Improving a Distributed Software System's Quality of Service via Redeployment

Sam Malek[1], Nenad Medvidovic[2], and Marija Mikic-Rakic[3]

Abstract –A distributed system's allocation of software components to hardware nodes (i.e., deployment architecture) can have a significant impact on its quality of service (QoS). For a given system, there may be many deployment architectures that provide the same functionality, but with different levels of QoS. The parameters that influence the quality of a system's deployment architecture are often not known before the system's initial deployment and may change at runtime. This means that redeployment of the software system may be necessary to improve the system's QoS properties. This paper presents and evaluates a framework aimed at finding the most appropriate deployment architecture of an arbitrarily large, distributed software system with respect to multiple, possibly conflicting QoS dimensions. The framework supports formal modeling of the problem and provides a set of tailorable algorithms for improving a system's deployment. We have realized the framework on top of an infrastructure for architectural analysis, implementation, deployment, monitoring, and runtime adaptation. The framework has been evaluated for precision and execution-time complexity on a large number of distributed system scenarios, as well as in the context of two third-party families of distributed applications.

Index Terms—Software Architecture, Software Deployment, Quality of Service, Self-Adaptive Software

I. INTRODUCTION

Consider the following scenario, which addresses distributed deployment of personnel in cases of natural disasters, search-and-rescue efforts, and military crises. A computer at "Headquarters" gathers information from the field and displays the current status and locations of the personnel, vehicles, and obstacles. The headquarters computer is networked to a set of PDAs used by "Team Leads" in the field. The team lead PDAs are also connected to each other and to a large number of "Responder" PDAs. These devices communicate with one another, and potentially with a large number of wireless sensors deployed in the field, 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: exchanging information, viewing the current field map, managing the resources, etc.

Such an application is frequently challenged by the fluctuations in the system's parameters: network disconnections, bandwidth variations, unreliability of hosts, etc. Furthermore, the different users' usage of the functionality (i.e., services) provided by the system and the users' quality of service (QoS) preferences for those services will differ, and may change over time. For example, in the case of a natural disaster (e.g., wildfire) scenario, "Team Lead" users may require a secure and reliable messaging service with the "Headquarters" when exchanging search-and-rescue plans. On the other hand, "Responder" users are likely to be more interested in having a low latency messaging service when sending emergency assistance requests.

1 Department of Computer Science, George Mason University, Fairfax, VA, 22030, USA. E-mail: smalek@gmu.edu

2 Computer Science Department, University of Southern California, Los Angeles, CA, 90089, USA. E-mail: neno@usc.edu

3 Google Inc., 604 Arizona Ave., Santa Monica, CA, 90401, USA. E-mail: marija@google.com

For any such large, distributed system, many *deployment architectures* (i.e., mappings of software components onto hardware hosts) will be typically possible. While all of those deployment architectures may provide the same functionality, some will be more effective in delivering the desired level of service 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 among the components involved in delivering the service occur either locally or over reliable and capacious network links. The task of quantifying the quality of a system's deployment and determining the most effective deployment architecture becomes quickly intractable for a human engineer 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 host Y). Further exacerbating the problem is the fact that many of the parameters that influence the optimal distribution of a system may not be known before the system's initial deployment and are likely to change during the system's lifetime.

In this paper, we consider the problem of quantifying the quality of a system's deployment and finding a deployment architecture such that the QoS preferences stated by a collection of distributed end-users are met; in other words, the overall utility of the system to all its users is maximized. Our objective is for the solution to be applicable in a wide range of application scenarios (i.e., differing numbers of users, hardware hosts, software components, application services, QoS dimensions, etc.). Providing a widely applicable solution to this problem is difficult for several reasons:

- In the general case, the number of possible deployments for a given distributed software system is exponential in the number of the system's components. The amount of computation required for exploring a system's deployment space may thus be prohibitive, even for moderately large systems.
- A very large number of parameters influence a software system's QoS dimensions. In turn, many services and their corresponding QoS dimensions influence the users' satisfaction. Developing generic solutions that can then be customized for any application scenario is non-trivial.
- Different QoS dimensions may be conflicting, and users with different priorities may have conflicting QoS preferences. Fine-grain trade-off analysis among the different dimensions and/or preferences is very challenging without relying on simplifying assumptions (e.g., particular definition of a QoS objective or predetermined, fixed constraints).
- Different application scenarios are likely to require different algorithmic approaches. For example, a system's size, the users' usage of the system, stability of system parameters, and system distribution characteristics may influence the choice of algorithm.

- Traditional software engineering tools are not readily applicable to this problem. Instead, engineers must adapt tools intended for different purposes (e.g., constraint satisfaction or multidimensional optimization) to the deployment improvement problem, which limits the potential for reuse and cross-evaluation of the solutions.

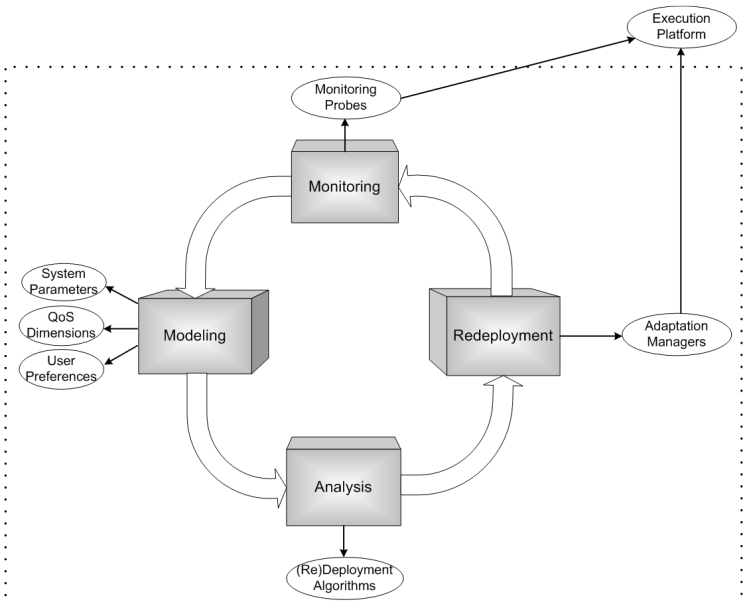


Figure 1. Framework for continuous QoS driven deployment improvement. While the execution platform is an integral part of the framework's realization, its details have been published previously and it is outside this paper's scope.

To deal with the above challenges, we have developed a tailorable framework for quantifying, continuously monitoring, and *improving* a software-intensive system's deployment architecture. This framework, depicted in Figure 1, is a specialization of our architecture-based dynamic system adaptation framework [38, 39]. It consists of the following activities:

(1) *Modeling* – An extensible architectural model supports inclusion of arbitrary *system parameters* (software and hardware), definition of arbitrary *QoS dimensions* using those parameters, and specification of QoS *preferences* by system users. Using the model, utility of any system's deployment can be quantified.

(2) Analysis – Several tailored *algorithms* allow rapid exploration of the space of possible deployment architectures, the effects of changes in system parameters on a system's QoS, as well as the effects of system downtime incurred during runtime adaptation. No other approach known to us provides such capabilities in tandem.

(3) *(Re)Deployment* – The framework controls the *execution platform* to manage the activities required for the initial deployment as well as runtime redeployment of the software system onto its hardware hosts.

(4) *Runtime Monitoring* – The *execution platform* is instrumented with *probes* of arbitrary hardware and software system parameters, which are used in concert with *gauges* [10] to continuously assess a system's satisfaction of QoS requirements and possibly initiate the deployment improvement process.

The focus of this paper is on the modeling and analysis components of the framework. We have previously realized the monitoring and redeployment facilities on top of an existing middleware platform, called

Prism-MW, the details of which can be found elsewhere [25][28]. The key contributions of this paper are (1) the theoretical underpinnings for modeling, quantifying, assessing, and improving a software system's deployment; (2) support for quantitative exploration of a system's, typically very large, deployment space; and (3) the accompanying tool suite that enables software engineers to use our framework. We provide an overview of the framework's redeployment and monitoring components only to the extent necessary for the reader to follow its evaluation. We have used the full implementation of the framework's four activities to empirically evaluate the feasibility of the approach in both simulated and real-world settings.

For any given application, 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 hardware and software parameters from a distributed application and by the users' preferences for the QoS dimensions of each application service. Using these values, the utility (i.e., the cumulative satisfaction with the system by all its users) of a given deployment is calculated. Finally, one of the algorithms supplied by the framework is used to find a deployment architecture that improves the overall utility.

The model and algorithms underlying this work are empirically analyzed for execution-time complexity and accuracy on a large number of distributed system scenarios. In particular, the algorithms are compared with the current state-of-the-art approaches in this area. For the past two years, the framework has been in use by a third-party organization, in the context of their family of sensor network applications [28]. We report on this, as well as our own experience with applying the framework to another software system developed in collaboration with an external organization.

The remainder of the paper is organized as follows. Section II provides some background on the challenges of assessing a system's deployment architecture, and our approach to overcoming them. We then describe in detail two components of the framework that are the focus of this paper: Section III details our system model, and Section IV describes the QoS-driven architectural analysis, including the redeployment algorithms we have devised. We provide only brief overviews of the other two components of the framework in order to allow the reader to follow the framework's evaluation: Section V discusses our approach to effecting system redeployment; and Section VI discusses the realization of the framework's monitoring capability. Our experience and evaluation results are provided in Section VII, while Section VIII discusses the related work. The paper concludes with a recap of lessons learned and an outline of future work.

II. ASSESSING DEPLOYMENT ARCHITECTURES

For illustration, let us consider the very simple software system conceptually depicted in Figure 2, consisting of one application-level service (*ScheduleResources*) provided to two users by two software components (*ModifyResourceMap* and *ResourceMonitor*) that need to be deployed on two network hosts (a laptop and a PDA). In the absence of any other constraints, the system has four possible deployments (i.e., *number of hosts*$^{\text{number of components}} = 2^2$) that provide the same functionality. Two of the deployments correspond to the situations where the two components are collocated on the same host, while the other two deployments correspond to the situations where each component is deployed on a separate host.

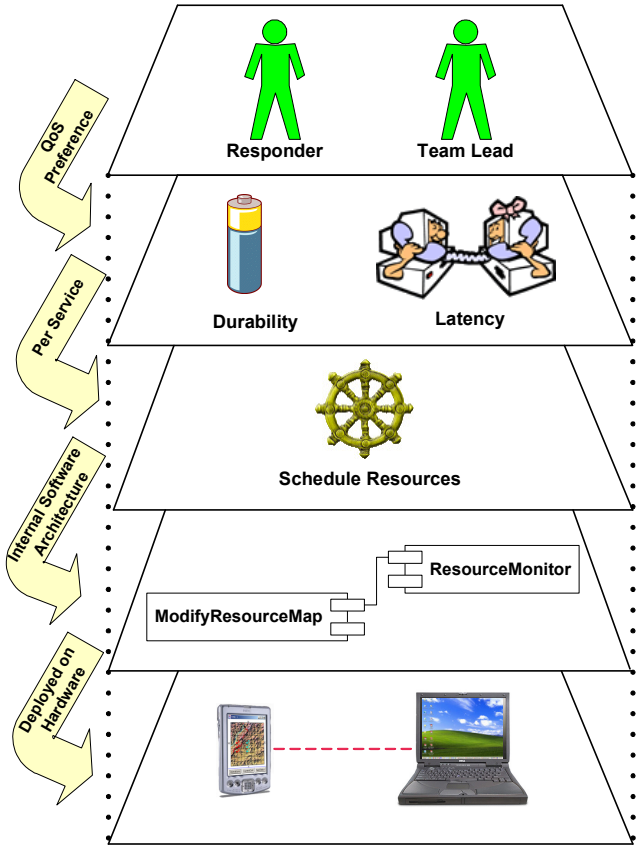


Figure 2. A hypothetical application scenario.

Let us assume that it is possible to measure (or estimate, provided that appropriate models and analytical tools are available to the engineers) the four deployments' QoS properties, as shown in Figure 3a.[4] It is clear that deployment *Dep1* has the shortest latency, while deployment *Dep3* has the highest durability (defined in this case as the inverse of the system's energy consumption rate). In other words, *Dep1* is the optimal deployment with respect to *latency,* while *Dep3* is the optimal deployment with respect to *durability*. If the objective is to minimize the latency and at the same time maximize the durability of the system, none of the four deployments can be argued to be optimal. This phenomenon is known as *Pareto Optimal* in multidimensional optimization [51].

For any software system composed of many users and services, the users will likely have varying QoS preferences for the system's services. To deal with the QoS trade-offs, we can leverage the users' QoS prefe-

4 This assumption is essential in order to be able to consider any objective strategy for assessing and improving a software system's deployment. We assume that, even if it is not feasible to quantify a QoS property as a whole, it is possible to quantify different aspects of it. For example, while it may not be possible to represent a system's security as a single numerical quality, it is possible to quantify different aspects of security for the system (e.g., security of communication, encryption, and authentication).

rences (i.e., the *utility* that achieving a given level of quality for a given service would have for a user). As an example, Figure 3b shows the *Team Lead*'s utility for the rates of change in the latency and durability of *ScheduleResources*. While the figure shows linear functions, users may express their QoS preferences in many ways, including by using much less precise phrasing. Any deployment improvement model must be able to capture effectively such preferences. We will discuss this issue further in Sections III and VII.

