wait of 10ms. However, a variation of DecAp++ that used thread notification executed the same problem in 1s on a mid-range PC.[9]

8.2 Algorithmic Trade-Offs

As mentioned previously, one of our primary goals in the development of the framework has been to provide a generic environment that can be customized to the unique concerns of each application scenario. While we have discussed the framework's ability to model the variation points among application scenarios (system parameters, QoS dimensions, users, and so on), it may not be clear which algorithms are best suited for each scenario. In support of hypothesis 5 below we discuss various classes of systems and the trade-offs between the different algorithms for improving their deployment architecture.

One aspect of a distributed system that influences the complexity of improving its deployment architecture is its design paradigm, or *architectural style*. The two predominant design paradigms for distributed systems are Client-Server and Peer-to-Peer. Traditional Client-Server applications are typically composed of bulky and resource-expensive server components, which are accessed via thin and compara-

9.Since we only wanted to illustrate the execution time of the algorithm's logic, and not that of agents' synchronization, to obtain this result we leveraged the thread notification technique instead of the random thread wait times described in Chapter 6. Note that employing thread notification is possible only in a single-processor simulation of the algorithm.

tively more efficient client components. The resource requirements of client and server components dictate a particular deployment pattern, where the server components are deployed on capacious back-end computers and the client components are deployed on user workstations. Furthermore, the stylistic rules of Client-Server applications disallow interdependency among the clients, while the exact client components that need to be deployed on the users' workstations are determined based on user requirements and are often fixed throughout the system's execution. Therefore, the software engineer is primarily concerned with the deployment of server components among the back-end hosts. Given that usually there are fewer server components than client components, and fewer server computers than user workstations, the actual problem space of many client-server applications is much smaller than it may appear at first blush. In such systems, one could leverage the locational constraint feature of our framework to limit the problem space significantly. Therefore, it is feasible to run the MIP algorithm for a large class of client-server systems and find the optimal deployment architecture in a reasonable amount of time.

In contrast, a growing class of Peer-to-Peer applications are not restricted by stylistic rules or resource requirements that dictate a particular deployment architecture pattern. Therefore, locational constraints cannot be leveraged in the above manner, and the problem space remains exponentially large. For any, even medium-sized Peer-to-Peer system, the MIP algorithm becomes infeasible and the software engineer has to

leverage one of the three approximative algorithms to arrive at a sub-optimal, but significantly improved deployment architecture.

In large application scenarios, both the greedy and genetic approaches have an advantage over the MINLP approach, since they exhibit better performance and have a higher chance of finding a good solution. When the application scenario contains a large number of QoS dimensions, the genetic algorithm will typically outperform the greedy algorithm. This is because the genetic algorithm is only linearly affected by the number of QoS dimensions, while the greedy algorithm is polynomialy affected by this parameter. On the other hand, when the application scenario includes very restrictive constraints, the greedy algorithm has an advantage over the genetic algorithm. This is because the greedy algorithm makes incremental improvements to the solution, while the genetic algorithm depends on random mutation of individuals and may result in many invalid individuals in the population.

Another class of systems that are significantly impacted by the quality of deployment architecture are mobile and resource constrained systems, which are highly dependent on unreliable wireless networks on which they are running. For these systems, the genetic algorithm is the best option: it is the only algorithm in the framework that allows for parallel execution on multiple decentralized hosts, thus distributing the processing burden of running the algorithm among several hosts.

Finally, as mentioned earlier, in a growing class of decentralized systems, a complete model of a system may not exist on any of the hosts. In this type of environment, a decentralized algorithm, such as DecAp++, could be leveraged to improve the system's deployment architecture.

8.3 Prism-MW

As mentioned earlier, since the framework is geared towards embedded and resource-constrained computing domains, it is important for the implementation support to be highly efficient and scalable. In this section we summarize our evaluation of Prism-MW's efficiency and scalability. We provide the benchmarking results for the Java version of Prism-MW. In fact, the C++ implementation of Prism-MW is more efficient than its Java implementation. More details about the C++ implementation of Prism-MW can be found in [31]. Our goals have been to (1) provide empirical results of the performance trade-offs that are associated with our design decisions, and (2) demonstrate the middleware's efficiency and scalability in large, possibly distributed systems with different structures.

In support of the first goal, we have evaluated two types of performance trade-offs that were discussed in Section 7.1.3:

- The trade-off between the two alternative configurations of a local architecture (recall Figures 7-3 and 7-4). We used architectures in which a single component

communicates with a varying number of identical components, either through a single connector or via ports that directly connect the components.

- The trade-off between the three alternative configurations of a distributed architecture (recall Figure 7-7). We considered a special case, where the connectors broadcast events, which results in five semantically identical configurations of a distributed architecture, as detailed below.

In support of the second goal, we have measured the overhead in application size caused by Prism-MW. We have also evaluated the execution of large architectures with different topologies and processing loads:

- The sensitivity analysis of the middleware's performance to the size of the architecture. We consider an architecture configured in a manner similar to the one depicted in Figure 7-4a, and execute it on different platforms for varying numbers of events and "bottom" components.
- The middleware's scalability to large, distributed architectures with different topologies. We considered two primary architectures with different topologies: a "flat" architecture, where a large set of components interacts with a set of other components directly (see Figure 8-11); and a "tall" architecture, where a large set of components interact indirectly via intermediating components (see Figure 8-12). These two architecture types can then be combined to produce arbitrary "hybrid" topologies.

Since we are interested in measuring the overhead induced on an application by Prism-MW's, the software components used throughout the example scenarios contain minimal application logic (e.g., counting the number of events sent/received, forwarding events). Furthermore, all the events exchanged between local and remote components are simple events with no payload. We selected the size of the thread pool and event queue based on the expected load and the size of each architecture. Note that Prism-MW allows for the specification of arbitrarily large thread pools and event queues. In the case of most benchmarks we kept a constant size thread pool and event queue to simplify the assessment of the middleware's performance under varying loads.

The environment set-up consisted of (1) mid-range PCs with Intel Pentium IV 1.5 GHz processors and 256 MB of RAM running JVM 1.4.2 on Microsoft Windows XP, (2) PDAs of type Compaq iPAQ H3870 with 200 MHz processors and 64 MB of RAM running Jeode JVM on WindowsCE 2002, and (3) a dedicated network leveraging a dual-band wireless 2.4 GHz router.

8.3.1. Middleware Overhead

The performance of Prism-MW is comparable to solutions using a plain programming language (PL). Each Prism-MW event exchange causes five PL-level method invocations (typically highly optimized in a PL), and a comparatively more expensive context switch if the architecture is instantiated with more than one shepherd thread.[10]

Analogous functionality would be accomplished in a PL with two invocations and, assuming concurrent processing is desired, a context switch. It should also be noted that it is unlikely that a plain PL could support a number of development situations for which Prism-MW is well suited (e.g., asynchronous event multicast) and due to which it introduces its performance overhead in the first place.

Memory usage of Prism-MW's core (*mw_mem*), recorded at the time of architecture initialization, is 2.3 KB. The overhead of a "base" Prism-MW component (*comp_mem*), without any application-specific methods or state, is 0.12 KB, while the overhead of a "base" connector (*conn_mem*) is 0.09 KB. The memory overhead of a "base" port (*port_mem*) is 0.04 KB, while the overhead of an *ExtensiblePort* is 0.2 KB. The memory overhead of each connection object is 8 KB, which leverages Java's implementation of socket-based TCP/IP communication protocol. The memory overhead of a *DistributionEnabledPort* that contains a single connection instance is 8.5 KB. The memory overhead of creating and sending a single event (*evt_mem*) can be estimated using the following formula, obtained empirically:

$$evt\_mem\ (in\ KB) = 0.04 + 0.01 * num\_of\_parameters$$

The formula assumes that the parameters do not contain complex objects, but may contain simple objects (e.g., Java Integer or String).[11]

10. The five method invocations involve traversing the ports, placing the event in the event queue, and dispatching the event to the recipient component.
11. In this sense, the measure represents minimum event overhead. Use of complex objects as event parameters is independent of the middleware, but is an application-level decision.