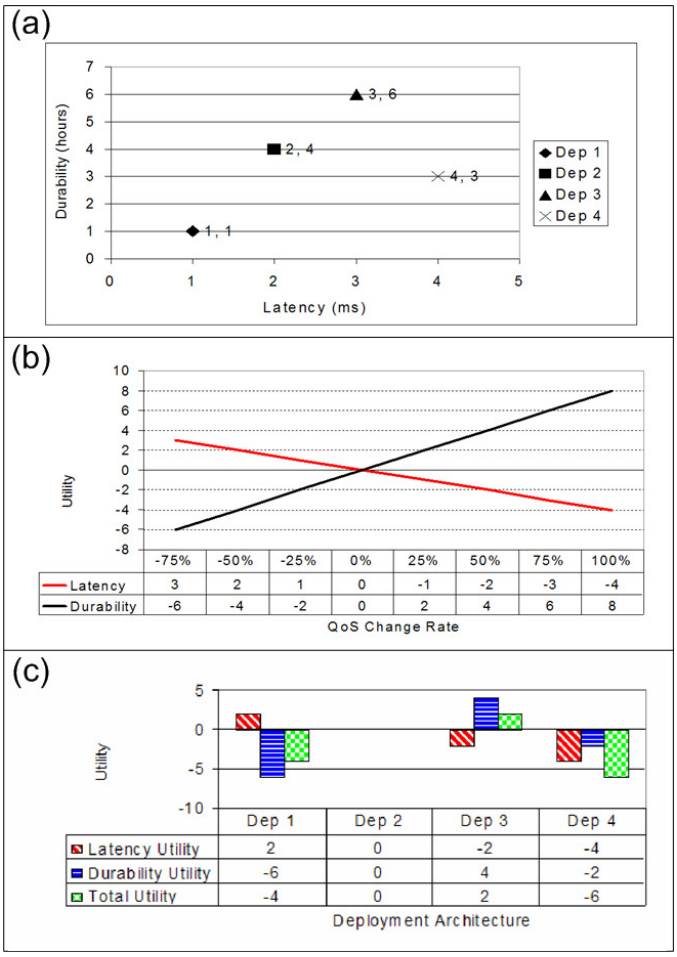


Figure 3. (a) Latency and durability measurements for the four candidate deployments of the system in Figure 2. (b) *Team Lead*'s utility for latency and durability of the *ScheduleResources* service. (c) Utility calculation for each deployment architecture.

In this particular example, Figure 3c shows the results of assessing the four deployments based on *Team Lead*'s utility functions. As shown in Figure 3c, assuming *Dep2* is the current deployment of the system, the quality of the other three deployments can be calculated as follows: (1) from Figure 3a find the rate of change in each QoS dimension from the current deployment to the new deployment, (2) look up the utility for that rate of change in Figure 3b, and (3) aggregate the utilities, as shown in Figure 3c. In this (hypothetical) case, *Dep3* provides the highest total utility to *Team Lead* and can be considered to be the optimal deployment for her. In a multi-user system, the other users also have to be considered in a similar manner, and all of the users' utilities must be aggregated to determine a globally optimal deployment.[5]

However, even this solution quickly runs into problems. Consider the following, slightly more complex variation of the scenario depicted in Figure 2, where we introduce a single additional element to each layer of the problem: we end up with three users who are interested in three QoS dimensions of two services provided

5 Note that, according to our definition of utility, a globally optimal deployment may result in suboptimal QoS levels for specific services and specific users.

by three software components deployed on three hosts. In this case, the engineer will have to reason about 18 utility functions (3 users * 3 QoS dimensions * 2 services) across 27 different deployments (3 hosts and 3 components, or 3^3 deployments). The problem quickly becomes intractable for a human engineer. This has been the primary motivation for this research.

III. DEPLOYMENT MODEL

Modeling has been a key thrust of software architecture research and practice, resulting in a large number of specialized architecture description languages (ADLs) [21,23] as well as the broadly scoped UML [37]. These languages enable the specification of a software system's architectural structure, behavior, and interaction characteristics. Languages such as AADL [1] and UML [37] even allow capturing the hardware hosts on which the software components are deployed. However, none of the existing languages provide the modeling constructs necessary for properly examining a system's deployment architecture in terms of its, possibly arbitrarily varying, QoS requirements. This motivated us to develop our framework's underlying model, which provides the foundation for precisely quantifying the quality of any system's deployment and for formally specifying the deployment improvement problem. This model is intended to be used in conjunction with traditional software architecture modeling approaches; as a proof-of-concept we have demonstrated the model's integration with two third-party ADLs, xADL and FSP [8,9].

A primary objective in designing the framework was to make it practical, i.e., to enable it to capture realistic distributed system scenarios and avoid making overly restrictive assumptions. To that extent, we want to avoid prescribing a predefined number of system parameters or particular definitions of QoS dimensions. Therefore, the framework provides: (1) a minimum skeleton structure for formally formulating the basic concepts that are common across different instances of our problem, and (2) a technique for representing the variations among the different instances of this problem by refining and extending this skeleton.

A. BASIC MODELING CONSTRUCTS

We model a software system's deployment architecture as follows:

1. A set H of hardware nodes (hosts), a set HP of associated parameters (e.g., available memory or CPU on a host), and a function $hParam_H: HP \rightarrow R$ that maps each parameter to a value.[6]
2. A set C of software components, a set CP of associated parameters (e.g., required memory for execution or JVM version), and a function $cParam_C: CP \rightarrow R$ that maps each parameter to a value.

6 *For simplicity, we show the functions as mappings to real numbers. This need not be the case for certain parameters relevant in a given context. Our model will support such additional mappings in the same manner.*

3. A set *N* of physical network links, a set *NP* of associated parameters (e.g., available bandwidth, reliability of links), and a function $nParam_N: NP \rightarrow R$ that maps each parameter to a value.
4. A set *I* of logical interaction links between software components in the distributed system, a set *IP* of associated parameters (e.g., frequency of component interactions, average event size), and a function $iParam_I: IP \rightarrow R$ that maps each parameter to a value.
5. A set *S* of services, and a function $sParam_{S,\{H \cup C \cup N \cup I\}}: \{HP \cup CP \cup NP \cup IP\} \rightarrow R$ that provides values for service-specific system parameters. An example such parameter is the number of component interactions resulting from an invocation of an application service (e.g., "find best route").
6. A set *DepSpace* of all possible deployments (i.e., mappings of software components *C* to hardware hosts *H*), where $|DepSpace| = |H|^{|C|}$.
7. A set *Q* of QoS dimensions, and associated functions $qValue_Q: S \times DepSpace \rightarrow R$ that quantify a QoS dimension for a given service in the current deployment mapping. For instance, the information presented in Figure 3a can be modeled using two *qValue* functions (for latency and durability).
8. A set *U* of users, and a function $qUtil_{U,Q,S}: R \rightarrow [MinUtil, MaxUtil]$ that denotes a user's preferences (in terms of the achieved utility) for a given level of QoS. In other words, *qUtil* returns the utility, established by a user, for a rate of change in the quality of a service. A user may denote a utility between the range of *MinUtil* and *MaxUtil*. Relative importance of different users is determined by *MaxUtil*. In general, a larger value of *MaxUtil* indicates higher importance (or relative influence) of the user, allowing us to specify a particular user priority scheme, including for example completely subordinating the priorities of some users to those of others. As an example, the information presented in Figure 3b can be modeled using two *qUtil* functions, one for the *Team Lead*'s latency preferences and another for her durability preferences.
9. A set *PC* of parameter constraints, and a function $pcSatisfied_{PC}: DepSpace \rightarrow \begin{cases} 1 \\ 0 \end{cases}$ that given a constraint and a deployment architecture, returns 1 if the constraint is satisfied and 0 otherwise. For example, if the constraint is "bandwidth satisfaction", the corresponding 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. The set *PC* and function *pcSatisfied* could also be used to specify QoS constraints.
10. Using the following two functions, the deployment of components can be restricted:

$$loc: C \times H \rightarrow \begin{cases} 1 \; if \; c \in C \; can \; be \; deployed \; onto \; h \in H \\ 0 \; if \; c \in C \; cannot \; be \; deployed \; onto \; h \in H \end{cases}$$

$$colloc: C \times C \rightarrow \begin{cases} 1 \; if \; c1 \in C \; has \; to \; be \; on \; the \; same \; host \; as \; c2 \in C \\ -1 \; if \; c1 \in C \; cannot \; be \; on \; the \; same \; host \; as \; c2 \in C \\ 0 \; if \; there \; are \; no \; restrictions \end{cases}$$

For example, *loc* can be used to restrict the deployment of a computationally expensive component to hosts with sufficient CPU power, while *colloc* may prohibit two components providing the primary and backup of the same service to be collocated on the same host.

Note that some elements of the framework model are intentionally left "loosely" defined (e.g., the sets of system parameter or QoS). These elements correspond to the many and varying factors that are found in different distributed applications. As we will show below, when the framework is instantiated, the system architect specifies these loosely defined elements.

For brevity we use the following notations in the remainder of the paper: H_c is a host on which component c is deployed; $I_{c1,c2}$ is an interaction between components *c1* and *c2*; $N_{h1,h2}$ is a network link between hosts *h1* and *h2*; finally, C_s and H_s represent respectively a set of components and hosts that participate in provisioning the service s.

Figure 4 shows the formal definition of the utility of a single system's deployment, and the deployment optimization problem based on our framework model. The function *overallUtil* represents the overall satisfaction of the users with the QoS delivered by the system. The goal is to find a deployment architecture that maximizes *overallUtil* and meets the constraints on location, collocation, and system parameters (recall items 9 and 10 above). Note that the scope of our problem is limited to finding and effecting an improved software deployment. Other types of adaptation decisions that may also impact a system's QoS (e.g., changing the amount of system resources at the software component's disposal, or replacing one version of a component with another) are complementary to our work, but we do not consider them here.

B. MODEL INSTANTIATION

As mentioned earlier, some aspects of our framework's model have been intentionally left loosely de-

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

$$overallUtil(d,d') = \sum_{u \in U} \sum_{s \in S} \sum_{q \in Q} qUtil_{u,s,q}(\Delta q) \text{ , where } \Delta q = \frac{qValue_q(s,d') - qValue_q(s,d)}{qValue_q(s,d)}$$

is maximized, and the following conditions are satisfied:

1. $\forall c \in C \qquad loc(c, H_c) = 1$
2. $\forall c1 \in C \quad \forall c2 \in C \qquad if \ \ (colloc(c1,c2) = 1) \Rightarrow (H_{c1} = H_{c2})$
   $if \ \ (colloc(c1,c2) = -1) \Rightarrow (H_{c1} \neq H_{c_l})$
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, some of these deployments may not satisfy one or more of the above three conditions.

Figure 4. Problem definition.

fined. To actually use the framework, one needs to precisely specify an application scenario. Framework instantiation is the process of configuring the framework model for an application scenario. We illustrate the instantiation using four QoS dimensions: availability, latency, communication security, and energy consumption. Note that any QoS dimension that can be quantitatively estimated can be used in our framework..

Our framework does not place any restrictions a priori on the manner in which QoS dimensions are estimated. This allows an engineer to tailor the framework to the application domain and to her specific needs. Our instantiation of the framework model, shown in Figure 5 and discussed below, is based on our experience with emergency response and wireless sensor network-based software systems [25,27,28], such as those described in Section I. In other application domains, some of the simplifying assumptions we have made may not hold, and other estimations of QoS may be more appropriate.

The first step in instantiating the framework is to define the relevant system parameters. Item 1 of Figure 5 shows a list of parameters that we have identified to be of interest for estimating the four selected QoS dimensions in mobile emergency response setting. If additional parameters are found to be relevant, they can be similarly instantiated in our framework. Once the parameters of interest are specified, the parameter realization functions (e.g., *hParam*, *cParam*) need to be defined. These functions can be defined in many ways: by monitoring the system, by relying on system engineers' knowledge, by extracting them from the architectural description, etc. For example, $hParam_{h1}$ *(hostMem)* designates the available memory on host *h1*.

A software system's *availability* is commonly defined as the degree to which the system is operational when required for use [16,41]. In the context of mobile software systems, where a common source of failure is the network, we estimate availability in terms of successfully completed inter-component interactions in the system. Item 2 of Figure 5 estimates the availability for a single service *s* in a given deployment *d*. In other domains, such as data centers, other system parameters (e.g., unavailability due to high workload) may need to be considered.

A software service's *latency* is commonly defined as the time elapsed between making a request for service and receiving the response [16,40]. The most common causes of communication delay in a distributed mobile system are the network transmission delay, unreliability of network links, and low bandwidth. Item 3 of Figure 5 estimates latency for a service *s*. Note that in our specification of latency we did not consider the computational delay associated with each software component's execution. This can be justified, e.g., in the context of emergency response mobile systems, where the implementations of system services are typically relatively simple and network communication is the predominant source of delay. As before, in other domains, such as data centers or scientific applications, a reasonable estimation of latency would certainly need

to account for the execution time of software components. An important property of our framework is that it does not prevent one from including a more elaborate and appropriate estimation of a QoS dimension.

A major factor in the *security* of distributed mobile systems is the level of encryption supported in remote communication (e.g., 128-bit vs. 40-bit encryption) [48]. This is reflected in item 4 of Figure 5. This formulation assumes that a single encryption algorithm is used, where the size of the key indicates the difficulty of breaking ciphered data. In cases where this assumption is not true or multiple encryption algorithms are available, the model would have to be extended to account for the unique behavior of each algorithm. Regardless of the definition of security used, system architects would follow the procedure described here.

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 participating in providing the service. Item 5 of Figure 5 estimates energy consumption of service *s*. For ease of exposition, we have provided a simplified definition of energy consumption; a more sophisticated model can be found in our recent work [44,45].

The definitions in Figure 5 are intended to serve as an illustration of how QoS dimensions are estimated and used in our framework. We do not argue that these are the only, "correct", or even most appropriate

1. System parameters

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

2. Availability:
$$qValue_{availability}(s,d) = \sum_{c1 \in C_s} \sum_{c2 \in C_s} \left(sParam_{s,I_{c1,c2}}(freq) \times nParam_{N_{H_{c1},H_{c2}}}(rel) \right)$$

3. Latency:
$$qValue_{latency}(s,d) = \sum_{c1 \in C_s} \sum_{c2 \in C_s} \left(sParam_{s,I_{c1,c2}}(freq) \times nParam_{N_{H_{c1},H_{c2}}}(td) + \frac{sParam_{s,I_{c1,c2}}(freq) \times sParams_{s,I_{c1,c2}}(evtSize)}{nParam_{N_{H_{c1},H_{c2}}}(bw) \times nParam_{N_{H_{c1},H_{c2}}}(rel)} \right)$$

4. Communication security:
$$qValue_{security}(s,d) = \sum_{c1 \in C_s} \sum_{c2 \in C_s} \left(sParam_{s,I_{c1,c2}}(freq) \times sParam_{s,I_{c1,c2}}(evtSize) \times nParam_{N_{H_{c1},H_{c2}}}(enc) \right)$$

5. Energy consumption:
$$qValue_{energy}(s,d) = \sum_{c1 \in C_s} \sum_{c2 \in C_s} \left(\frac{sParam_{s,I_{c1,c2}}(freq) \times sParam_{s,I_{c1,c2}}(evtSize)}{nParam_{N_{H_{c1},H_{c2}}}(commEnrCons)} + cParam_{s,I_{c1,c2}}(freq) \times \left(\frac{cParam_{c1}(opcodeSize)}{hParam_{H_{c1}}(hostEnrCons)} + \frac{cParam_{c2}(opcodeSize)}{hParam_{H_{c2}}(hostEnrCons)} \right) \right)$$

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. Framework instantiation example.

quantifications. As indicated in the preceding discussion, our framework can accommodate arbitrary estimation of these as well as other QoS dimensions, so long as they are quantifiable.

To illustrate parameter constraints we use the memory available on each host. The constraint in item 6 of Figure 5 specifies that the cumulative size of the components deployed on a host may not be greater than the total available memory on that host. Other constraints are included in an analogous manner.

As the reader may recall from the previous section, we also need to populate the set *S* with the system services and the set *U* with the users of the system. Finally, each user's preferences are determined by defining the function *qUtil*. The users define the values this function takes based on their preferences, with the help of our tool support described below.

IV. DEPLOYMENT ANALYSIS

Once a system's architectural model is completed, the model can be analyzed for properties of interest. Traditionally, architectural analysis has been the prevailing reason for using ADLs [21]. The objective of our framework's analysis activity is to ensure an effective deployment of the modeled system, both prior to and during the system's execution. This analysis is based on a set of algorithms specifically developed for this purpose. Determining an effective deployment for a system is challenging for reasons discussed in Section I: the problem space is exponential, while a very large set of system parameters must be taken into account and must satisfy arbitrary constraints. Because of this, most of our algorithms are heuristic-based and provide approximate solutions. We have developed multiple algorithms in order to best leverage the characteristics of different scenarios involving varying QoS requirements, user preferences, and hardware and/or software performance characteristics. These algorithms provide the foundation for an automated deployment analyzer facility that is a centerpiece of our framework. In this section, we first present our deployment improvement algorithms and then discuss other factors that impact a system's preferred deployment architecture.

A. DEPLOYMENT IMPROVEMENT ALGORITHMS

The deployment optimization problem as we have stated it 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 goal has been to devise reusable algorithms that provide accurate results (assuming accurate model parameter estimates are available and appropriate QoS dimension quantifications are provided) regardless of the application scenario. A study of strategies applicable to this problem has resulted in four algorithms to date, where each algorithm has different strengths and is suited to a particular class of systems. Unlike previous works, which depend on the knowledge of specific system parameters

[2,3,15,18], we have introduced several heuristics for improving the performance and accuracy of our algorithms independently of the system parameters. Therefore, regardless of the application scenario, the architect simply executes the algorithm most suitable for the system (e.g., based on the size of the system or stability of parameters, further discussed in Section VII.A.6) without any modification.

Of the four approaches that we have used as the basis of our algorithms, two (Mixed-Integer Nonlinear Programming, or MINLP, and Mixed Integer Linear Programming, or MIP [51]) are best characterized as *techniques* developed in operations research to deal with optimization problems. They are accompanied by widely used algorithms and solvers. We tailor these techniques to our problem, and thereby improve their results. The remaining two approaches (greedy and genetic) can be characterized as generally applicable *strategies*, which we have employed in developing specific algorithms tailored to our problem.

1. Mixed-Integer Nonlinear Programming (MINLP) Algorithm

The first step in representing our problem as a MINLP problem is defining the decision variables. We define decision variable $x_{c,h}$ to correspond to the decision of whether component c is to be deployed on host h. Therefore, we need $|C| \times |H|$ binary decision variables, where $x_{c,h}$=1 if component c is deployed on host h, and $x_{c,h}$=0 otherwise.

The next step is defining the objective function, which in our case is to maximize the *overallUtil* function, as shown in Eq. 1 of Figure 6. The definition of *overallUtil* is the same as in Figure 4. However, note that the *qValue* functions of our instantiated model (recall Figure 5) are now rewritten to include the decision variables $x_{c,h}$. This is illustrated for the availability dimension in Eq. 2 of Figure 6.

Maximize $$overallUtil(d,d') = \sum_{u \in U} \sum_{s \in Sq \in Q} \sum \left(qUtil_{u,s,q} \left(\frac{qValue_q(s,d') - qValue_q(s,d)}{qValue_q(s,d)} \right) \right)$$ **Eq. 1**

$$qValue_{availability}(s,d) = \sum_{h1 \in H_s} \sum_{h2 \in H_s} \sum_{c1 \in C_s} \sum_{c2 \in C_s} \left(sParam_{s,I_{c1,c2}}(freq) \times nParam_{N_{H_{c1},H_{c2}}}(rel) \times x_{h1,c1} \times x_{h2,c2} \right)$$ **Eq. 2**

… other *qValue* functions …

Subject to $$\sum_{h \in H} \sum_{c \in C} x_{c,h} = 1$$ **Eq. 3**

$$\sum_{h \in H} \sum_{c \in C} \left(x_{c,h} \times cParam_c(compMem) \right) \leq hParam_h(hostMem)$$ **Eq. 4**

… other locational and system constraints …

Figure 6. MINLP representation of the deployment problem.

Finally, we need to specify the constraints. We have depicted two common constraints: Eq. 3 enforces the constraint that a single software component can only be deployed on a single host and Eq. 4 enforces the memory constraint that was defined previously in our instantiated model (item 6 of Figure 5).

The product of variables in Eq. 2 of Figure 6 demonstrates why the deployment problem is inherently non-linear. There is no known algorithm for solving a MINLP problem optimally other than trying every possible deployment. Furthermore, for problems with non-convex functions (such as ours), MINLP solvers are not guaranteed to find and converge on a solution [51]. Finally, given the non-standard techniques for solving MINLP problems, it is hard to determine a complexity bound for the available MINLP solvers.[7] For all of these reasons, we needed to investigate other options.

2. *Mixed-Integer Linear Programming (MIP) Algorithm*

An important characteristic of MIP problems is that they can be solved to find the optimal solution. We have leveraged a technique for transforming the above MINLP problem into MIP by adding new "auxiliary" variables. We introduce $|C|^2 \times |H|^2$ new binary decision variables $t_{c1,h1,c2,h2}$ to the specification formula of each QoS, such that $t_{c1,h1,c2,h2}$=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 add the following three constraints:

$$t_{c1,h1,c2,h2} \le x_{c1,h1}, \quad t_{c1,h1,c2,h2} \le x_{c2,h2}, \text{and } 1 + t_{c1,h1,c2,h2} \ge x_{c1,h1} + x_{c2,h2}$$

This allows us to remove the multiplication of decision variables from the *qValue* functions (e.g., Eq. 2 of Figure 6). However, this transformation significantly increases the complexity of the original problem. Since MIP solvers use branch-and-bound to solve a problem efficiently, our problem has the upper bound of:

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

One heuristic that we have developed is to assign a higher priority to *x* variables and lower priority to *t* variables. Thus, we are able to reduce the complexity of the algorithm to $O(2^{|H||C|})$: after solving the problem for the *x* variables, the values of *t* variables trivially follow from the three constraints introduced above. Finally, the constraint that each component can be deployed on only one host (Eq. 3 of Figure 6) allows for significant pruning of the branch-and-bound tree, thus reducing the complexity of our problem to $O(|H|^{|C|})$.

By fixing some components to selected hosts, the complexity of the exact algorithm reduces to

$$O\left(\prod_{c \in C} \sum_{h \in H} loc(c,h)\right)$$

[7] All state-of-the-art MINLP solvers are based on confidential algorithms, which are claimed to run in polynomial time [6].

Similarly, specifying that a pair of components c_i and c_j have to be collocated on the same host could further reduce the algorithm's complexity.

Our heuristics significantly reduce the complexity of the MIP problem. However, as we will see in Section VII, even after this reduction, the MIP algorithm remains computationally very expensive. It may still be used in calculating optimal deployments for smaller systems, or those whose characteristics are stable for a very long time. However, 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.

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 algorithms. Our 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.

In each step of the algorithm, we take a single component *aComp* and estimate the new deployment location (i.e., a host) for it such that the objective function *overallUtil* is maximized. Our heuristic 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:

$$svcImp(s) = \sum_{u \in U} \sum_{q \in Q} qUtil_{u,s,q}\left(\sigma_q\right)$$

Where σ_q is a configurable threshold in the algorithm and denotes a small rate of improvement in the QoS dimension q. Going in the decreasing order of service importance, the algorithm iteratively selects component *aComp* that participates in provisioning that service, and searches for the host *bestHost* for deployment. *bestHost* is a host $h \in H$ that maximizes $overallUtil(d, d_{aComp \to h})$, where d is the current deployment, and $d_{aComp \to h}$ is the new deployment if *aComp* were to be deployed on h.

If the *bestHost* for *aComp* satisfies all the constraints, the solution is modified by mapping *aComp* to *bestHost*. Otherwise, the algorithm leverages a heuristic that finds all "swappable" components *sComp* on *bestHost*, such that after swapping a given *sComp* with *aComp* (1) the constraints associated with H_{aComp} and *bestHost* are satisfied, and (2) the overall utility is increased. If more than one swappable component satisfies

the two conditions, we choose the component whose swapping results in the maximum utility gain. Finally, if no swappable components exist, the next best host (i.e., the host with the next highest value of *overallUtil* if *aComp* were to be deployed on it) is selected and the above process repeats.

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.

The complexity of this algorithm in the worst case with k iterations to converge is[8]

$$O(\#iteration \times \#services \times \#comps \times \#hosts \times overallUtil \times \#swappable\ comps \times overallUtil) =$$

$$O(k \times |S| \times |C| \times |H| \times (|S||U||Q|) \times |C| \times (|S||U||Q|) = O(|S|^3(|C||U||Q|)^2)$$

Since often only a small subset of components participate in providing a service and swappable components are only a small subset of components deployed on *bestHost*, the average complexity of this algorithm is typically much lower. Finally, just like the MIP algorithm, further complexity reduction is possible through specification of locational constraints, which reduces the number of times *overallUtil* is calculated.

The component swapping heuristic is important in that it significantly decreases the possibility of getting "stuck" in a bad local optimum. Further enhancements to the algorithm are possible at the cost of higher complexity. For example, simulated annealing [43] could be leveraged to explore several solutions and return the best one by conducting a series of additional iterations over our algorithm.

4. Genetic Algorithm

Another approximative solution we have developed is based on a class of stochastic approaches called genetic algorithms [43]. An aspect of a genetic algorithm that sets it apart from the previous three algorithms is that it can be extended to execute in parallel on multiple processors with negligible overhead. Furthermore, in contrast with the 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 a genetic algorithm significantly depend on its design (i.e., the representation of the problem and the heuristics used in promoting the good properties of individuals). In fact, the genetic algorithm we developed initially without deployment-specific heuristics significantly under-performed in comparison to the other algorithms. As a result, we had to devise a novel mechanism specifically tailored at our problem, discussed below.

---

[8] 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) with which they are associated.

In a genetic algorithm an individual represents a solution to the problem and consists 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 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 7a shows a 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 on which the component is deployed. 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 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 7a 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 7b shows a heuristic we have developed for representing an individual in response to this problem. 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 in the gene sequence of an individual to which a component is mapped. For example, component 2 is mapped to block 5 in the gene sequence (denoted by *i5*). Thus, block 5 of *Individual 1* in Figure 7b corresponds to block 2 of *Individual 1* in Figure 7a, and both denote 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 deemed most important. Sim-

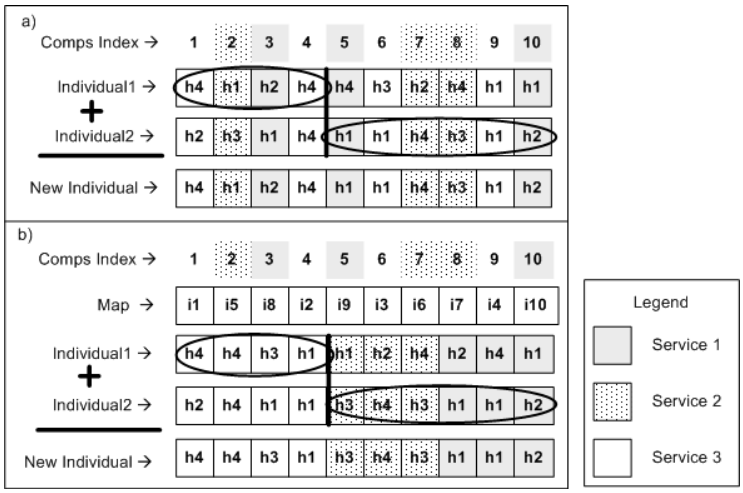


Figure 7. Application of the genetic algorithm to a problem comprising 10 components and 4 hosts: a) simple representation, b) representation based on services.

ilar to the heuristic used in our greedy algorithm, the most important service results in the highest utility gain for the smallest improvement in its QoS dimensions. We only allow cross-overs to occur on the borders of services. For example, in Figure 7b, we allow cross-overs at two locations: the line dividing blocks 4 and 5 and the line dividing blocks 7 and 8 of *Individuals 1* and *2*. As a result of the cross-over in Figure 7b, we have created an individual that has inherited the deployment of service 1 from *Individual 1* and the deployment of services 2 and 3 from *Individual 2*.

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, another simple heuristic we employ is to directly copy the individual with the highest fitness value (i.e., performing no cross-over or mutation) 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:

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

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

B. ADDITIONAL FACTORS IN (RE)DEPLOYMENT ANALYSIS

If the characteristics of an execution scenario change while the system is running, i.e., after the initial deployment is calculated and effected, the system is analyzed again by applying the most appropriate redeployment algorithm(s). Then, the improved deployment architecture is effected by redeploying the system's components as recommended by the algorithm(s). This process was depicted in Figure 1.

For illustration, Figure 8 shows the overall utility of a hypothetical distributed system to its users over a given time interval. Let us assume that the system's initial overall utility, U_1, is deemed unsatisfactory. This

can happen either because the system's runtime characteristics change (e.g., a network link fails, a new host enters the system) or because QoS dimensions and/or user preferences change. In either case, the system's operation is monitored and its model updated as appropriate during the interval T_M. The new model is then analyzed and an improved configuration may be determined during the period T_A. At that point, the system's redeployment is triggered if necessary. The triggering agent may be a human architect; alternatively, an automated deployment analyzer may initiate the task if pre-specified utility thresholds are reached. We provide such an automated deployment analyzer, further discussed in the remainder of this section.

Redeployment involves suspending some subset of the system's running components, relocating those components among the hosts, as well as possibly removing existing and/or introducing new components. During this time period, T_R, the system's overall utility will likely decrease as services become temporarily unavailable. The utility "dip" depends on the characteristics of the system (e.g., frequencies of interactions, number of dependencies) as well as the support provided by the implementation platform (e.g., rerouting of events, replicating components). Once the redeployment is completed, however, the overall utility increases to U_2 for the remainder of the time period T. If the system parameters change again, the system's utility may decrease, and this process may be repeated (illustrated in Figure 8 with period T').