As an illustration, the memory overhead induced by using Prism-MW in the largest instantiation of the TDS architecture consisting of a single *Headquarters* subsystem, four *Commander* subsystems, and 100 *Soldier* subsystems can be closely approximated as follows:

*num_arch * (mw_mem + (q_size * evt_mem)) + num_comps * comp_mem + num_conns * conn_mem + num_ports * port_mem + num_dist_ports * dist_port_mem =105 * (2.3 + (25 *(0.04 + (0.01 * 1))) + (245 * 0.12) + (217* 0.09) + (875* 0.04)+(109*0.5) = 511.5 KB*

The above formula uses the average size of the event queue for each *Architecture* object (25), and average number of parameters for TDS events (one). The formula also assumes that each event queue is full (which we have never observed during actual execution of TDS). Note that the dynamic size of the application is approximately 1 MB for the *Headquarters* subsystem, 600 KB for each *Commander*, and 90 KB for each *Soldier* subsystem, resulting in the total application size of 12.5 MB. Therefore, Prism-MW induced at most a 4% overhead on the application's dynamic memory consumption.

Our measurements of the memory overhead for the awareness, deployment, mobility, and disconnected operation support showed that on average the Java implementation of the Prism-MW skeleton configuration (*Admin*, *DistributionEnabledPort*, and

Prism-MW's core) occupies around 14 KB on each host. The *Admin* itself occupies 4 KB of memory.

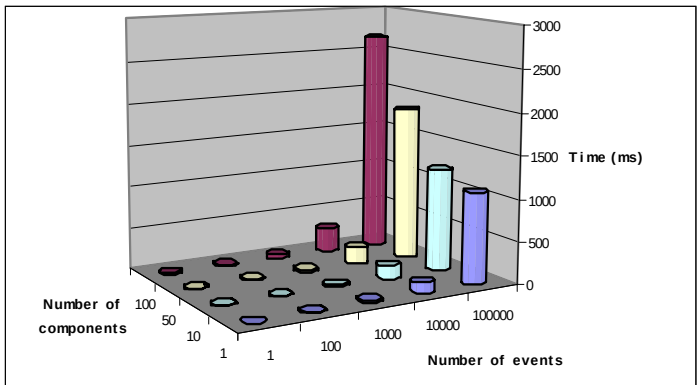
8.3.2. Middleware Performance in a Local Setting

To empirically evaluate Prism-MW's core we use an architecture where one component is communicating with a varying number (*n*) of identical recipient components via a connector (Figure 7-4a shows such an architecture with two recipient components). Thus, all the components in this architecture are part of the same *Architecture* object and reside in a single address space. For this architecture we use a pool of 10 shepherd threads and a queue of 1000 events (*q_size*).

Figures 8-7 and 8-8 show representative benchmark results. In Figure 8-7, a maximum of 100,000 simple (parameter-less) events were sent asynchronously by the single sender component to a maximum of 100 recipient components (resulting in between 100 to 10,000,000 invocations of component *handle* methods) for the application running on a PC. The 10 million events are processed in under 3 seconds on the PC. Figure 8-8 shows the results obtained on a PDA, a comparatively much less capacious and performant platform: a maximum of 10,000 events are sent to a maximum of 100 components (resulting in up to 1,000,000 invocations of component *handle* methods).

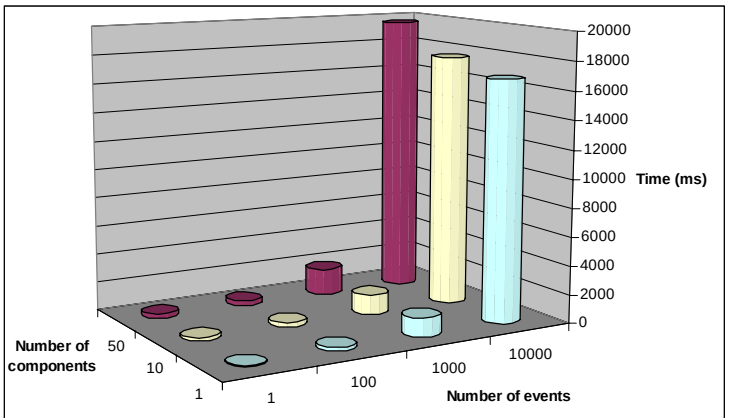
In addition to the above "flat" architecture, another series of benchmarks we ran involved a "chain" of *n* components communicating either directly through ports or

via *n-1* intervening connectors. For example, the total round-trip time for a single event in the case where the architecture involved 100,001 components and 100,000 connectors was 1.1 milliseconds on a PC. In addition to demonstrating Prism-MW core's efficiency, these benchmarks also served to highlight its scalability.



	1E+05	10000	1000	100	1
100	2674	300	50	20	20
50	1843	211	40	20	10
10	1222	150	30	11	10
1	1081	131	30	10	1

Figure 8-7: Benchmark results of executing Prism on a PC.



	10000	1000	100	1
50	19649	1871	385	289
10	17654	1403	325	217
1	16784	1320	256	62

Figure 8-8: Benchmark results of executing Prism on a PDA.

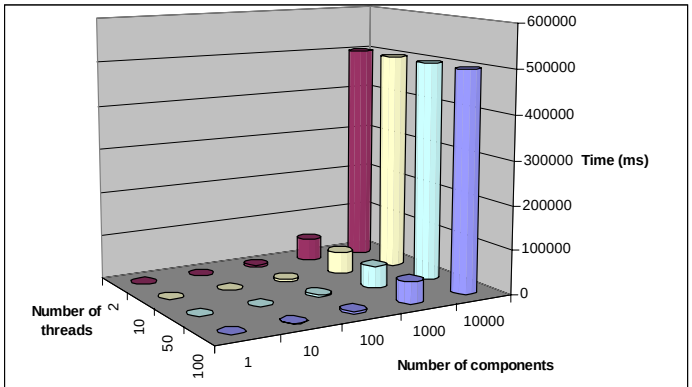
To evaluate the performance trade-off between two alternative usage scenarios of Prism-MW (recall Figures 7-3 and 7-4), we employed two variations of the above "flat" architecture. In the first variation, discussed above, the communication takes place through a single connector, while the second variation employs direct links between component ports. Each one of the *n* components was implemented with a fixed event handling delay of 50 msec, to simulate application-specific processing (*comp_proc_time*) and to utilize the benefits of parallel processing.

The results of the benchmark are shown in Figures 8-9 and 8-10. One parameter-less event was sent asynchronously by the single sender component to all the recipient components, resulting in *n* events being handled. The results demonstrate that a higher degree of parallelism, and therefore better performance, can be achieved by using direct connections among components. On the other hand, the use of a connector resulted in lower memory consumption, since each outgoing event is not replicated *n* times. Finally, note that the total processing time in the case of direct communication (illustrated in Figure 8-10) can be approximated using the following formula, where *numComps* represents the number of components, and *numThreads* the number of shepherd threads in an architecture:

$$total\_proc\_time \approx \begin{cases} comp\_proc\_time, & if \quad numComps < numThreads \\ comp\_proc\_time * numComps / numThreads, & if \quad numComps \geq numThreads \end{cases}$$

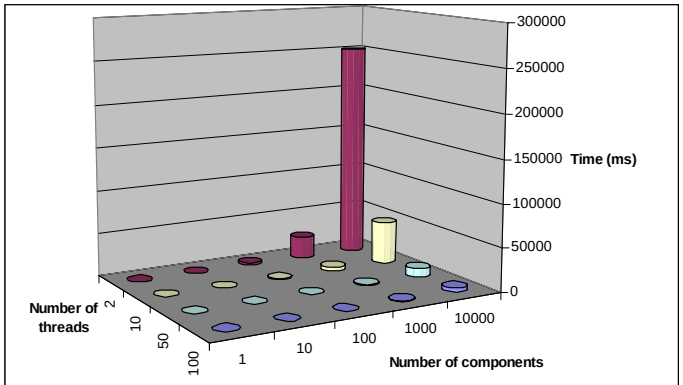
8.3.3. Middleware Performance in a Distributed Setting

While Prism-MW's *DistributionEnabledPort*s are in principle independent of the employed communication protocols (recall Section 7.1.5), their performance is directly impacted by the underlying implementations of those protocols. The results presented here are based on *DistributionEnabledPort*s that leverage Java's implemen-



	10000	1000	100	10	1
2	500730	50072	5017	511	60
10	500760	50082	5017	501	60
50	500785	50082	5057	511	60
100	500760	50079	5027	551	60

Figure 8-9: Components communicating through a connector.



	10000	1000	100	10	1
2	251302	25056	2514	261	60
10	50102	5028	500	60	60
50	10075	1032	111	50	60
100	5077	541	70	50	60

Figure 8-10: Components communicating directly via ports.

tation of TCP/IP sockets. In a large number of benchmarks involving architectures of varying sizes, topologies, and communication profiles, we compared the performance of a *DistributionEnabledPort* with a "pure" Java implementation of TCP/IP. Our results indicate that a Prism-MW *DistributionEnabledPort* adds no more than 2% in performance overhead to Java's implementation of TCP/IP.

In Section 7.1.5 we identified three different ways of instantiating a distributed architecture with identical event routing semantics. In fact, when the routing policy is event broadcast (i.e., no filtering is performed by connectors), five semantically equivalent ways of instantiating a distributed architecture are possible. To study their performance, we created five example scenarios, comprising three distributed *Architecture* objects with five components each, that were configured as follows:

- *Scenario 1:* Each component communicates directly to every other component via a separate *DistributionEnabledPort* (see Figure 7-7a).
- *Scenario 2:* Each component on the requesting device uses a single *DistributionEnabledPort* to communicate directly to every other component (see Figure 7-7b, where the requesting device corresponds to address space A1).
- *Scenario 3:* Each component on the requesting device uses a local bidirectional broadcasting connector to communicate with remote components. The connector has a separate *DistributionEnabledPort* for each remote component (see Figure 7-7c).

- *Scenario 4:* This is similar to the architecture of Scenario 3, with the exception that the connector has only a single *DistributionEnabledPort* to communicate with all of the remote components.
- *Scenario 5:* Local bidirectional broadcasting connectors with single *DistributionEnabledPort*s are used to mediate the communication in all three architectures.

Table 8-5 shows the performance measurements under each of the scenarios described above, with different event loads sent by the requesting architecture. The measurements reflect the time elapsed before the requesting architecture receives all the reply events from the remaining architectures. Given that the ports and connectors used in this example broadcast the events, each event sent by one of the 5 requesting components (i.e., between 5 and 500 events sent in the scenarios depicted in Table 8-5) results in a total of 10 replies returned (i.e., between 50 and 5000 events returned).

Table 8-5: Distributed Architecture Scenarios

	1 event	**30 events**	**100 events**
Scenario 1	351kb, 210ms	351kb, 2377ms	351kb, 6679ms
Scenario 2	310kb, 210ms	310kb, 2313ms	310kb, 6786ms
Scenario 3	132kb, 110ms	132kb, 1155ms	132kb, 2760ms
Scenario 4	123kb, 110ms	123kb, 1185ms	123kb, 2533ms
Scenario 5	86kb, 86ms	86kb, 731ms	86kb, 1735ms

We make the following two observations from the results of Table 8-5. First, architectures with lower numbers of *DistributionEnabledPort*s have lower memory footprints and faster running times. This is expected since each *DistributionEnabledPort* adds overhead both in terms of memory and execution. Therefore, in response to this issue

Prism-MW allows multiple connections to be associated with a single *DistributionEnabledPort*. More significantly, architectures with lower numbers of network connections have much lower memory footprints and faster running times. This is expected since each network connection has its own internal thread that reads/writes events from/to the network link. To minimize the number of network connections, one may leverage Prism-MW's connectors. For example, in Scenario 5 the usage of connectors in all the three architectures resulted in the most efficient configuration, in which a total of only three *DistributionEnabledPort*s and four connections are instantiated.

As discussed earlier, to evaluate the middleware's scalability, we benchmarked two large, distributed architectures with different topologies under heavy loads. For both of these architectures we used a pool of 10 shepherd threads and a queue of 10,000 events on each device.

"Flat" Architecture. An example of such architecture is depicted in Figure 8-11, where 11 distributed Prism-MW *Architecture* objects are comprised of 10 components each. In one benchmark, each of the ten components on the first device sends 100 events of type request to each of the ten remaining devices. Each component on the replying hosts receives each request event and sends a reply event back to the first device. The connectors we used in our benchmarks did not perform any filtering of events. Therefore, 1000 request events were sent from *Device 1* (10 components *

100 events each), resulting in 10,000 reply events from each of the ten receiving devices (1000 events * 10 components), for a total of 100,000 reply events to be received by each requesting component, i.e., a total of 1,000,000 reply events to be handled by *Device 1*'s components. When run on a PC, this scenario required 18 seconds for all of the requesting components to send the requests and receive all of the reply events.

"Tall" Architecture. An example of such architecture is depicted in Figure 8-12, where 10 distributed Prism-MW *Architecture* objects are composed into a chain. In one benchmark, the first device was composed of a single component sending requests; the next eight devices were composed of two components that received each event and propagated it; finally, the two components on the tenth device sent back a

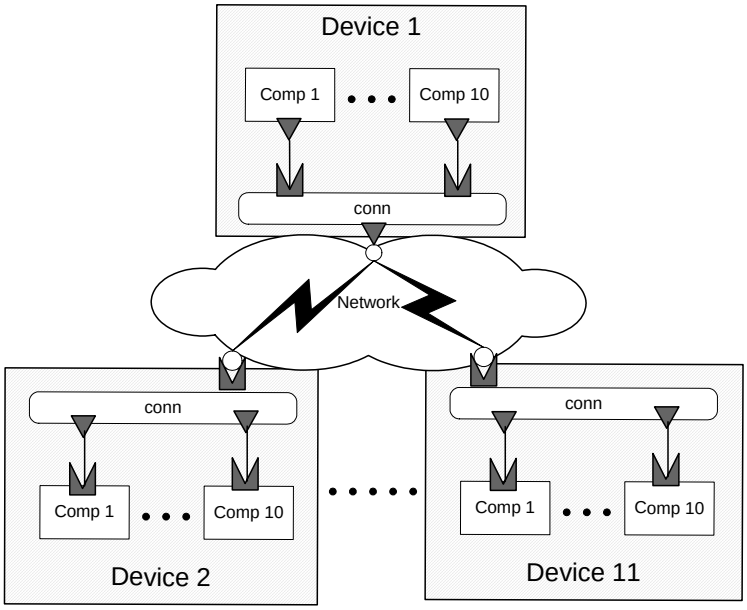


Figure 8-11: A "flat" architecture composed of 110 components deployed over 11 distributed devices.

reply event for every request event they received. Our benchmarks measured the time it took for the component on *Device 1* to send a request and receive back all of the replies. Note that due to the propagation of events by the two components on each device, for the configuration shown in Figure 8-12 a total of 2^9=512 request events are received by *Device 10*, and 2^{17}=131,072 reply events are received by *Device 1*. The total time for the first device to send a request and receive all of the replies back was 31 seconds on a PC.

To assess the extent of Prism-MW's scalability, we have instantiated the "flat" and "tall" configurations in benchmarks with up to 1 million components and/or connectors, exchanging up to 100 million events [39]. The limiting factors in these cases would invariably become the capabilities of the underlying hardware platforms. In principle, Prism-MW's modularity and separation of concerns directly aid its scalability in the numbers of supported devices, components, connectors, ports, threads, and events.

8.4 DeSi

The goal of DeSi is to allow visualization of different characteristics of software deployment architectures in highly distributed settings, the assessment of such architectures, and possibly their reconfiguration. In this section we evaluate DeSi in terms of four properties that we believe to be highly relevant in this context: tailorability,

scalability, efficiency, and ability to explore the problem space. Each property is discussed in more detail below.