The above process makes a critical assumption: The times required to monitor a system, update its model, analyze, and redeploy the system are small relative to the system's normal operation interval T_O (i.e., $T_M+T_A+T_R<<T_O$). If this is not the case, the system's parameters will be changing too frequently, triggering constant redeployments. This will result in the opposite of the intended effect since the system users would repeatedly experience the utility "dip". Therefore, in order to properly determine whether and how often to redeploy a system, our framework's deployment analyzer needs to know (1) the rate of change in system parameters, (2) the time required to apply the algorithms, (3) the time needed to actually redeploy the system, and (4) the overall utility gain of the calculated redeployment. We elaborate on these factors below.

For some systems, it may be possible to predict the frequency with which system parameters are likely to change (e.g., based on domain characteristics or similar past systems). In other cases, the deployment analyzer leverages the

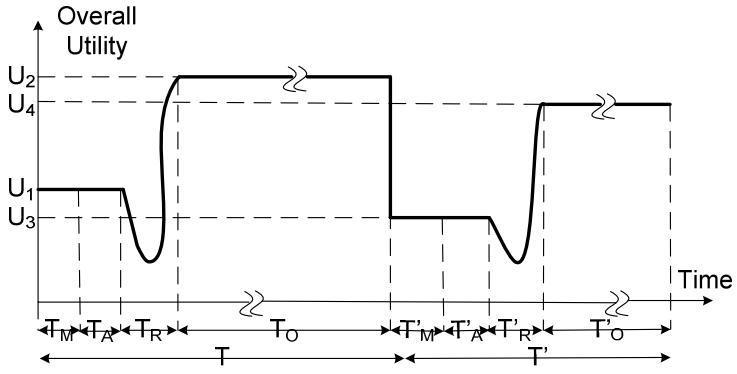


Figure 8. System's overall utility over time.

system's execution history to estimate the change rate in parameters; collecting this history is further discussed in Section VII. It is, of course, impossible to guarantee that the predicted behavior will be borne out in practice or that past behavior is representative of future behavior. In particular, our approach will not be effective in the case of highly unstable and unpredictable systems. At the same time, an argument can be made that other remedies need to be considered for such systems before determining an effective deployment becomes a priority.

While in theory one should always strive for the (re)deployment that maximizes the system's overall utility, in practice the time needed to apply the different redeployment solutions and the overall utility gain must be taken into account. Even though, with the exception of MIP, the algorithms we have devised run in polynomial time, they have very different characteristics that significantly affect their runtime performance as well as accuracy. Those characteristics are extensively evaluated in Section VII, allowing us to automatically select the most appropriate algorithm in a given situation.

Finally, deployment analysis must also take into account the amount of time required to actually redeploy the system and the intervening loss of system services. The time required to migrate a single component depends on the component's size as well as the reliability and bandwidth of the network link between the source and destination hosts. For example, if two hosts are connected by a 100 Kbps connection with 50% reliability, a 100 KB component would get migrated in 16s on average. The upper bound on the total time to effect a system's redeployment, T_R, for the scenario instantiated in Section III can be estimated as follows:

$$T_R = \sum_{r \in R} \left(\frac{cParam_r(compMem)}{nParam_{source_r,dest_r}(bw) \times nParam_{source_r,dest_r}(rel)} \right)$$

where R is the set of components to be redeployed; *source* and *dest* are the source and destination hosts, respectively; and *bw* and *rel* are the bandwidth and reliability of the network link, respectively. The above equation does not include network latency introduced by the underlying network protocol. Existing network latency estimation techniques (e.g., [11]) can be applied to estimate T_R more accurately. Additionally, the migrations on different hosts usually happen in parallel, thus significantly reducing the value of T_R. Finally, the above discussion assumes that either no additional mechanisms (e.g., rerouting of events, component replication) are provided to reduce service disruption during redeployment or, if they are provided, that they are captured by updating our system model.

The unavailability of system services during the T_R interval (depicted by the utility "dip" in Figure 8) may be a critical factor in deciding on the best deployment for a system in a given scenario. In some cases, users may be willing to accept temporary losses of service, so the algorithm suggesting a deployment with

the highest utility will be preferable. In other cases, however, losing certain critical services beyond a specific amount of time will be unacceptable to users, so redeployments that guarantee short downtimes for the components providing those services will be preferable (this may also result in simply retaining the current deployment). In such cases, the objective of applying a deployment improvement algorithm is to trade-off the utility gain achieved by a suggested new deployment (interval T_O in Figure 8) against the utility loss caused by the redeployment process (interval T_R). In order for our framework's deployment analyzer to be able to make this trade-off autonomously, it must know the system users' preferred utility thresholds. The thresholds can be obtained in a manner analogous to capturing the users' QoS preferences (recall Section II). If such thresholds do not exist, the output of a deployment improvement algorithm becomes a set of suggested deployments, with their resulting utility gains during T_O and losses during T_R. In those cases, a human is tasked with determining the best option. Finally, note that if any loss of a given service is unacceptable, this becomes a redeployment constraint in the system model (recall Section III).

C. DEPLOYMENT ANALYZER'S IMPLEMENTATION

We have implemented our framework's analysis capabilities discussed above by extending and tailoring DeSi [34], an Eclipse-based environment for deploying distributed systems.[9] DeSi leverages xADL, a highly extensible third-party ADL [7], to capture system models. The xADL schemas underlying DeSi simply had to be extended and modified so that it could support our deployment framework model. We then constructed a GUI front end (see Figure 9) and provided an API that allow, respectively, an engineer and another software component to manipulate DeSi's model. As a result, DeSi enables specification of arbitrary distributed software system models, including all the parameters and constraints defined in Section III. It is capable of exporting a data structure containing the system's suggested deployment architecture, which can be used by third-party tools to effect that architecture.

DeSi can randomly generate arbitrary distributed software system scenarios (see *Input* panel on the left side in Figure 9a). This feature can be used to explore system deployment utility variations and gains, by generating large numbers of deployment scenarios whose parameters fall within specified ranges. Additionally, this feature can be used to evaluate and compare the employed redeployment algorithms.

The next two sections outline the extensions to our architectural implementation and monitoring support, required to complete the (re)deployment framework depicted in Figure 1.

9 For ease of exposition, in this paper we will refer to the deployment analyzer's implementation on top of DeSi simply as "DeSi".

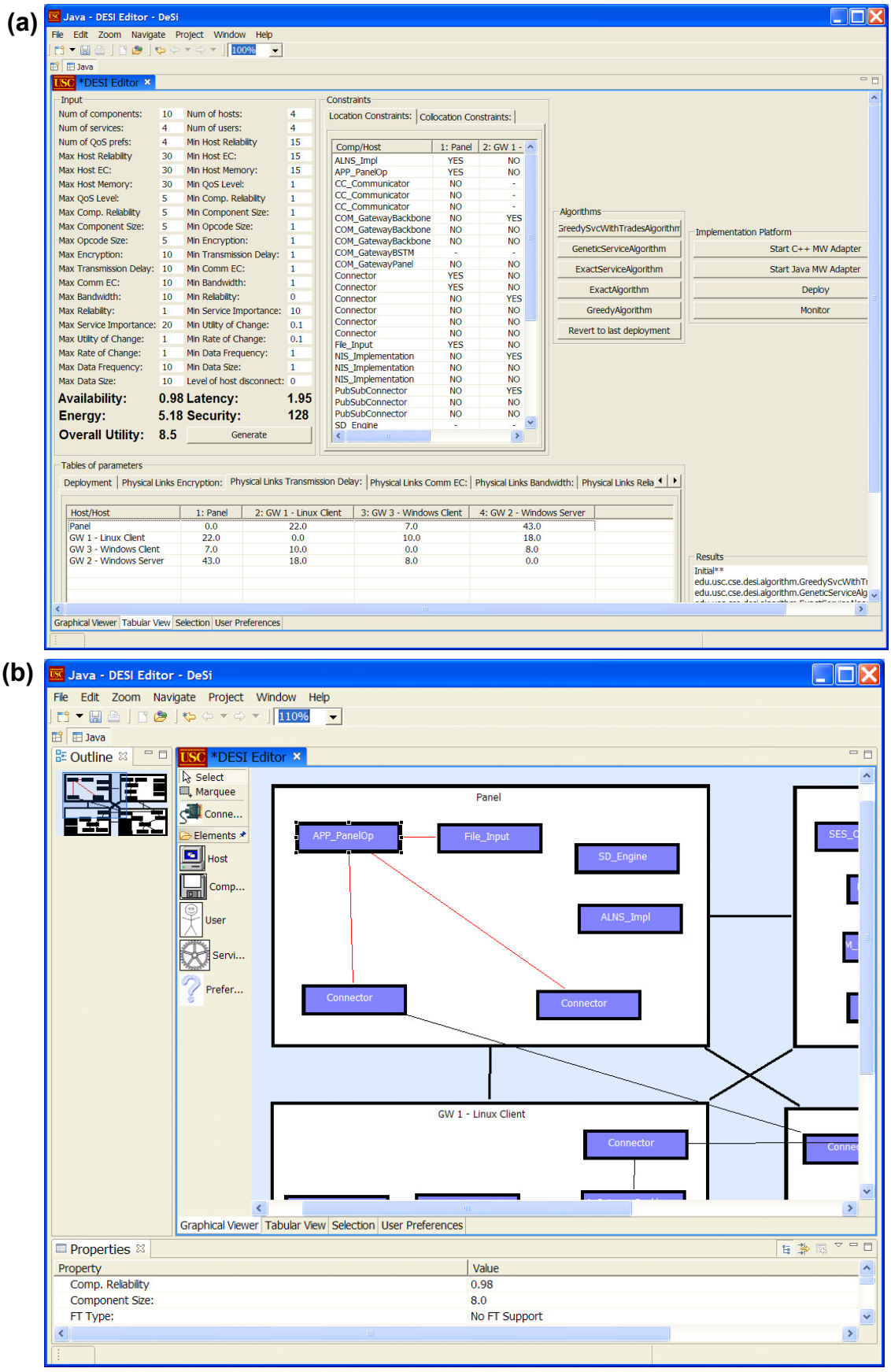


Figure 9. User interface of the deployment analyzer's implementation in DeSi: (a) the editable tabular view of a system's deployment architecture, and (b) the graphical view of the architecture.

V. DEPLOYMENT AND REDEPLOYMENT

Architecture modeling and analysis bound the scope of much of existing software architecture research [21, 23]. However, a system's characteristics established at the level of architecture are only relevant if we are able to transfer them consistently and reliably to the system's implementation. This is certainly the case with QoS-driven deployment and runtime redeployment. A major obstacle in maintaining the desired relationship between a software system's architecture and its implementation is that the two rely on very different abstractions and, in the general case, the correspondence between their respective constituent elements is many-to-many. For example, a given architectural component may be implemented via a set of classes, interfaces, and data files; conversely, an implementation-level library may realize a number of architectural components and connectors.

To deal with the problem of ensuring the reliable transfer of architectural properties into system implementations, we have previously proposed the concept of *architectural middleware* [25], a platform that natively supports architectural abstractions: components, connectors, configurations, ports, events, styles, and so on. We use such a platform in enabling QoS-driven (re)deployment. This platform is an extension of our Prism-MW middleware [25]. It allows us to maintain a 1-to-1 mapping between the architecture and implementation, which in turn simplifies (re)deploying the system as suggested by its architectural analysis.

Figure 10 shows the design of Prism-MW's extension used in this research. The shaded portion of the figure shows the "core" of Prism-MW, which is the minimum required for the implementation and execution of an architecture. The *Architecture* class contains the configuration of a given system's *Component*s and *Connector*s, and provides facilities for their addition, removal, and reconnection. A distributed system is implemented as a set of interacting *Architecture* objects, communicating via distribution-enabled *Port*s.

To support system (re)deployment, we have leveraged Prism-MW's *ExtensibleComponent* class, which contains a reference to *Architecture*. This allows an instance of *ExtensibleComponent* to access all architectural elements in its local configuration, acting as a meta-level component that can automatically effect changes to the configuration. In support of this work, we have introduced two instances of *ExtensibleComponent*: *Deployer* and *Admin*. The two collaborate to accomplish their task as follows (the reader can also reference Figure 15 for a graphical depiction in the context of an actual system):

1. *Deployer* receives a new deployment architecture from our framework's deployment analyzer. *Deployer* sends events to inform each *Admin* of the *Admin*'s new local configuration, and of the remote locations of components required for effecting changes to its configuration.

2. Each *Admin* determines the difference between its current and new configurations, and issues a series of Prism-MW *Event*s 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.
3. Each *Admin* that receives an event requesting its local component(s) to be remotely deployed detaches the required component(s) from its local configuration, serializes them, and sends them as a series of events via its distribution-enabled *Port* to the requesting device.
4. A recipient *Admin* reconstitutes the migrant components from the events and invokes the appropriate methods on its *Architecture* to attach the components to the local configuration.

The details of each of the steps performed by *Deployer* and *Admin*s (e.g., ensuring that a component is quiescent [19] before it is removed from the local configuration or that messages are properly routed to the redeployed components) are enabled by Prism-MW following the technique Oreizy et al. pioneered in [38,39]. We elide these details here as they are not necessary to understand the subsequent discussion.