8.4.1 Tailorability

DeSi is an environment intended for exploring a large number of issues concerning distributed software systems. As discussed before, to date we have focused on the impact a deployment architecture has on a system's four QoS dimensions of availability, latency, communication security, and energy consumption. However, DeSi's MVC architecture and component-based design allow it in principle to be customized for visualizing and assessing arbitrary system properties (e.g., fault-tolerance, scalability, performance). All four components of DeSi's *Model* subsystem (*SystemData*, *UserPrefData*, *GraphViewData*, and *AlgoResultData)* could be easily extended to represent

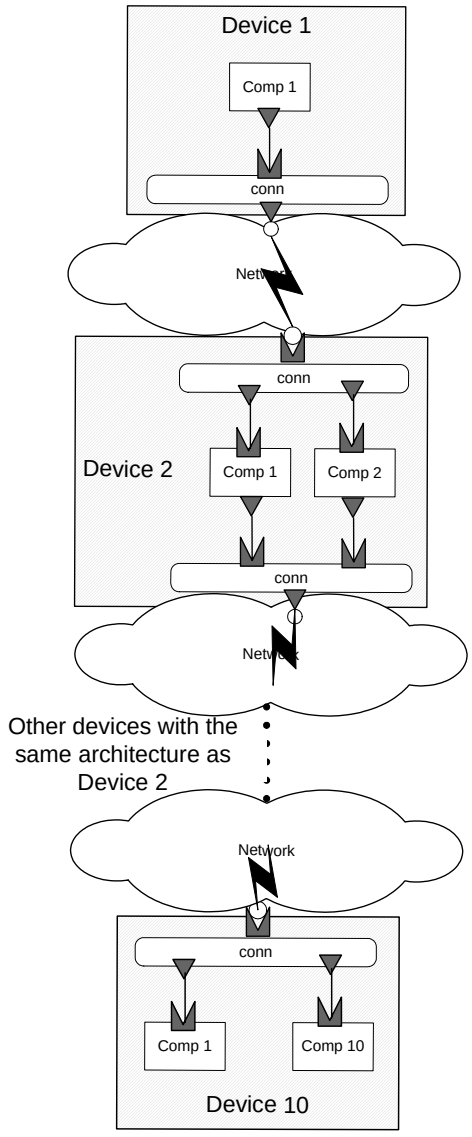


Figure 8-12: A "tall" architecture with 19 components deployed over 10 devices.

other system properties through the addition of new attributes and methods to the corresponding classes. Another aspect of DeSi's tailorability is the ability to add new *View*s or modify the existing ones. Clear separation between DeSi *View* and *Model* components makes creating different visualizations of the same model easy. In fact, we leveraged this dichotomy in constructing two different visualizations of the underlying model: the system deployment view (recall Figure 7-18) and the user preference view (recall Figure 7-19). DeSi also enables quick replacement of the *Control* components without modifying the *View* components. For example, two *Control* components may use the *GraphView*'s API for setting the thickness of inter-host links differently: one to depict the reliability and the other to depict the available bandwidth between the hosts. Finally, DeSi also provides the ability to interface with an arbitrary middleware platform.

8.4.2 Scalability

DeSi is targeted at systems comprising many components distributed across many hosts. DeSi supports scalability in the (1) size of its *Model*, (2) scalability of its *View*s, and (3) scalability of the *Controller*'s algorithms. *Model*s of systems represented in DeSi can be arbitrarily large since they are centralized and capture only the subset of system properties that are of interest. Another aspect of DeSi's scalability are *Controller*'s algorithm implementations: with the exception of the *MIP*, all of the algorithms are polynomial in the number of components. As shown in Section 8.1, we have tested DeSi *Model*'s scalability by generating random models with hundreds of

hosts and thousands of components, and *Controller*'s scalability by successfully running the algorithms on these models. Finally, the combination of the hierarchical viewing capabilities of the DeSi *View* subsystem (i.e., system-wide view, single host view, single component view), the zooming capability, and the ability to drag components and hosts to view their connectivity, enables one to effectively visualize distributed systems with very large numbers of components and hosts.

8.4.3 Efficiency

One of our goals in the development of DeSi has been to ensure that its scalability support does not come at the expense of its performance. As a result of developing the visualization components using Eclipse's GEF, DeSi's support for visualizing deployment architectures exhibits much better performance than an older version that was implemented using Java Swing libraries. We also enhanced the performance of the *GraphView* component by repainting only parts of the screen that correspond to the modified parts of the model. As discussed in Section 7.2.1.2, we also provide three options for running the redeployment algorithms. This enables us to customize the overhead associated with running the algorithms and displaying their results. Finally, with the exception of the MIP algorithm, all of the provided algorithms run in polynomial time (recall Chapter 6).

8.4.4 Exploration Capabilities

The nature of highly distributed systems, their properties, and the effects of their parameters on those properties is not well understood. This is particularly the case with the effect a system's deployment architecture has on the four QoS properties that we have modeled in this research. The DeSi environment provides a rich set of capabilities for exploring deployment architectures of distributed systems. DeSi provides side-by-side comparisons of different algorithms along multiple dimensions (achieved QoS, running time, and estimated redeployment time). It also supports tailoring of the individual parameters of the system model, which allows quick assessment of the sensitivity of different algorithms to these changes. Next, DeSi provides the ability to tailor its random generation of deployment architectures to focus on specific classes of systems or deployment scenarios (e.g., by modifying desired ranges of certain parameters such as the minimum and maximum frequencies of component interactions). DeSi supports algorithm benchmarking by automatically generating, assessing, and comparing the performance of different algorithms for a large number of randomly generated systems. DeSi's extensibility enables rapid evaluation of new algorithms. Finally, through its *MiddlewareAdapter*, DeSi provides the ability to visualize, reason about, and modify an *actual* system.

8.5 Experience

We have applied various aspects of this dissertation on two application families developed in collaboration with two external software development organizations.

The first application family is TDS that was introduced in Chapter 1 and was implemented using the Java version of Prism-MW in collaboration with the U.S. Army. It is representative of a large number of mobile pervasive systems that are intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. The second application family is MIDAS, and was implemented using the C++ version of Prism-MW in collaboration with the Bosch Research and Technology Center. There are several different classes of users (i.e., users with different roles and missions) in a typical TDS scenario. We have leveraged this characteristic of TDS in providing support for modeling, and improving a system's deployment based on user preferences. MIDAS is implemented in C++, which unlike Java does not natively provide support for dynamic class loading and code mobility. We describe our experience with using the Prism-MW infrastructure for monitoring and (re)deployment of MIDAS. This experience shows that the techniques presented in this dissertation can be applied to systems implemented in compiled languages as well as interpreted languages. These experiences have increased our confidence in the validity of the approach and the results obtained in the laboratory setting.

8.5.1. TDS

We have constructed several instances of the TDS application. Figure 8-13 shows the instance of TDS that was introduced in Chapter 1 while it is being monitored and deployed via Prism-MW's meta-level components (recall Sections 7.1.9 and 7.1.10). Figure 8-14 shows the deployment architecture view of a portion of this TDS scenario

in DeSi. The models were populated with the data available at design-time (e.g., upper-bound estimates on the sizes of components, locational constraints, available memory on the hosts) and monitored data (e.g., frequency of invocations, network reliabilities). Figure 8-15 shows a portion of the models constructed in DeSi to represent the users of the system, their QoS preferences, and user-level services provisioned by the system. In this case, the users were *Headquarters Commander*, *Left Commander*, *Right Commander*, *Left Soldier*, and *Right Soldier*. Altogether, the users specified ten different QoS preferences for a total of eight services. The services were *Analyze Strategy*, *Move Resources*, *Get Weather*, *Get Map*, *Update Resources*, *Remove Resources*, *Simulate Fight*, and *Advise Deployment*.

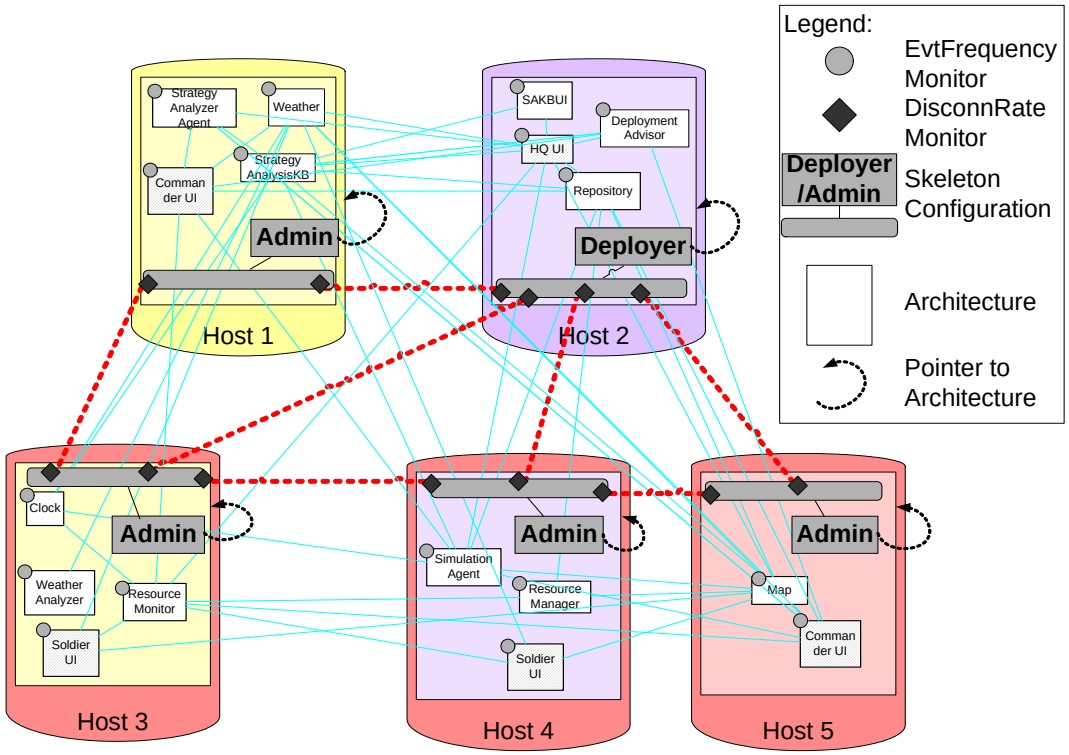


Figure 8-13: Instance of TDS that is deployed and monitored using Prism-MW.

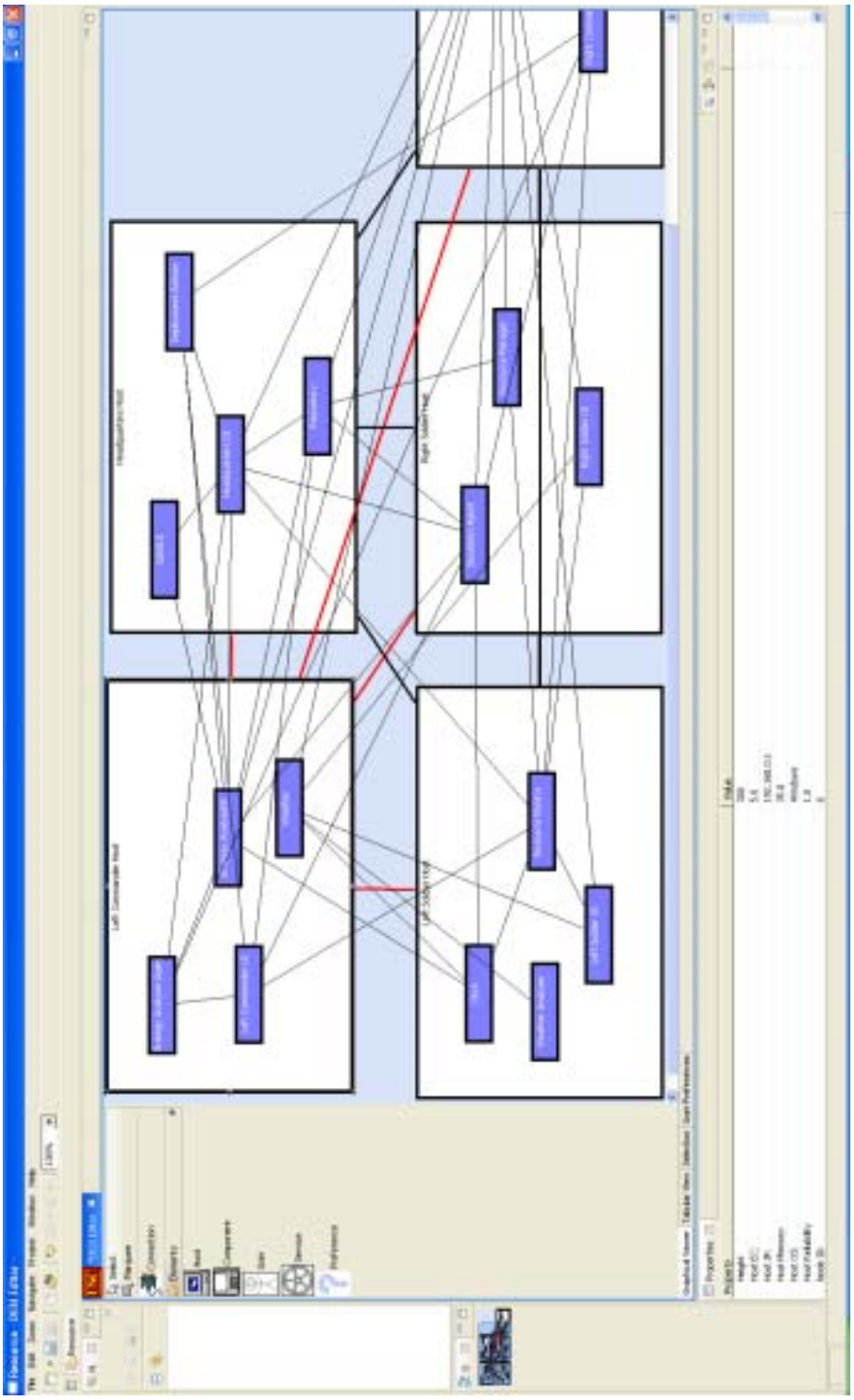


Figure 8-14: Instance of the TDS deployment architecture modeled in DeSi.

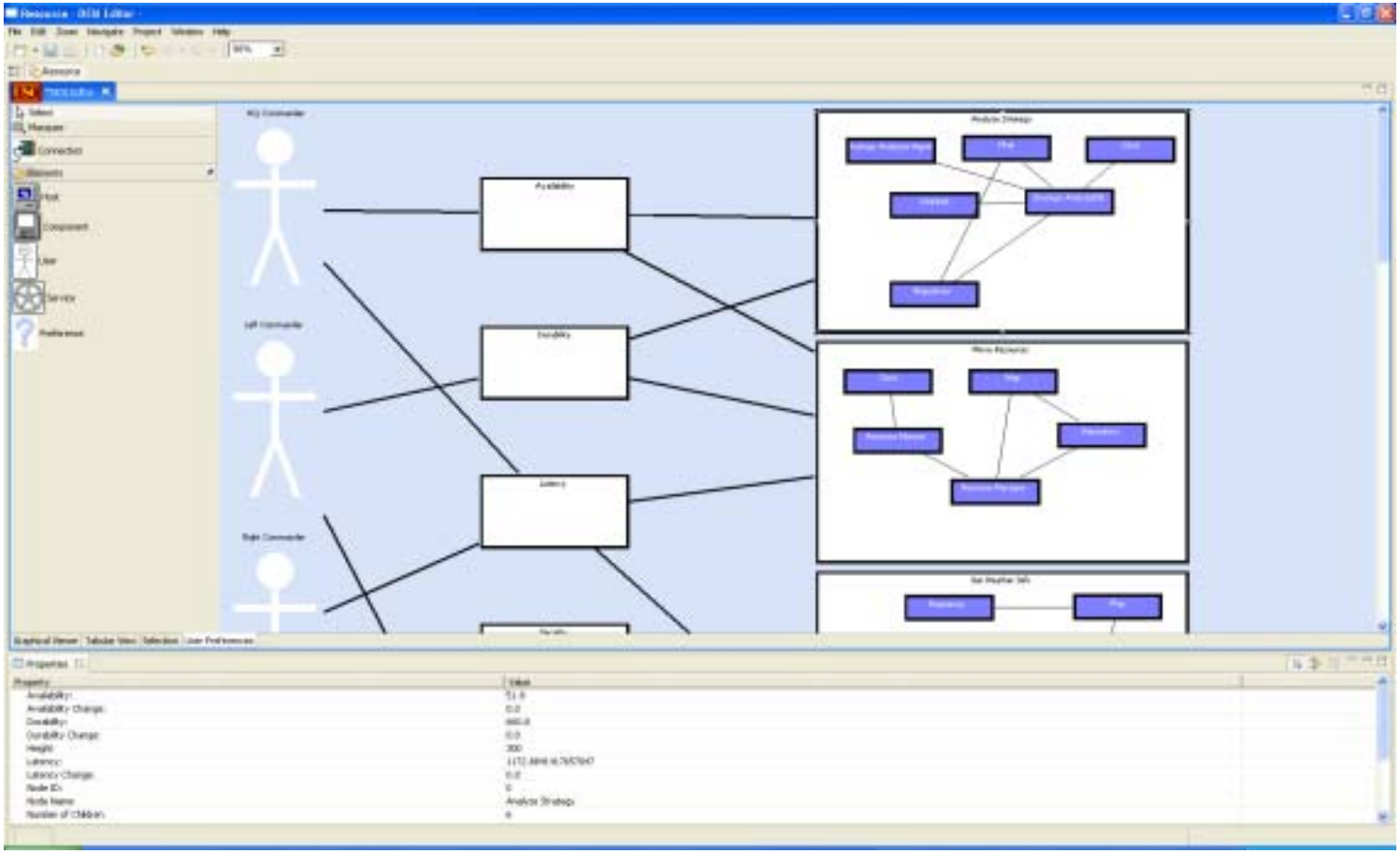


Figure 8-15: Instance of TDS users, QoS preferences, and user-level services modeled in DeSi.