Note that other coordination policies are possible by providing alternative implementations of the *Deployer* and *Admin* components. In fact, the coordination policy and the order in which components are deployed/started may be application-specific. For example, *Deployer* may initiate redeployment incrementally, on selected subsets of hosts, to reduce the number of unavailable system services at one time. Likewise, an *Admin* may execute the redeployment commands in a specific order (e.g., to minimize loss of events). While we have provided these incremental redeployment capabilities as proofs of concept, and our intuition is that in certain situations they will improve a system's overall utility, they are also likely to extend the interval T_R from Figure 8, which may be undesirable. Ultimately, the system engineers must decide what runtime redeployment policy is most appropriate for

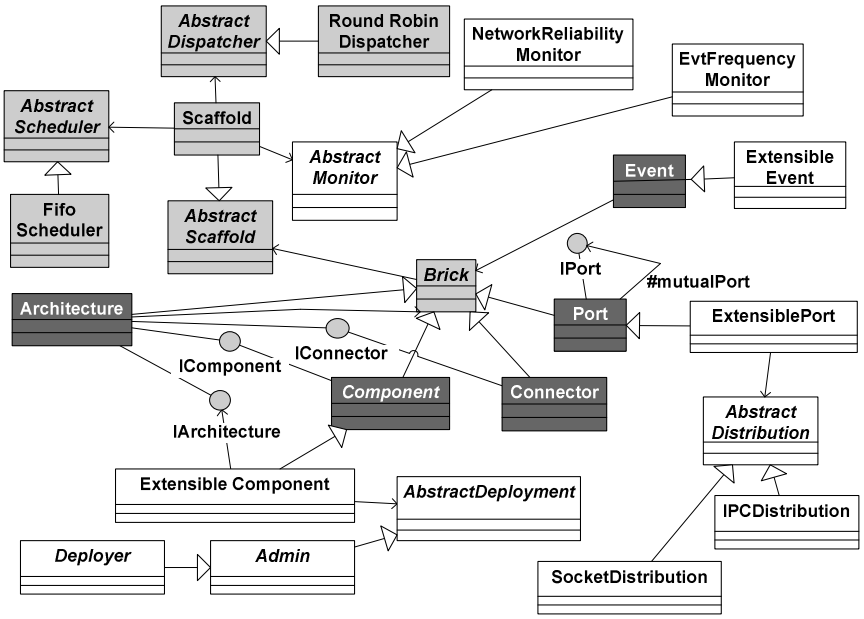


Figure 10. Elided UML class design view of Prism-MW. The unshaded classes are introduced in support of (re)deployment and monitoring.

their specific circumstances. Our framework allows the introduction of arbitrary new policies by making the desired modifications to the *Deployer* and *Admin* components.

VI. MONITORING

The final activity in our deployment improvement framework is continuous monitoring of the system to observe any changes in the relevant system parameters (recall Figure 1). As in the case of (re)deployment, we provide monitoring support by adapting Prism-MW [25]. We associate a *Scaffold* with every architectural element (see Figure 10). One of the *Scaffold*'s roles is to aid architectural self-awareness by probing the element's behavior, via different implementations of the *AbstractMonitor* class. Figure 10 shows two instances of this class: a *NetworkReliabilityMonitor* is attached to a distribution-enabled *Port* to assess the status of the given network link, while *EvtFrequencyMonitor* observes event traffic going through its *Port*. We have implemented several other monitors required to support the QoS dimensions outlined in Section III.B. Note that monitoring of some system parameters (e.g., power consumption of a component) may be challenging to implement. Our goal was not to provide ready-to-use monitors for each possible system parameter, but rather facilities that can enable the monitoring process. It is up to the engineer to leverage these facilities in implementing the most suitable monitor for the parameters of interest.

Each *Monitor* compares the collected data with a "base" value and reports any discrepancies to the local host's *Admin* component. In turn, the *Admin* acts as a monitoring *gauge* [10] that makes adjustments to its local architecture or notifies the system's *Deployer*. Modifications initiated by the *Admin* include changing the size of the Prism-MW event queue or thread pool on the local host. When non-localized changes to the system are required, the *Deployer* aggregates any variations in the system parameters received over a prespecified period of time, and makes requests of DeSi (recall Section IV.C) to update the system's model and analyze its utility. This reinitiates the entire process depicted in Figure 1 and described above.

Some properties of a system may be determined at design time, reducing the monitoring overhead, e.g., an upper bound on the memory required for executing a component may be determined either from its specification or by benchmarking it under the maximum envisaged workload. Using such upper bound values may render the analysis conservative, however, and the system's engineers must decide whether this is acceptable.

VII. EVALUATION

We have used our deployment improvement framework in a number of distributed system scenarios. While a majority of those were developed internally to observe and refine the framework, one was motivated by an external collaborator's real-world needs and another has resulted in the adoption of the framework and

its accompanying tools by an industrial organization. This demonstrates that the framework is effective in some situations, but it does not indicate the extent of its effectiveness. To do that, we have carefully evaluated different facets of the framework. We summarize that evaluation here. We first assess the framework's modeling and analysis capability along several dimensions. We then discuss the efficiency of the redeployment and monitoring facilities. We also outline our collaborative experience in applying our technique to two third-party families of distributed applications.

A. MODELING AND ANALYSIS

We have leveraged DeSi (recall Section IV.C) in our evaluation. DeSi's extensible modeling and visualization capabilities allowed us to configure it arbitrarily. The starting point of our evaluations was the instantiation of the framework with the system parameters and QoS dimensions discussed in Section III.B and shown in Figure 5. We then varied the parameters and extended the model with additional QoS dimensions. The largest scenarios we have worked with to date have involved on the order of dozens of users, QoS dimensions, and hardware hosts, and hundreds of software components and system services. The model from Section III has proven highly expressive in capturing the details of distributed system scenarios, as will be demonstrated in the remainder of this section.

$hostMem \in [10,30]$, $hostEnrCons \in [1,20]$,
$compMem \in [2,8]$, $opcodeSize \in [5,500]$,
$freq \in [1,10]$, $evtSize \in [10,100]$,
$bw \in [30,400]$, $enc \in [1,512]$, $rel \in [0,1]$,
$td \in [5,100]$, $commEnrCons \in [50,200]$
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
$MinUtil = 0.01$ *and* $MaxUtil = 1$

Figure 11. Input for DeSi's deployment scenario generation.

We instrumented DeSi to transform its internal model to GAMS [5], an algebraic optimization language. This allowed us to integrate DeSi with state-of-the-art MIP and MINLP solvers [6]. DeSi also exports an API for accessing its internal model, which we leveraged to implement the greedy and genetic algorithms. Finally, we leveraged DeSi's hypothetical deployment generation capability [34] to evaluate the algorithms on a large number of deployment scenarios. In the generation of deployment scenarios, some 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 automatically.

Figure 11 shows the input into DeSi for the generation of example scenarios and benchmarks. The values in Figure 11 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 our 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.).

Table 1. Results of an example scenario with 12C, 5H, 8S, and 8U. A positive number indicates improvement.

	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			

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 utilities (*qUtil*) for the QoS dimensions of each service. While users 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 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. Our evaluation focused on five different aspects of our analysis support, as detailed next.

1. Improving Conflicting QoS Dimensions

Table 1 shows the result of running our algorithms on an example application scenario generated for the input of Figure 11 (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 (randomly selected) 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 the QoS dimensions of each service. As discussed in Section IV, the MIP algorithm found the optimal deployment (with the objective value of 64 in this case).[10] The other algorithms found solutions that are within 20% of the optimal. We have observed similar trends across many other scenarios as discussed below.

10 An objective value (i.e., result of *overallUtil*) provides a scalar number indicating the quality of a deployment solution in comparison to other deployments within the same application scenario.

2. *Sensitivity to Users' Preferences*

Recall from Section III that the importance of a QoS dimension to a user is determined by the amount of utility specified for that dimension. QoS dimensions of services that have higher importance to the users typically show a greater degree of improvement. For example, in the scenario of Table 1, the users have placed a great degree of importance on service 4's availability and security. This is reflected in the results: for example, in MIP's solution, the 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 as important to the users and had to be degraded for improving other, more important QoS dimensions. As another illustration, a benchmark of 20 randomly generated application scenarios showed an average QoS improvement of 89% for the top one half of system services in terms of importance as rated by the users, and an average improvement of 34% for the remaining services.

3. *Performance and Accuracy*

Figure 12 shows a comparison of the four algorithms in terms of performance (execution time) and accuracy (value of the objective function *overallUtil*). 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 algorithms on them. The results correspond to the average values attained from these benchmarks. These results are reflective of a number of other tests. 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. 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. Our

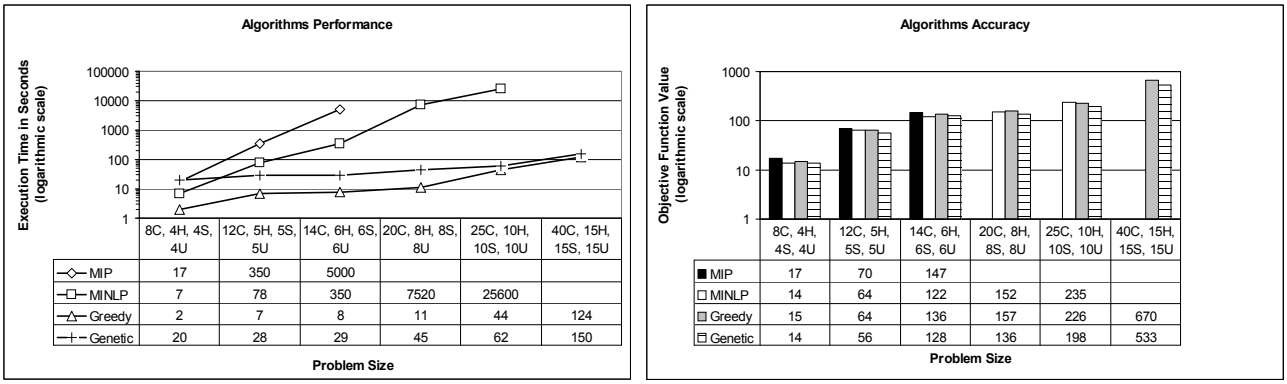


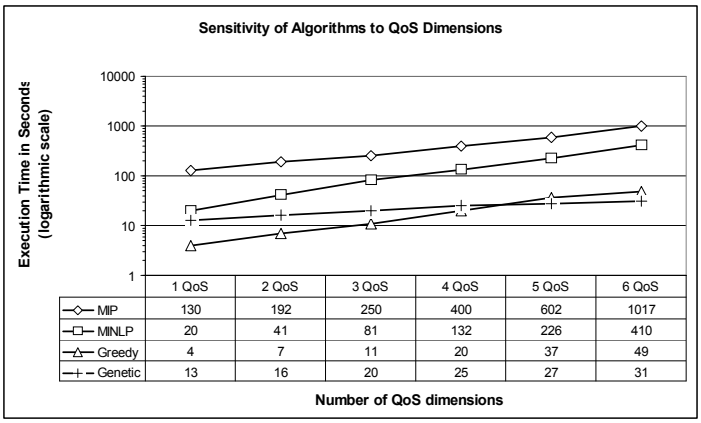
Figure 12. Comparison of the four algorithms' performance and accuracy.

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). For a meaningful comparison of the benchmark results, Figure 12 does not include problems that could not be solved by the MINLP solvers.

4. Sensitivity to QoS Dimensions

Figure 13 shows a representative result for the sensitivity of each algorithm's performance to the number of QoS dimensions. In this particular example, we analyzed an architecture of 12 components, 5 hosts, 5 services, and 5 users with varying numbers of QoS dimensions. In addition to the four QoS dimensions from Section III.B, we added arbitrary "dummy" QoS dimensions. Our framework's modeling support was able to capture those additional QoS dimensions easily.

As expected, the performance of all four algorithms is affected by the addition of new dimensions. However, the algorithms show different levels of sensitivity. The genetic algorithm shows the least degradation in performance. This is corroborated by our theoretical analysis: the complexity of the genetic algorithm increases linearly in the number of QoS dimensions, while the complexity of the greedy algorithm increases polynomially. While we do not have access to the proprietary algorithms of MIP/MINLP solvers, it is evident that their performance also depends significantly on the number of QoS dimensions.



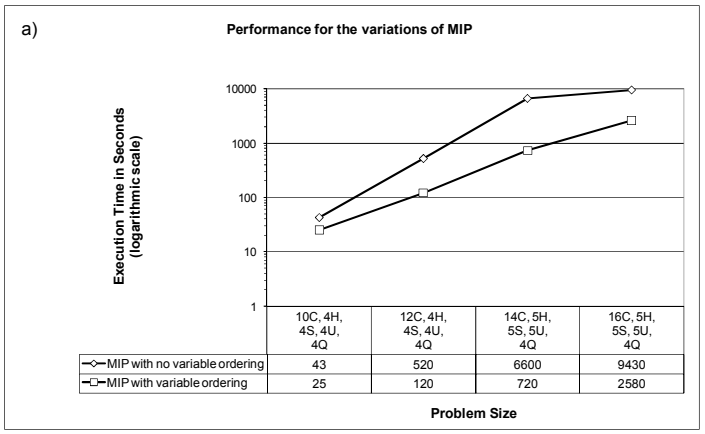
	1 QoS	2 QoS	3 QoS	4 QoS	5 QoS	6 QoS
MIP	130	192	250	400	602	1017
MINLP	20	41	81	132	226	410
Greedy	4	7	11	20	37	49
Genetic	13	16	20	25	27	31

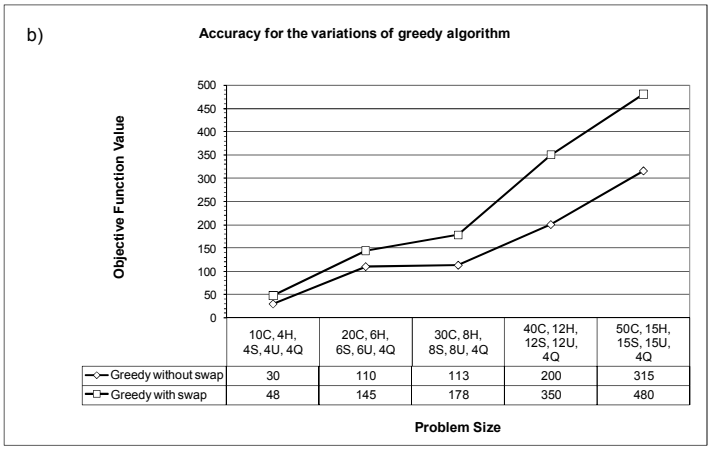
Figure 13. Sensitivity of performance to QoS dimensions.

5. Sensitivity to Heuristics

In Figure 14 we illustrate our evaluation of the heuristics we have introduced in the development of our algorithms (recall Section IV). Again, the results shown here are representative of other examples we have run. Figure 14a shows the effect of variable ordering on the performance of the MIP algorithm. As discussed in Section IV.A.2 and shown in the results of Figure 14a, 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 14b compares the greedy algorithm against a version that does not swap components when the parameter constraints on the *bestHost* are violated. As was discussed in Section IV.A.3, by swap-

ping components we decrease the possibility of getting "stuck" in a bad local optimum. The results of Figure 14b 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 14c compares three variations of the genetic algorithm. The first two variations were discussed in Section Section IV.A.4, where one uses the Map sequence to group components based on services 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 over the simple scenario where only one population of individuals was used.