Table 8-6 shows the preferences of five hypothetical users in this TDS scenario. For example, the table shows that *Headquarters Commander* has specified a utility of 0.4 for 0.1 (or 10%) change in the availability of the *Analyze Strategy* service. Note that unlike the benchmark results shown in Section 8.1 where we evaluated the algorithms in the most stringent scenarios (i.e., all users specifying preferences for all of the QoS dimensions of all services), in this application scenario the users have not specified preferences for some of the QoS dimensions of services. This is representative of what happens in many real-world scenarios, where a user may not use all of the services provided by the system.

Table 8-7 shows the results of executing the greedy algorithm on this instance of the TDS application. We chose the greedy algorithm as the best approach to solving this

Table 8-6: Users' preferences in the TDS scenario.

	HQ Commander	**Left Commander**	**Right Commander**	**Left Soldier**	**Right Soldier**
Analyze Strategy	Availability (rate=0.1, util=0.4)	Energy Cons. (rate=0.05, util=0.2)			
Move Resources	Availability (rate=0.1, util=0.4)	Energy Cons. (rate=0.05, util=0.2)	Latency (rate=0.3, util=0.4)		
Get Weather				Security (rate=0.9, util=0.1) Durability (rate=0.1, util=0.4)	Energy Cons. (rate=0.1, util=0.4)
Get Map	Latency (rate=0.3, util=0.4)	Security (rate=0.9, util=0.1)	Latency (rate=0.3, util=0.4)	Energy Cons. (rate=0.1, util=0.4)	Energy Cons. (rate=0.1, util=0.4)
Update Resources			Latency (rate=0.05, util=0.1)		Latency (rate=0.05, util=0.1)
Remove Resources				Energy Cons. (rate=0.1, util=0.4)	Energy Cons. (rate=0.1, util=0.4)
Simulate Fight					
Advise Deployment		Latency (rate=0.05, util=0.1)	Latency (rate=0.05, util=0.1)		Energy Cons. (rate=0.05, util=0.1)

Table 8-7: Results of running the greedy algorithm on the TDS scenario.

	Availability	Latency	Communication Security	Energy Consumption
Analyze Strategy	63%	12%	5%	79%
Move Resources	78%	68%	2%	63%
Get Weather	-3%	11%	81%	59%
Get Map	29%	59%	82%	73%
Update Resources	9%	92%	19%	-6%
Remove Resources	-1%	25%	23%	103%
Simulate Fight	17%	-8%	1%	-6%
Advise Deployment	61%	134%	22%	92%

problem for two reasons (recall Section 8.2): 1) the architecture was fairly large, and we were not able to solve the problem optimally via MIP, 2) there were many locational constraints, which hamper the accuracy of the genetic algorithm. The results demonstrate that on average significant improvements were achieved. Furthermore, typically the QoS preferences that are most important to the users are improved more than others. The few cases where QoS dimensions have slightly degraded can be attributed to the fact that the users have not specified any preferences.

8.5.2. MIDAS

MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly in the manner shown in Figure 8-16. The sensors are used to monitor the environment around them. They communicate their status to one another and to the gateways. The gateway nodes are responsible for managing and coordinating the sensors. Furthermore, the gateways translate, aggregate, and fuse the data

received from the sensors, and propagate the appropriate data (e.g., event) to the hubs. Hubs in turn are used to evaluate and visualize the sensor data for human users, as well as to provide an interface through which the user can send control commands to the various sensors and gateways in the system. Hubs may also be configured to propagate the appropriate sensor data to PDAs, which are then used by the mobile users of the system.

To facilitate the integration of MIDAS with hardware and software that had been developed previously by Bosch, we were required to develop the portion of MIDAS that runs on the gateways and hubs in C++ and the portion that runs on the PDA in Java. C++ was also selected due to its expected efficiency and low-overhead.

Figure 8-16: MIDAS system.

Figure 8-17 shows a subset of MIDAS's software architecture. As denoted on the left side of the figure, three types of architectural styles have been leveraged: *service-oriented*, *publish-subscribe*, and *peer-to-peer*. The peer-to-peer portion of this architecture corresponds to the meta-level functionality of monitoring, analysis, and adaptation of the system via DeSi. Therefore, the MIDAS application itself has been

developed using both service-oriented and publish-subscribe styles, resulting in a hybrid architectural style. The publish-subscribe portion of the MIDAS software architecture corresponds to the communication backbone of the MIDAS systems, and is responsible for: transformation of raw sensor data to the MIDAS format and vice versa (*SensorProcessor*), sensor management and delivery of commands (*SessionOperator, SessionAdministrator*), data fusion and selective propagation of events (*GWToHubProcessor*, *GWToGWProcessor*)), and sensor control interface and display (*HubOperator, EventDisplay*). The service-oriented portion of MIDAS corresponds to a set of facilities that are essential, but much less frequently used than the facilities provided by the publish-subscribe portion of the architecture. Therefore, to accommodate the overall lack of computing resources these services are dispersed throughout the distributed system. Some examples of services provided by the service-oriented portion are: *NodeInfoSvc* that provides a look-up service for a sensor's properties (e.g., location of a sensor), *TroubleLogSvc* that provides a facility for storing the errors in the system (e.g., failure to communicate with a sensor), *EventNotificationSvc* that given an event type determines the best procedure for handling events of that type, and so on.

For component deployment we needed a mechanism to load the components at runtime. However, unlike Java, C++ does not provide a mechanism for dynamic class loading. Therefore, to address this shortcoming, we first compiled each component's implementation either as a Dynamic Link Library (DLL) for Windows or as a Shared

Library for Linux. We then used the file serialization facility implemented in the C++ version of Prism-MW for serializing the DLL or Shared Library file into byte-array format (more details can be found in [31]). Finally, the byte-array was transmitted over the network to its target host, deserialized, and stored as a DLL or a Shared Library locally. When the software system is started, the application's implementation encapsulated within the DLL or Shared Library is bound to the middleware facilities. For stateful component redeployment, the application logic of a software component (which extends the *AbstractImplementation* class of Prism-MW) needs to provide implementation of two static methods: *toArray* and *fromArray. toArray* returns the state of a software component in a byte array, while *fromArray* instantiates the software component given an appropriate byte array.