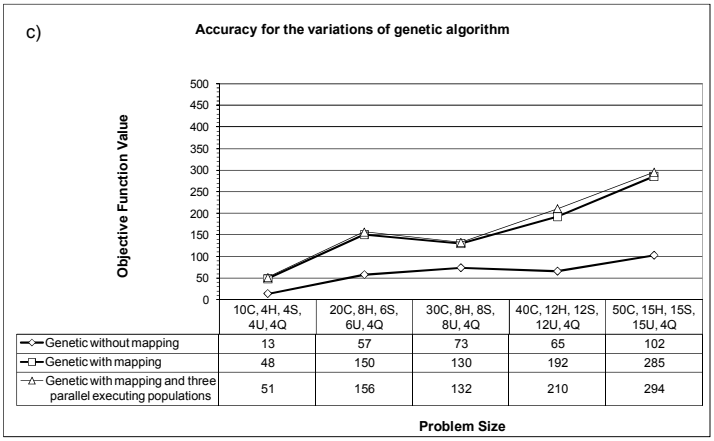


Figure 14. Results of the heuristics used by the algorithms.

6. Algorithm Selection

Since each of our algorithms has different strengths and weaknesses, two important questions are (1) when to use a given algorithm and (2) which algorithm is most appropriate in a given situation. We can provide some guidelines depending on the system's context. Unlike the results presented above, these guidelines

are qualitative, which is a reflection of the inexact nature of software architectural design. 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 comparatively thin and 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 users' 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 users' requirements and are often fixed throughout the system's execution (e.g., client components are often GUI components that do not need to be redeployed). 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 even medium-sized peer-to-peer systems, the MIP algorithm becomes infeasible and the software engineer has to leverage one of the three approximative algorithms to arrive at an improved deployment architecture.

In large application scenarios, the greedy and genetic approaches have an advantage over MINLP since they exhibit better performance and have a higher chance of finding a good solution. When the application scenario includes very restrictive constraints, greedy 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

its current form that allows for parallel execution on multiple decentralized hosts, thus distributing the processing burden of running the algorithm. We believe other types of parallel algorithms (e.g., variations of greedy) could be developed as well. We consider this to be an interesting avenue of future research.

The above discussion demonstrates that there is no one-size-fits-all solution for the deployment problem. Our intent with the development of our algorithms has not been to be exhaustive, but rather to demonstrate the feasibility of developing effective heuristics and algorithms that are generally applicable across different application scenarios. We expect that our framework will need to be augmented with additional algorithms. In fact, we are in the process of generalizing our auction-based algorithm for improving system availability [26] and integrating it into the framework to support (re)deployment in decentralized settings.

B. (RE)DEPLOYMENT AND MONITORING

Prism-MW is a light-weight middleware platform that is intended for use in mobile and embedded settings. Its efficiency and performance has been evaluated extensively in our previous work [25]. For example, memory usage of Prism-MW's core (recall Figure 10) has been recorded at 2.3 KB, while over ten million events with no application-specific data can be transferred using inter-process communication on an instance of the middleware running on an average PC in less than 3 seconds. Our measurements of the memory overhead for our framework's (re)deployment support showed that on average the Java implementation of the required Prism-MW "skeleton" configuration (*Admin* Component, *DistributionEnabledPort*, and Prism-MW's core) occupies around 14 KB. The *Admin* Component itself occupies 4 KB of memory.

Our empirical assessment of Prism-MW's monitoring support under a large number of scenarios suggests that monitoring on each host may induce as little as 0.1% and no greater than 10% in memory and computation overheads with respect to the running application. Prism-MW's monitoring introduces typically only two additional method calls: a call to the monitor, which, in turn, invokes a *put* method of its internal java.util.*Hashtable* class used to store the monitoring data. The memory overhead of the base *NetworkReliabilityMonitor* (*NRM_base*) is 0.21 KB, while the memory overhead of the base *EvtFrequencyMonitor* (*EFM_base*) is 0.22 KB. This overhead does not include the dynamic increase of the *Hashtable* class. Each addition to the *Hashtable* on the average adds 0.02 KB of memory. Of course, the total overhead of monitoring will depend on the number of software components and hardware hosts in the system.

On top of the memory overhead, the framework also introduces overhead on the network resources. There are three types of network overhead: the network bandwidth used for (1) the redeployment of software components, (2) the transfer of monitoring data to DeSi, and (3) the Prism-MW events exchanged for coor-

dinating the adaptation. Unlike the memory overhead, since the network overhead depends heavily on the application and its runtime characteristics (e.g., size of the components that are redeployed, the size of monitored data, and the level of fluctuation in system resources), this aspect of the framework can only be evaluated on a case-by-case basis. We provide estimates of the framework's overhead in the context of two software systems in the next section.

Additionally, since our monitoring support leverages Prism-MW's extensibility mechanism, the monitoring can be installed and removed from the running system at desired times. This enables us to eliminate the monitoring overhead in the cases when we know the rate of change in system parameters or when we want to prevent the system from frequently redeploying itself.

Finally, note that in highly unstable systems, the system parameters may fluctuate significantly, potentially requiring a large number of probes and a high sampling rate to obtain a reasonable estimate on the system resources, thereby increasing the overhead associated with monitoring. As mentioned in Section IV.C, redeployment may not be the most effective approach for improving the QoS of such systems, and they have not been the focus of our evaluation.

C. PRACTICAL APPLICATIONS OF THE FRAMEWORK

We have applied the described framework on two application families developed with external collaborators. The first application family, Emergency Deployment System (EDS) [22], is from the domain of mobile pervasive systems intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. Our work on EDS initially motivated this research. The second application family is MIDAS [28], a security monitoring distributed application composed of a large number of wirelessly connected sensors, gateways, hubs, and PDAs. In both cases, the applications involved varying numbers of hosts, components, system parameters, and QoS, allowing our collaborators to apply the framework. Below we describe our experience with applying our framework in the context of these two application families. We provide an overview of their functionalities and architecture to the extent necessary for explaining the role of the framework. An interested reader can find more details about these systems in [22,27,28].

1. MIDAS

Figure 15 shows a subset of MIDAS's software architecture. MIDAS is composed of a large number of wirelessly connected sensors, gateways, hubs, and PDAs. 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

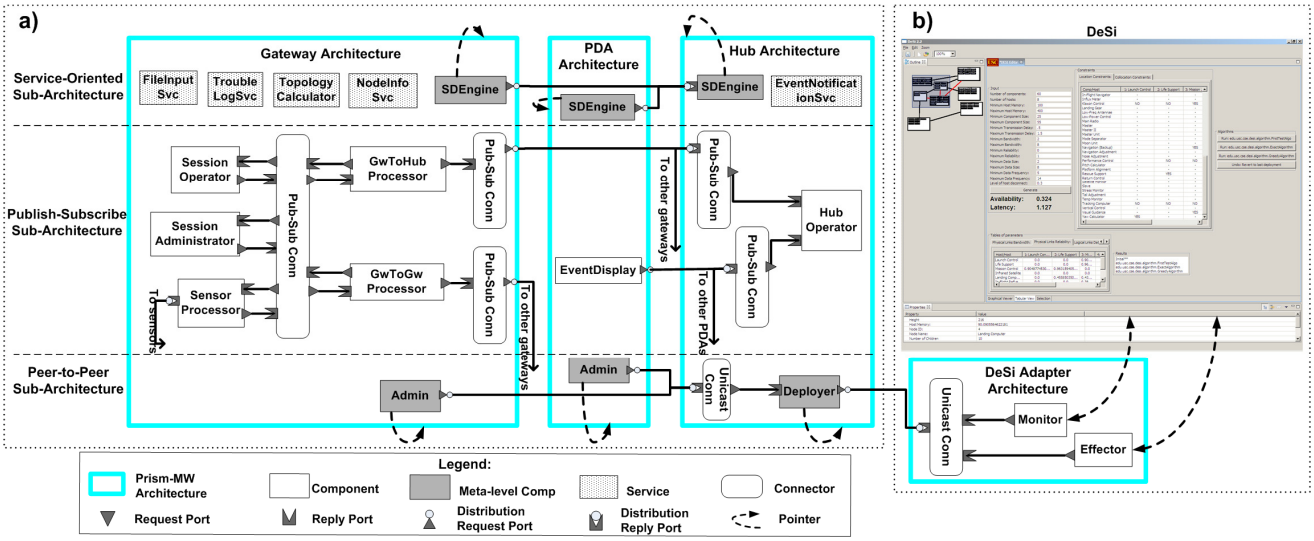


Figure 15. An abridged view of MIDAS's architecture that is monitored, analyzed, and adapted at runtime: a) portions of MIDAS's software architecture, including its three sub-architectures; b) DeSi and its Prism-MW Adapter.

fuse the data received from the sensors, and propagate the appropriate data (e.g., events) 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 a 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 used by the mobile users of the system. As denoted on the left side of the figure, three architectural styles were used in MIDAS: service-oriented, publish-subscribe, and peer-to-peer. The peer-to-peer portion of this architecture corresponds to the meta-level functionality of system monitoring, analysis, and adaptation via DeSi and Prism-MW.

MIDAS has several QoS requirements that need to be satisfied in tandem. The two most stringent requirements are latency and energy consumption: MIDAS is required to transmit a high-priority event from a sensor to a hub and to receive an acknowledgement back in less than two seconds; given that some of the MIDAS platforms (e.g., sensors and some gateways) are battery-powered, minimizing the energy consumption is also of utmost concern. MIDAS also has several deployment constraints that need to be satisfied. Some examples of these constraints are as follows: exactly one instance of the *SDEngine* component, which is responsible for the discovery of currently available services in MIDAS, should be deployed on each host; every *SessionOperator* component should be collocated with the corresponding *SessionAdministrator* component; a *HubOperator* component can only be deployed on a hub; and so on. On top of the locational constraints, MIDAS has several resource (system) constraints. Most notably, the sensors, PDAs, and some of the gateway platforms are memory-constrained devices that could only host a small number of components.

Our objective was to find and maintain an effective deployment for different instantiations of MIDAS. Prior to this, MIDAS's deployments were determined manually, at times guided by dubious rationales, and

their effectiveness was never evaluated quantitatively. This is of particular concern in a mobile embedded system, which is affected by unpredictable movement of target hosts and fluctuations in the quality of wireless network links. Moreover, since engineers did not know a priori the properties of the target environment and the system's execution context, they would often make deployment decisions that were inappropriate. In turn, given that the majority of platforms in MIDAS lack a convenient interface (e.g., monitor, disk drive, or keyboard) that could be used for the download and installation of software, redeploying the software was a cumbersome task that required bringing MIDAS down each time.

One set of application scenarios in which our framework was applied was similar to that shown in Figure 15 but with two additional gateways and a total of 30 software components. We could not execute the MIP algorithm due to size of the problem (i.e., 5^{30} combinations). Instead, we executed the genetic, greedy, and MINLP algorithms, and selected the best solutions. For these particular scenarios, the genetic algorithm outperformed greedy and MINLP. It took the genetic algorithm slightly over 40 seconds on a mid-range PC to find the solutions every time, in comparison to the average of 4.5 hours required for an engineer to manually find a deployment for the same system that satisfied only the system's constraints (i.e., without even attempting to optimize the latency and energy consumption QoS dimensions). It is then not surprising that the genetic algorithm's solutions were on the average 23% better than the manual deployments in terms of QoS provisioned by the system's services.

After a design-time analysis of the model, our framework was leveraged to deploy and execute each instance of the system. To this end, the *Effector* component in the DeSi *Adapter Architecture* converted the deployment model exported by DeSi into a total of 165 commands, which were sent to the *Deployer* and *Admin* components running on MIDAS's hosts. Each command corresponds to a runtime adaptation (e.g., add component, attach communication port) that is performed by Prism-MW. The total time for automatically deploying the software, which included shipping the component's logic via DLL files, instantiating and initializing the component, and configuring the architecture, was measured to be just under 11 seconds on the average. In comparison, it took the MIDAS engineers 6 hours on the average to deploy and configure the same software system using common build scripts (e.g., Apache Ant).

The system was then monitored, and the framework's ability to react to changes in the environment was tested repeatedly during the system's execution. The framework proved useful in improving the system's QoS in several instances. In particular, our collaborators found the framework effective in mitigating two types of situations. When the movement of a PDA caused the available wireless network bandwidth to drop off significantly, the framework would selectively redeploy some of the services in the service-oriented layer

of the gateways (e.g., *NodeInfoSvc*) to the PDA, which resulted in close to a 30% improvement in the application's response time on the average. Similarly, in situations where unexpected load on the system depleted some gateways' or PDAs' batteries, the framework redeployed the computationally intensive components (e.g., *TopologyCalculator*) to the back-end hubs, and prolonged the life of the depleted hosts. Our measurements showed that these particular redeployments reduced the energy consumption of the depleted hardware hosts by 40% or more, although they also resulted in increased latencies. While it is conceivable that an engineer may have been able to effect the same redeployments manually, our framework allowed her to rapidly explore a number of options, set the appropriate parameter priorities, and obtain accurate quantitative results.

The analysis of the MIDAS architecture in approximately 100 different scenarios resulted in the redeployment of three software components on average. That is, only a small fraction of the system's 30 components were redeployed at any point in time. This is attributed to the fact that the changes were localized to a small portion of the system at any point in time (e.g., heavy load on a host, weak network signal due to movement of a particular PDA). Moreover, since the framework typically redeployed only small portions of the system at a given time, the system's users were often unaware that the changes were occurring.

The average size of the MIDAS components redeployed in our experiments was 113KB. Given that on average three components were redeployed per scenario, we estimate the network traffic associated with redeployment to be 340KB. On top of this, the size overhead induced by Prism-MW events carrying the monitored data ranged from 0.05KB to 0.2KB, while the size of Prism-MW events carrying the adaptation commands ranged from 0.05KB to 0.12KB. The framework's network usage was negligible in MIDAS, as the devices were connected using a dedicated IEEE 802.11b wireless network, which provides up to 11Mbps of bandwidth. However, the network impact of the approach would have to be considered more carefully in systems with a slower network.

Finally, an unexpected outcome of our experience was that the engineers found the framework to be helpful not only for improving QoS, but also for rapidly testing a fix to a problem during the development. This is a highly time-consuming task that is traditionally performed manually in this setting, which our framework's redeployment and monitoring facilities were able to streamline.

2. Emergency Deployment System

We now describe our experience with our deployment framework in the context of the Emergency Deployment System (EDS) application family, which was foreshadowed in the scenario used in the Introduction. An instance of EDS with single *Headquarters*, four *Team Lead*s, and 36 *Responder*s is shown in Figure 16a.

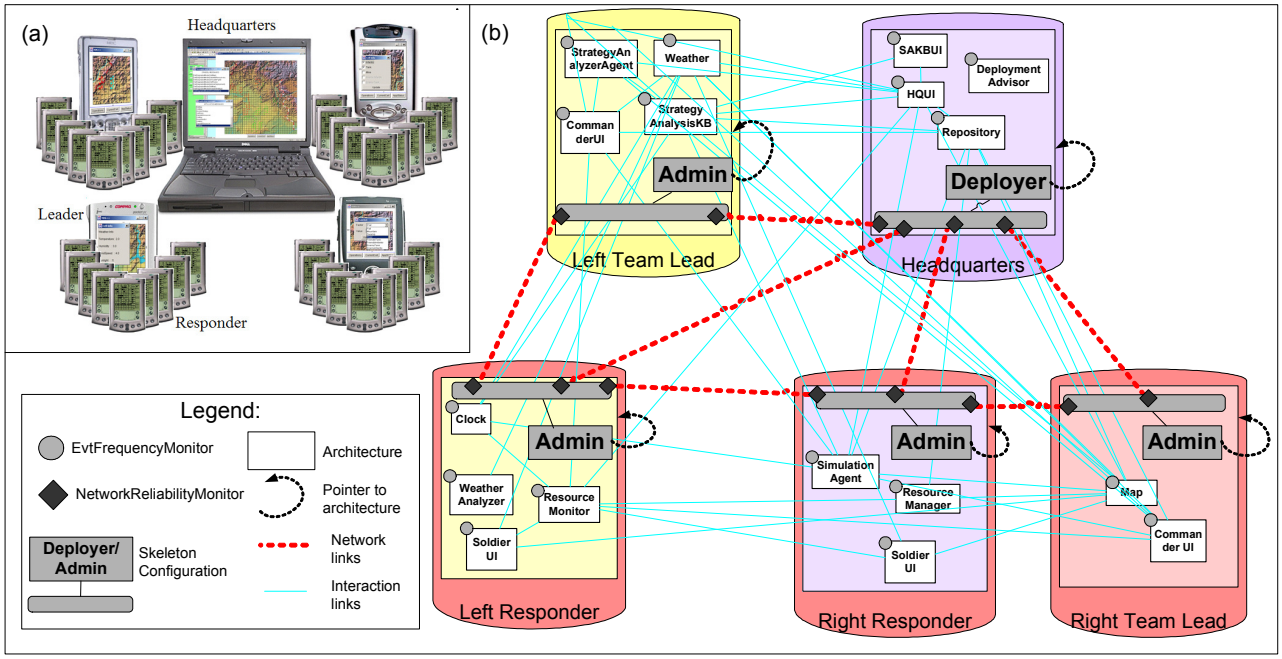


Figure 16. (a) An instance of EDS. (b) A subset of the EDS deployment architecture that is monitored and redeployed using our framework.

A computer at *Headquarters* gathers information from the field and displays the current field status: the locations of friendly and, if applicable, enemy troops, vehicles, and obstacles such as fires or mine fields. The headquarters computer is networked via secure links to a set of PDAs used by *Team Lead*s in the field. The team lead PDAs are connected directly to each other and to a large number of *Responder* PDAs. Each team lead is capable of controlling his own part of the field: deploying responders, analyzing the deployment strategy, transferring responders between team leads, and so on. In case the *Headquarters* device fails, a designated *Team Lead* assumes the role of *Headquarters*. *Responder*s can only view the segment of the field in which they are located, receive direct orders from the *Team Lead*s, and report their status.

Figure 16b shows the deployment architecture of an instance of EDS while it is being monitored and deployed via Prism-MW's meta-level components. The corresponding models were constructed in DeSi and populated with the information available at design time (e.g., upper-bound estimates on the sizes of components, locational constraints, available memory on the hosts) as well as the data monitored at runtime (e.g., frequency of invocations, network reliabilities).

Figure 17 shows a portion of the models constructed in DeSi to represent the system's users (left), their QoS preferences (middle), and user-level services (right) provisioned by the system's components for the deployment architecture from Figure 16b. In Figure 17, *Latency* is selected as the QoS dimension of interest. As a result, DeSi is showing the *Headquarters*' latency preferences for the *Analyze Strategy* service in the

property sheet (bottom). This figure shows one way of specifying preferences in DeSi: a 10% (0.1) improvement in latency results in a 20% (0.2) increase in utility for the *Headquarters* user.

The EDS scenario reported on here and depicted in Figure 16b consisted of 5 users: *Headquarters*, *Left Team Lead*, *Right Team Lead*, *Left Responder*, and *Right Responder*. Altogether, the users specified 10 different QoS preferences for a total of 8 services. The services were *Analyze Strategy*, *Move Resources*, *Get Weather*, *Get Map*, *Update Resources*, *Remove Resources*, *Simulate Drill*, and *Advise Deployment*. For the sake of clarity, we have chosen not to detail the larger instances of EDS (comprising up to 105 software components and deployed on up to 41 hosts) on which the framework has been applied.

Table 2 shows the preferences of the five users, captured in this case as linear functions. The table shows that *Headquarters* has specified a utility that increases twice as fast as the rate of change in latency of the *Analyze Strategy* service. Other forms of representing the users' preferences, such as sigmoid functions proposed in [46], are also possible. In EDS, we found simple linear representation of the users' preference to be sufficiently expressive. Note that users are not required to express their preferences in terms of functions. For example, they may express their preferences in terms of simple scalar values, which are then passed through commonly available regression tools to derive equivalent utility equations.

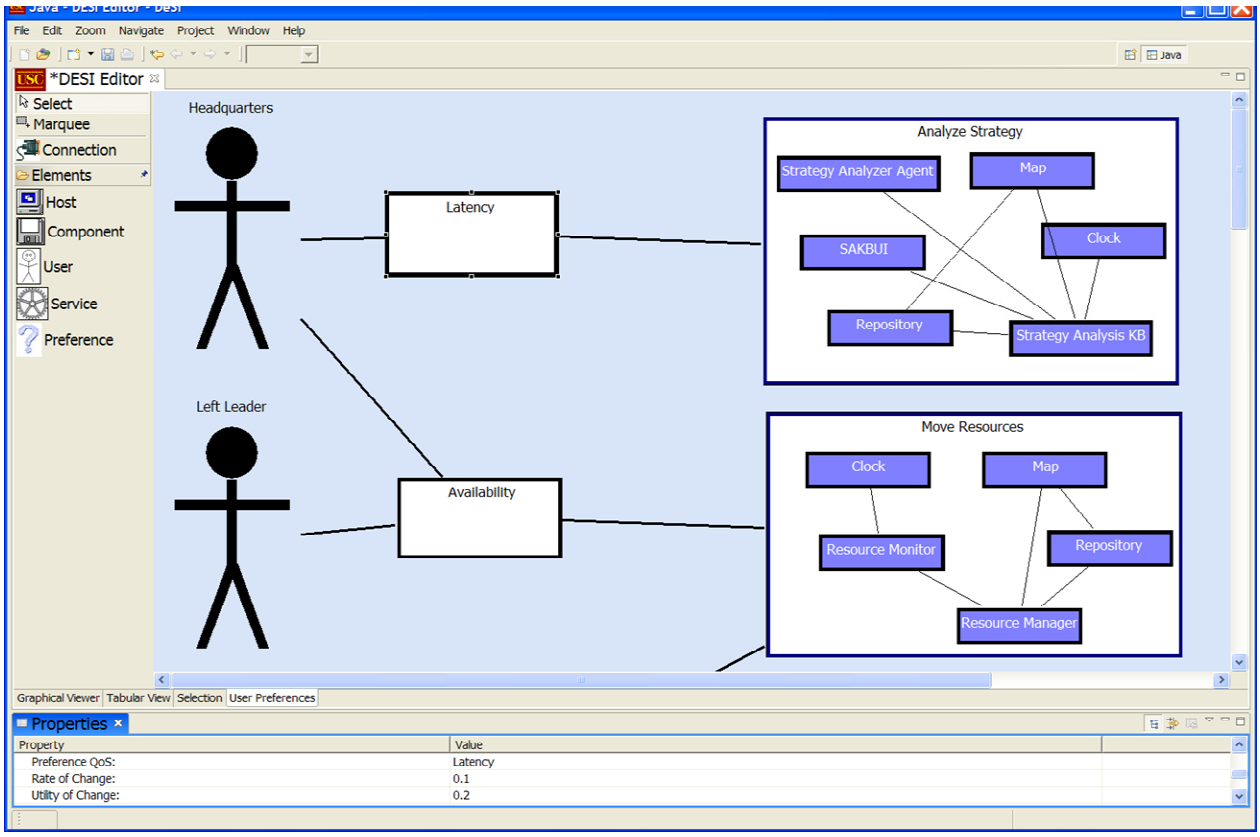


Figure 17. Some of the EDS users, their QoS preferences, and user-level services as modeled in DeSi.

Table 2. Users' preferences in EDS, where the rate of change in availability, energy consumption, latency, and security are represented by *a*, *e*, *l*, and *s*, respectively.

	Headquarters	Left Team Lead	Right Team Lead	Left Responder	Right Responder
Analyze Strategy	*qUtil(l) = 2l*		*qUtil(e) = 2e*		
Move Resources	*qUtil(a) = 4a*	*qUtil(a) = 4a*	*qUtil(l) = (4/3)l*		
Get Weather				*qUtil(s) = (1/9)s* *qUtil(e) = 4e*	*qUtil(e) = 4e*
Get Map	*qUtil(l) = (4/3)l*	*qUtil(s) = (1/9)s*	*qUtil(l) = (4/3)l*	*qUtil(e) = 4e*	*qUtil(e) = 4e*
Update Resources			*qUtil(l) = 2l*		*qUtil(l) = 2l*
Remove Resources				*qUtil(e) = 4e*	*qUtil(e) = 4e*
Simulate Drill					
Advise Deployment		*qUtil(l) = 2l*	*qUtil(l) = 2l*		*qUtil(e) = 2e*

Unlike the "synthesized" problems presented in Section VII.A, where we evaluated the algorithms under the most stringent conditions (i.e., all users specify preferences for all of the QoS dimensions of all services), in this scenario the users did not specify preferences for some of the QoS dimensions of certain services.

The greedy algorithm was the best approach for solving this problem for two reasons (recall Section VII.A.6): (1) the architecture was large enough (i.e., $5^{17} \approx 760$ billion possible deployments) that executing the MIP algorithm required approximately 3 hours and MINLP approximately 8 minutes; and (2) there were many locational constraints, which tends to hamper the accuracy of the genetic algorithm. The greedy algorithm executed in 7.6 seconds and was able to produce a solution that was within 4% of the optimal solution found by MIP.

Table 3 shows the results of executing the greedy algorithm on this instance of the EDS application. Careful analysis of this table allows us to relate the characteristics of the solution to the preferences specified by the users, and hence develop some insights into the quality of the solution produced by the algorithm:

- On the average the four QoS dimensions of the eight services improved by 42%. This indicates the extent to which a good deployment can impact a system's overall QoS.
- The QoS dimension of services for which the users have specified a preference improved on the average by 78%, while the remaining QoS dimensions improved on the average by 17%. This corroborates that the greedy algorithm indeed zeroes in on the QoS dimensions that are important to the users.
- The average improvement in the QoS dimensions of the *Simulate Drill* service is 4%, which is significantly lower than the average improvement of other services. This is attributed to the fact that no QoS preferences were specified for *Simulate Drill* (recall Table 2). In contrast, the average improvement in the delivered quality of the services for which multiple users specified preferences (e.g., *Move Resources*, *Get Map*, and *Advise Deployment*) is significantly higher than for the other services.

- The average improvements for *Energy Consumption* and *Latency* are significantly higher than for the other two QoS dimensions. This is attributed to the fact that users specified more preferences for these two dimensions.
- Notice that in a few cases the QoS dimensions have degraded slightly, reflecting the fact that the users have not specified any preferences for them.

Table 3. Results of running the greedy algorithm on the EDS scenario. The highlighted cells correspond to the QoS dimensions for which the users have specified preferences in Table 2.

	Availability	Latency	Communication Security	Energy Consumption	Avg. Improvement in Each Service
Analyze Strategy	63%	**12%**	5%	**79%**	39.7%
Move Resources	**78%**	**68%**	2%	63%	52.7%
Get Weather	-3%	11%	**81%**	**59%**	37%
Get Map	29%	**59%**	**82%**	**73%**	60.7%
Update Resources	9%	**92%**	19%	-6%	28.5%
Remove Resources	-1%	25%	23%	**103%**	37.5%
Simulate Drill	17%	-8%	1%	-6%	4%
Advise Deployment	61%	**134%**	22%	**92%**	77.2%
Avg. Improvement in Each QoS	31.6%	48.1%	29.3%	57.1%	

The results obtained in this instance are on par with those obtained from other examples (e.g., recall Table 1) as well as in other instances of EDS. The largest instance of EDS on which we applied the framework consisted of 105 software components and 41 hardware hosts. While in this instance it was not possible to execute the MIP and MINLP solvers, the genetic and greedy algorithms were able to find solutions in approximately 4 and 5 minutes, respectively. A detailed analysis of the solution found by these algorithms indicated the same level of fine-grained trade-off analysis based on the users' preferences.

VIII. RELATED WORK

Numerous researchers have looked at the problem of improving a system's QoS through resource scheduling [17] and resource allocation [20,36]. However, only a few have considered the users' preferences in improving QoS. The most notable of these approaches are Q-RAM [20] and the work by Poladian et al. [42]. Q-RAM is a resource reservation and admission control system that maximizes the utility of a multimedia server based on the preferences of simultaneously connected clients. Poladian et al. have extended Q-RAM by considering the problem of selecting applications among alternatives such that the cost of change to the user is minimized. Neither of these works considers the impact of the software system's deployment architecture on the provided QoS. Furthermore, these approaches are only applicable to resource-aware applications, i.e., applications that can be directly customized based on the available resources.

Carzaniga et al. [4] provide an extensive comparison of existing software deployment approaches. They identify several issues lacking in the existing deployment tools, including integrated support for the entire deployment life cycle. An exception is Software Dock [6], which provides software agents that travel among hosts to perform software deployment tasks. Unlike our approach, however, Software Dock does not focus on extracting system parameters, visualizing, or evaluating a system's deployment architecture, but rather on the practical concerns of effecting a deployment.