In the deployment and reconfiguration of MIDAS we have leveraged a centralized coordination mechanism, where DeSi determines a complete set of commands required for redeploying and reconfiguring the system based on the model of system's deployment architecture (recall Section 7.3). In a scenario that is similar to that shown in Figure 8-16 but with one more gateway, a total of 165 commands were sent from DeSi to the *Deployer* and *Admin* components. Each command encapsulated within an event corresponds to a runtime adaptation or reconfiguration that should be be performed by the middleware. The order in which commands are delivered and executed is critical. For example, the SDEngine component, which provides support for service discovery [31], should be instantiated before other software components

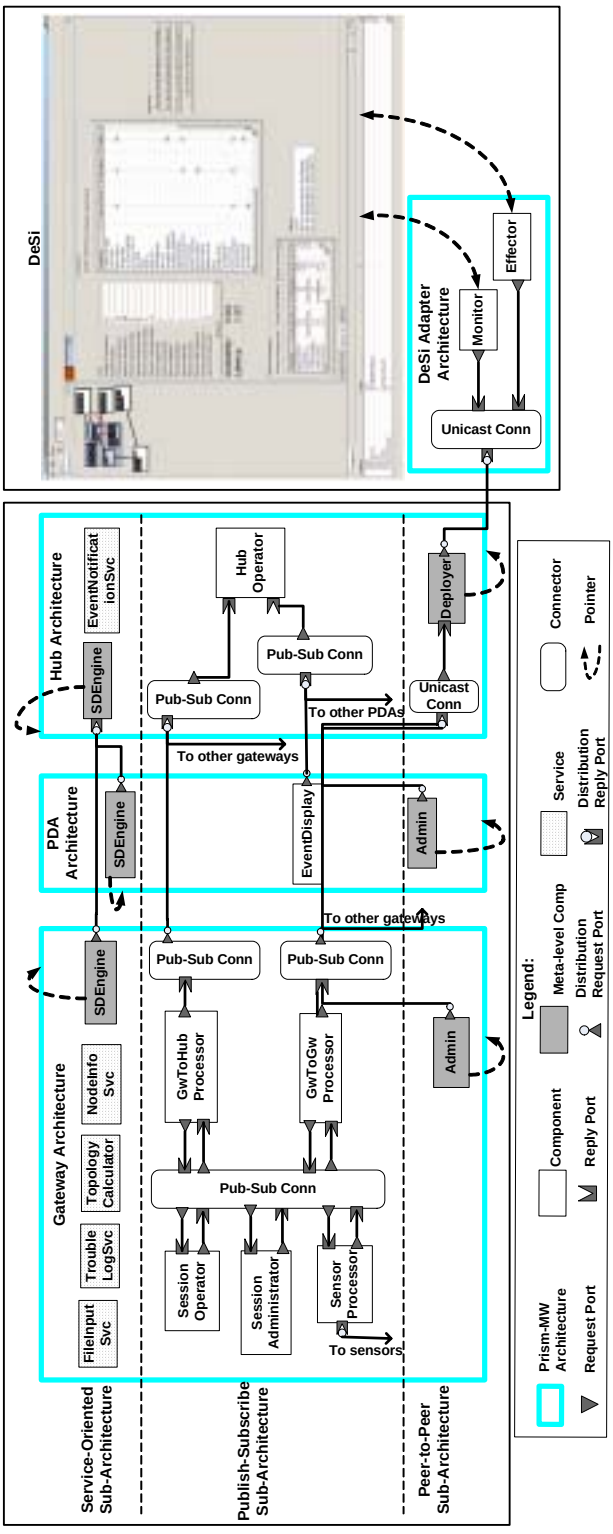


Figure 8-17: An abridged view of MIDAS's architecture that is monitored, analyzed, and adapted at runtime.

are started. We have built into DeSi the logic to ensure that the events are generated and sent in an order that does not result in a fault. We also leveraged the support for delivery guarantees (recall Section 7.1.6) to ensure that the events are delivered to the remote *Deployer* and *Admin* components in the correct order.

CHAPTER 9: Conclusions

As the distribution, decentralization, and mobility of computing environments grow, so does the influence of system's deployment architecture over the QoS dimensions of the system. In this dissertation, a novel extensible framework has been presented for improving a software system's QoS by finding the best deployment of software components. The framework is based on a QoS trade-off model and accompanying generic algorithms, where given the users' preferences for the desired levels of QoS, the most suitable deployment architecture is found. The framework has been realized via an integrated tool-suite composed of a visual deployment modeling and analysis environment, and an extensible architectural middleware platform.

9.1 Contributions

As mentioned earlier, this research is not the first effort of this kind. Several previous works [1,17,22,40] have realized the impact of a system's deployment architecture on its QoS and have explored techniques for finding an improved deployment of the system. However, these solutions are all either motivated by a particular application scenario or based on simplifying assumptions that make them inapplicable to most systems (e.g., single QoS dimension, single user, small system, and/or particular definition of a QoS dimension). Furthermore, since each approach has created its own, limited model of a system and domain-specific algorithms, it is extremely hard to compare and interchange the different approaches.

A primary contribution of our work has been to address these shortcomings by constructing a generic framework that can be tailored to the specific needs of a multitude of application scenarios. In the process, we have developed a highly expressive approach to specifying users' QoS preferences in terms of system parameters. We have devised and evaluated several algorithms that operate on the generic model and perform fine-grained trade-off analysis with respect to user preferences. The results of the evaluation have helped us in characterizing each algorithm. In turn, this has allowed us to determine the best algorithm for different classes of systems. Our implementation of the framework results in an environment that allows the system architect to bridge the system modeling, analysis, simulation, implementation, and deployment activities. Finally, the framework provides the means for comparing different solutions, which in turn paves the way for further research in this area.

9.2 Future Work

The framework presented in this dissertation can be extended and enhanced in several ways. Below we describe four directions in which we intend to expand this research.

9.2.1 Determining When to Redeploy

An underlying assumption in this dissertation has been that it is possible to determine when a distributed software system should be redeployed. While this is a reasonable assumption for a large class of systems that have well-defined usage or fluctuation patterns, it may not be feasible for systems where the monitored parameters undergo

significant and unpredictable fluctuations. As part of our future work we plan to investigate the applicability of our approach to unpredictable and ad hoc systems.

Another important issue is whether a certain amount of improvements in QoS is worth the amount of time and resources required to redeploy and reconfigure the system. Our current approach does not take into account the duration a (portion of) system may become unavailable due to its redeployment. We believe one potential solution is to account for the temporary degradations in QoS via a new utility function. For example, not only model users' preferences for the amount of improvements in QoS, but also their tolerance for the temporary disruptions in the services provisioned by the system.

9.2.2 Impact of architectural decisions on QoS properties.

In this dissertation we have focused on the impact of *deployment decisions* on a system's QoS properties. We hypothesize that a similar approach can be taken toward other types of architectural decisions. In our future work, we plan to extend the framework to allow for (1) the representation of other types of architectural decisions (e.g., selection of architectural style, composition of components), and (2) the improvement of QoS concerns by analyzing different types of such decisions. Our long-term goal is to devise a comprehensive framework that, given many, possibly conflicting architectural decisions under consideration, and a finite set of desirable QoS properties, can aid the software architect in making optimal decisions.

9.2.3 Architectural analysis

Static analysis of architectural models, such as the type of analysis provided by DeSi, can be useful for exploring the large space of possible solutions (i.e., deployment architectures) and finding one or more solutions that exhibit desired QoS properties. On the other hand, dynamic scenario-driven analysis, such as the type of analysis provided by XTEAM [9], is able to accurately capture the temporal variations in QoS properties. XTEAM (eXtensible Tool-chain for Evaluation of Architectural Models) is a suite of ADL extensions and model transformation engines targeted specifically at highly distributed, resource-constrained, and mobile computing environments. Architectural models that conform to the XTEAM ADL are constructed in an off-the-shelf meta-programmable modeling environment. XTEAM model translators transform these architectural models into executable simulations that furnish measurements and views of the executing system over time. These results allow an architect to better understand the consequences of architectural decisions, and weigh trade-offs between conflicting design goals.

As part of our future work, we plan to integrate DeSi with XTEAM, and thus take advantage of their respective strengths: DeSi's highly optimized and efficient algorithms will be used to determine a small number of candidate deployment architectures, while XTEAM's simulations will be used to generate a temporal view of each candidate deployment's QoS properties. The combination of static and dynamic anal-

ysis will increase the software architect's *confidence* in selecting the best deployment architecture.

9.2.4 Fault-tolerance

One common approach to improving certain system properties (e.g., fault tolerance, availability, durability) is by replicating software components. Note that, given finite computing resources, only a subset of software components can be replicated. At the same time, the improvements in certain QoS properties due to replication could degrade other QoS properties due to the overhead of keeping software replicas synchronized. In our future research, we plan to extend the deployment improvement framework to also analyze different replication strategies, and study their impact both on a system's deployment and on its QoS properties.

REFERENCES

1. M. C. Bastarrica, et al. A Binary Integer Programming Model for Optimal Object Distribution. *2nd Int'l. Conf. on Principles of Distributed Systems,* Amiens, France, Dec. 1998.

2. A. Carzaniga et. al. A Characterization Framework for Software Deployment Technologies. *Technical Report, Dept. of Computer Science, University of Colorado,* 1998.

3. E. Castillo, et al. *Building and Solving Mathematical Programming Models in Engineering and Science*. John Wiley & Sons, New York, NY, 2001.