The problem of improving a system's deployment has been studied by several researchers. I5 [1], proposes the use of the integer programming model for generating an optimal deployment of a software application over a given network, such that the overall remote communication is minimized. Solving their model is exponentially complex, rendering I5 applicable only to systems with very small numbers of software components and hosts. Coign [15] provides a framework for distributed partitioning of a COM application across only a pair of hosts on a 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. Kichkaylo et al. [18] provide a model, called component placement problem, for describing a distributed system in terms of network and application properties and constraints, and a planning algorithm for solving the CPP model. The focus of this work is to capture the constraints that restrict the solution space of valid deployment architectures and search for any valid deployment that satisfies those constraints, without considering the deployment's quality. Manolios et al. [30] propose a language for expressing the requirements of component assembly, including a set of properties and constraints that need to be satisfied. Requirements are compiled into a Boolean Satisfiability Problem (SAT) and solved using commonly available solvers. This approach checks only whether a given component assembly is legal, and does not take into consideration users, user preferences, or optimization of multiple objectives. Bennani and Menasce [3,31] have developed a technique for finding the optimal allocation of application environments to servers in data centers. Similar to our approach, their objective is to optimize a utility function, which is used to perform trade-off analysis on the different facets of performance. Unlike our approach, however, their utility functions are not driven by the users' preferences. Moreover, the models and algorithms presented are limited to predefined system and QoS properties that are deemed important in the data center domain. Finally, in our prior work [26,32], we devised a set of algorithms for improving a software system's availability by finding an improved deployment. The novelty of our approach was a set of algorithms that scaled well to large systems. 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, or attempts to improve more than one QoS dimension of interest. The only exception is our earlier work [35], in which we highlighted the need for user-driven multi-dimensional optimization of software deployment and briefly outlined a subset of the framework's components. Furthermore, 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, our approach is unique in that it provides a mathematical framework for quantifying the utility of an arbitrary system's deployment, taking into account a set of system parameters of interest. The framework thereby provides an objective scale for assessing and comparing different deployments of a given system.

The implementation and evaluation of the redeployment techniques mentioned above is done in an ad-hoc way, making it hard to adopt and reuse their results. One of the motivations for developing our framework and its accompanying tool support has been to address this shortcoming. Related to our work is the research on architecture-based adaptation frameworks, examples of which are the frameworks by Garlan et al. [10] and Oreizy et al. [38]. Similar to them, in [29], we present the application of a subset of our framework's components in enabling architecture-based adaptation of a mobile robotic system; we do not summarize those results in this paper's evaluation because the primary focus there is on addressing challenges posed by mobility rather than deployment. As opposed to general purpose architecture-based adaptation frameworks, we are only considering a specific kind of adaptation (i.e., redeployment of components). Therefore, we are able to create a more detailed and practical framework that guides the developers in the design of redeployment solutions. Related also is previous research on adaptation assurance and verification techniques [49,52], which we view as complementary to our approach for ensuring safe and sound reconfiguration of software components. Finally, optimizing allocation of software (logical) elements to hardware (physical) resources is an area of research that has been studied in a variety of contexts before, such as distributed process scheduling [12], task scheduling in grid computing [47], and process allocation to clustered computers [14]. These works have guided the development of our framework. However, unlike any of these works, our framework is targeting the allocation of software components, which are not only conceptually very different from OS-level processes or grid-level tasks, but are also realized and treated differently in practice. Moreover, almost none of these approaches are driven by complex and conflicting user QoS requirements, but rather focus on improving a particular system-level metric (e.g., maximizing CPU utilization/throughput) in isolation.

IX. CONCLUSION

As the distribution and mobility of computing environments grow, so does the impact of a system's deployment architecture on its QoS properties. While several previous works have studied the problem of assessing and improving the quality of deployment in a particular scenario or class of scenarios, none have addressed it in its most general form, which may include multiple, possibly conflicting QoS dimensions, many users with possibly conflicting QoS preferences, many services, and so forth. Furthermore, no previous work has developed a comprehensive solution to the problem of effectively managing the quality of a system's deployment. In this paper, we have presented an extensible framework for improving a software-intensive system's QoS by finding the best deployment of the system's software components onto its hardware hosts. The framework allows rapid, quantitative exploration of a system's, typically very large, deployment space.

From a theoretical perspective, the contribution of our approach is a QoS trade-off model and accompanying algorithms, which, given the users' preferences for the desired levels of QoS, find the most suitable deployment architecture. We have demonstrated the tailorability of our solution and its ability to handle trade-offs among QoS dimensions by instantiating it with four representative, conflicting dimensions. We also discussed four approaches to solving the resulting multi-dimensional optimization problem, and presented several novel heuristics for improving the performance of each approach. The design of the framework model and algorithms allows for arbitrary specification of new QoS dimensions and their improvement.

From a practical perspective, the contribution of our work is an integrated solution, in which the data about system parameters are either acquired at design-time (via an ADL or from a system architect) or at runtime (via Prism-MW), and an improved deployment architecture is calculated (via DeSi), and effected (via an interplay between Prism-MW and DeSi). The framework provides the foundation for comparing these and other solutions and for conducting future research into new distribution scenarios and new algorithms.

While our results have been very positive, a number of pertinent questions remain unexplored. We intend to extend the model to allow for the expression of negative utility due to the inconvenience of changing a system's deployment at runtime. This will make the approach more practical for use in highly unstable systems, where continuous fluctuations may force constant redeployments. We are also developing the capability to automatically select the best algorithm(s) based on system characteristics and execution profile. Since redeployment is only one approach for improving QoS of distributed software systems, we plan to extend the framework to other types of adaptation choices that may impact a system's QoS, and perform the analysis not only among the alternative deployments, but across a larger suite of adaptation choices [38,39,50].

X. REFERENCES

1. Architecture Analysis and Design Language (AADL). http://www.aadl.info/
2. M. C. Bastarrica, A. A. Shvartsman, and S. A. Demurjian, "A Binary Integer Programming Model for Optimal Object Distribution," *Int'l Conf. on Principles of Distributed Systems*, Amiens, France, Dec. 1998.
3. M. Bennani, and D. Menasce, "Resource Allocation for Autonomic Data Centers Using Analytic Performance Models," *Proc. IEEE International Conference on Autonomic Computing*, Seattle, WA, June 13-16, 2005.
4. A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. van der Hoek, and A. L. Wolf, "A Characterization Framework for Software Deployment Technologies," *Technical Report*, Department of Computer Science, University of Colorado, 1998.
5. E. Castillo, et al. "Building and Solving Mathematical Programming Models in Engineering and Science," *John Wiley & Sons*, New York, NY, 2001.
6. J. Czyzyk, M. P. Mesnier, and J. J. More, "The NEOS Server," *IEEE Journal of Computational Science and Engineering*, pages 68-75, 1998.
7. E. M. Dashofy, A. Van der Hoek, and R. N. Taylor, "A Comprehensive Approach for the Development of Modular Software Architecture Description Languages," *ACM Transactions on Software Engineering and Methodology*, 14(2), April 2005, pp. 199-245.
8. G. Edwards, S. Malek, and N. Medvidovic. "Scenario-Driven Dynamic Analysis of Distributed Architecture," *Int'l. Conf.on Fundamental Approaches to Software Engineering (FASE 2007)*, Braga, Portugal, March 2007.
9. G. Edwards and N. Medvidovic, "A Methodology and Framework for Creating Domain-Specific Development Infrastructures," *Int'l Conf. on Automated Software Engineering (ASE)*, L'Aquila, Italy, September 2008.
10. D. Garlan, S. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-based Self-repair," *In R. de Lemos, C. Gacek, A. Romanovsky, eds., Architecting Dependable Systems*, 2003.
11. K. Gummadi, et al. "King: Estimating Latency between Arbitrary Internet End Hosts," *SIGCOMM Computer Communication Review*, July 2002.
12. T. Gyires, "A Distributed Process Scheduling Algorithm Based on Statistical Heuristic Search," *IEEE Int'l Conf. on Systems, Man and Cybernetics: intelligent systems for the 21st century*, New Jersey, USA, 1995.
13. R. S. Hall, D. Heimbigner, and A. L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," *Int'l Conf. in Software Engineering (ICSE 1999)*, Los Angeles, CA, May 1999.
14. K. E. Hoganson, "Workload Execution Strategies and Parallel Speedup on Clustered Computers," *IEEE Transactions on Computers*, vol. 48, no. 11, pages 1173-1182, Nov. 1999.
15. G. Hunt, and M. L. Scott, "The Coign Automatic Distributed Partitioning System," *Symp. on Operating System Design and Implementation (OSDI 1999)*, New Orleans, Feb. 1999
16. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.
17. M. Jones, D. Rosu, and M. Rosu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," *Symp. on Operating Systems Principles (SOSP 1997)*, 1997.
18. T. Kichkaylo A. Ivan, and V. Karamcheti. "Constrained Component Deployment in Wide-Area Networks Using AI Planning Techniques," *Int'l Parallel and Distributed Processing Symposium*, Nice, France, April 2003.
19. J. Kramer and J. Magee. "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol 16, 1990.
20. C. Lee, et al. "A Scalable Solution to the Multi-Resource QoS Problem," *IEEE Real-Time Systems Symposium*, 1999.
21. N. Medvidovic, R. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pages 70-93 (January 2000).
22. N. Medvidovic, M. Mikic-Rakic, N. Mehta, and S. Malek. "Software Architectural Support for Handheld Computing," *IEEE Computer*, vol. 36, no. 9, pages 66-73, September 2003.
23. N. Medvidovic, E. Dashofy, R. Taylor, "Moving Architectural Description from Under the Technology Lamppost," *Journal of Information and Software Technology*, vol. 49, no. 1, pages 12-31, January 2007.
24. S. Malek, "A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture," *PhD Dissertation*, University of Southern California, August 2007.
25. 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.

26. S. Malek, M. Mikic-Rakic, and N. Medvidovic, "A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems," *Int'l Working Conf. on Component Deployment (CD 2005)*, Grenoble, France, Nov. 2005.
27. 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," *Int'l Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006)*, Melbourne, Australia, November 2006.
28. S. Malek, C. Seo, S. Ravula, B. Petrus, and N. Medvidovic, "Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support," *Int'l Conf. on Software Engineering (ICSE 2007)*, Minneapolis, Minnesota, May 2007.
29. S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, and G. Sukhatme. "An Architecture-Driven Software Mobility Framework." *Journal of Systems and Software*, Special Issue on Architecture and Mobility, under review.
30. P. Manolios, G. Subramanian, and D. Vroon. "Automating Component-Based System Assembly," *International Symposium on Software Testing and Analysis*, July 2007.
31. D. Menasce, "Allocating Applications in Distributed Computing," *IEEE Internet Computing*, Vol. 9, No. 2, March 2005.
32. M. Mikic-Rakic, S. Malek, and N. Medvidovic, "Improving Availability in Large, Distributed, Component-Based Systems via Redeployment," *Int'l Working Conf. on Component Deployment*, Grenoble, France, 2005
33. M. Mikic-Rakic, and N. Medvidovic, "Support for Disconnected Operation via Architectural Self-Reconfiguration," *Int'l. Conf. on Autonomic Computing*, New York, May 2004.
34. M. Mikic-Rakic, S. Malek, N. Beckman, and N. Medvidovic, "A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings," *Int'l Conf. on Component Deployment*, Edinburgh, UK, May 2004.
35. M. Mikic-Rakic, S. Malek, and N. Medvidovic, "Architecture-Driven Software Mobility in Support of QoS Requirements," *Int'l Workshop on Software Architectures and Mobility (SAM)*, Leipzig, Germany, May 2008.
36. R. Neugebauer, et al. "Congestion Prices as Feedback Signals: An Approach to QoS Management," ACM SIGOPS European Workshop, 2000.
37. Object Management Group UML. http://www.omg.org/uml
38. P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-Based run-time Software Evolution," *Int'l. Conf. on Software Engineering (ICSE 1998)*, Kyoto, Japan, April 1998.
39. P. Oreizy, N. Medvidovic, and R. N. Taylor. "Runtime Software Adaptation: Framework, Approaches, and Styles," *Int'l Conf. on Software Engineering (ICSE 2008)*, Leipzig, Germany, May 2008.
40. H. Pham. "Software Reliability and Testing," *Wiley-IEEE Computer Society*, 1st edition, 1995.
41. F. Piedad. "High Availability: Design, Techniques, and Processes," *Prentice Hall*, January 2001.
42. V. Poladian, et al. "Dynamic Configuration of Resource-Aware Services," *Int'l. Conf. on Software Engineering*, 2004.
43. S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," *Prentice Hall*, Englewood Cliffs, NJ, 1995.
44. C. Seo, S. Malek, and N. Medvidovic, "An Energy Consumption Framework for Distributed Java-Based Software Systems," *Int'l. Conf. on Automated Software Engineering (ASE 2007)*, Atlanta, Georgia, November 2007.
45. C. Seo, S. Malek, and N. Medvidovic, "Component-Level Energy Consumption Estimation for Distributed Java-Based Software Systems," *International Symposium on Component Based Software Engineering (CBSE 2008)*, Karlsruhe, Germany, October 2008.
46. J. P. Sousa, R. K. Balan, V. Poladian, D. Garlan, and M. Satyanarayanan. "User Guidance of Resource-Adaptive Systems," *International Conference on Software and Data Technologies*, Porto, Portugal, July 2008.
47. V. Subramani, et al. "Distributed Job Scheduling on Computational Grids Using Multiple Simultaneous Requests," *Int'l Symposium On High Performance Distributed computing (HPDC 2002)*, Edinburgh, Scotland, July 2002.
48. W. Stallings, "Cryptography and Network Security," Prentice Hall, Englewood Cliffs, NJ, 2003.
49. E. A. Strunk, and J. C. Knight, "Dependability Through Assured Reconfiguration in Embedded System Software," *IEEE Transactions on Dependable and Secure Computing*, Vol. 3, No. 3, pp 172-187, July 2006.
50. R. N. Taylor, N. Medvidovic, and E. Dashofy. "Software Architecture: Foundations, Theory, and Practice," *John Wiley & Sons*, 2008.
51. L. A. Wolsey, "Integer Programming," *John Wiley & Sons*, New York, NY, 1998.
52. J. Zhang, and B. H. C. Cheng, "Model-Based Development of Dynamically Adaptive Software," *Int'l Conf. on Software Engineering (ICSE 2006)*, Shanghai, China, May, 2006.