4. J. Cook, A. Orso. MonDe: Safe Updating through Monitored Deployment of New Component Versions. *6th ESEC/FSE Workshop on Program Analsysis for Software Tools and Engineering*, Lisbon, Portugal, Sept. 2005.

5. J. Czyzyk, M. Mesnier, and J. Moré. The NEOS Server. *IEEE J. Comp. Science and Engineering*, pages 68-75, 1998.

6. E. Dashofy, A. van der Hoek, and R. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *ICSE 2002*, Orlando, FL, May 2002.

7. F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE Transactions on Software Engineering,* SE 2-2, pp 321-327, June 1976.

8. P. A. Dinda, et. al. The Architecture of the Remos System. *IEEE International Symposium on High Performance Distributed Computing (HPDC 2001),* San Francisco, CA, August 2001.

9. G. Edwards, S. Malek, and N. Medvidovic. Scenario-Driven Dynamic Analysis of Distributed Architectures. In proceedings of the *10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, Braga, Portugal, March 2007.

10. M. Ewing and E. Troan. The RPM Packaging System. In Proceedings of *the First Conference on Freely Redistributable Software*, Cambridge, MA, USA, February 1996. Free Software Foundation.

11. R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UCI, June 2000.

12. D. Garlan, S. Cheng, B. Schmerl. Increasing System Dependability through Architecture-based Self-repair. In R. de Lemos, C. Gacek, A. Romanovsky, eds., *Architecting Dependable Systems*, 2003.

13. J. Greenfield (ed.). UML Profile for EJB. *Public Review Draft JSR-000026*, Java Community Process, 2001.

14. R. Haas et. al. Autonomic Service Deployment in Networks. *IBM Systems Journal*, Vol. 42, No. 1, 2003.

15. R. S. Hall, D. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *International Conference in Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999.

16. HP OpenView, http://www.managementsoftware.hp.com/, 2005.

17. G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. *3rd Symposium on Operating System Design and Implementation*, New Orleans, Feb. 1999.

18. IBM Tivoli Composite Application Management System, http://www-306.ibm.com/software/tivoli/solutions/application-management/, 2005.

19. InstallShield, http://www.installshield.com, 2005.

20. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

21. M. Jones et al. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Symposium on Operating Systems Principles (SOSP)*, 1997.

22. T. Kichkaylo et al. Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques. *Int'l. Parallel and Distributed Processing Symposium.* April 2003.

23. D. Kreps. Game Theory and Economic Modeling. *Clarendon Press*, Oxford, 1990.

24. E. A. Lee. Embedded Software. In *Advances in Computers*, Ed Zelkowitz (Ed), Vol .56, Academic Press, 2002.

25. C. Lee, et al. A Scalable Solution to the Multi-Resource QoS Problem. *Proc. IEE Real-Time Systems Symposium*, 1999.

26. C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46-61.

27. C. Luer, and D. Rosenblum. UML Component Diagrams and Software Architecture - Experiences from the Wren Project. *1st ICSE Workshop on Describing Software Architecture with UML*, pages 79-82, Toronto, Canada, 2001.

28. S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. on Software Engineering*, Vol. 31, No. 4, March 2005.

29. S. Malek, M. Mikic-Rakic, and N. Medvidovic. A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In proceedings of the *3rd International Working Conference on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005.

30. S. Malek, N. Beckman, M. Mikic-Rakic, and N. Medvidovic. A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In *R. de Lemos, C. Gacek, and A. Romanowski, eds., Third Book on Architecting Dependable Systems*, Lecture Notes in Computer Science, Springer Verlog, 2005.

31. S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In proceedings of the *29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, Minnesota, May 2007.

32. S. Malek. A User-Centric Framework for Improving a Distributed Software System's Deployment Architecture. In proceedings of the *doctoral track at the 14th ACM SIGSOFT Symposium on Foundation of Software Engineering (FSE 2006)*, Portland, Oregon, November 2006.

33. S. Malek, C. Seo, and N. Medvidovic. Tailoring an Architectural Middleware Platform to a Heterogeneous Embedded Environment. In proceedings of the *6th International Workshop on Software Engineering and Middleware (SEM 2006)*, Portland, Oregon, November 2006.

34. S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic. Providing Middleware-Level Facilities to Support Architecture-Based Development of Software Systems in Pervasive Environments. In proceedings of the *4th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006)*, Melbourne, Australia, November 2006.

35. N. Medvidovic , and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), pp.70-93, January 2000.

36. N. Medvidovic, M. Mikic-Rakic, N.R. Mehta, S. Malek. Software Architectural Support for Handheld Computing. *IEEE Computer Special Issue on Handheld Computing*, pp. 66-73, September 2003.

37. N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *Proc. Int'l Conference on Software Engineering*, pp 178-187, Limerick, Ireland, June, 2000.

38. Microsoft Windows Installer, http://www.windows.com/download, 2005.

39. M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. In Proceedings of *the ACM/IFIP/ USENIX International Middleware Conference*, Rio de Janeiro, Brazil, June 2003.

40. M. Mikic-Rakic, S. Malek, N. Medvidovic. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. In proceedings of the *3rd Int'l. Working Conference on Component Deployment*, Grenoble, France, Nov. 2005

41. M. Mikic-Rakic and N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, May 2001.

42. M. Mikic-Rakic and N. Medvidovic. Support for Disconnected Operation via Architectural Self-Reconfiguration. *Int'l. Conf. on Autonomic Computing*, New York, May 2004.

43. M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *2nd Int'l. Working Conference on Component Deployment*, Edinburgh, UK, May 2004.

44. A. Mos, J. Murphy. "COMPAS: Adaptive Performance Monitoring of Component-Based Systems." *Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, May 2004, Edinburgh, Scotland, UK.

45. R. Neugebauer and D. McAuley. Congestion Prices as Feedback Signals: An Approach to QoS Management. *Proc. ACM SIGOPS European Workshop*, 2000.

46. Object Management Group. The Unified Modeling Language v1.4. Tech. report, 2001.

47. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture Based run time Software Evolution. *International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

48. A. Orso, D. Liang, M.J. Harrold, and R. Lipton. Gamma System: Continous Evolution of Software after Deployment. *International Symposium on Software Testing and Analysis (ISSTA 2002)*. July 2002.

49. D. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, Vol. 17, No. 4, pp. 40-52, October 1992.

50. L. L. Peterson and B. S. Davie. Computer Networks. Morgan Kaufmann Publishers, 2000.

51. V. Poladian et al. Dynamic Configuration of Resource-Aware Services. *ICSE 2004*, Edinburgh, Scotland, 2004.

52. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1995.

53. D. Salomon. Data Compression: The Complete Reference. *Springer Verlag*, December 1997.

54. D. Schmidt et. al. Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers. *Kluwer Journal of Realtime Systems*, Volume 21, Number 2, 2001.

55. C. Seo, S. Malek, N. Medvidovic. A Generic Approach for Estimating the Energy Consumption of Component-Based Distributed Systems. *Tech. Report USC-CSE-2005-506*, 2005.

56. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.

57. J. P. Sousa, and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Proc. Working IEEE/IFIP Conference on Software Architecture*, pp. 29-43, Montreal, Canada, August 2002.

58. W. Stallings. *Cryptography and Network Security.* Prentice Hall, Englewood Cliffs, NJ, 2003.

59. SUN Microsystems. SunOS 5.8 Manual, 2000.

60. System Modeling Language (SysML) Specification, Version 1.0 alpha, SysML Partners.

61. C. Szyperski. Component Software – Beyond Object-Oriented Programming. Addison-Wesley, ACM Press, 1998

62. A. Tanenbaum. Computer Networks. *Prentice Hall*, Englewood Cliffs, New Jersey.

63. R. N. Taylor, N. Medvidovic, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, Vol. 22, pp. 390-406, June 1996.

64. B. Tierney, et. al. A Monitoring Sensor Management System for Grid Environments. *IEEE International Symposium on High Performance Distributed Computing*, Pittsburgh, PA, August 2000.

65. H. R. Varian. *Intermediate Microeconomics: A Modern Approach.* W. W. *Norton*, 6th edition, 2003.

66. Web Services Description Language, http://www.w3.org/TR/wsdl, 2005.

67. L. A. Wolsey. *Integer Programming*. John Wiley & Sons, New York, NY, 1998.