A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture

Sam Malek[*] Nenad Medvidovic[*] Chiyoung Seo[*] Marija Mikic-Rakic[**]

[*]Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{malek,neno,cseo}@usc.edu

[**]Google Inc.
2644 30th Street
Santa Monica, CA, 90405
marija@google.com

Abstract

The quality of service (QoS) provided by a distributed software system depends on many system parameters, such as network bandwidth, reliability of links, frequencies of software component interactions, etc. A distributed system's deployment architecture can have a significant impact on its QoS. Furthermore, the deployment architecture will influence user satisfaction, as users typically have varying QoS preferences for the system services they access. Finding a deployment architecture that will maximize the users' overall satisfaction is a challenging, multi-faceted problem. In this paper, we present a framework model and a set of generic algorithms that can be tailored and instantiated to address this problem. We also provide an evaluation of our approach by applying it on a large number of representative scenarios.

1. Introduction

Consider the following scenario, representative of a large number of present-day distributed software applications. The scenario 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 "Commanders" in the field. The commander PDAs are connected directly to each other and to a large number of "Troop" PDAs. These devices communicate and help to coordinate the actions of their distributed users. The distributed software system running on these devices provides a number of *services* to the users: requesting and sending information to each other, viewing the current field map, managing the resources, etc.

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) preference for those services will differ, and may change over time. For example, in the case of a disaster search-and-rescue effort scenario, "Commander" users may require a secure and reliable messaging service with the "Headquarters" when exchanging search-and-rescue plans. On the other hand, "Troop" users may be more interested in having a low latency messaging service with other "Troop" users when sending assistance requests.

For any such large, distributed system, many deployment architectures (i.e., mappings of software components onto hardware hosts) will be typically possible. Some of those deployment architectures 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. This problem 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 hosts Y and Z).

In this paper, we consider the problem of finding a deployment architecture such that the QoS preferences (i.e., *utility*) accrued by a collection of distributed end-users is maximized. We would like our solution to be applicable to a wide range of *application scenarios* (i.e., differing numbers of users, hardware hosts, software components, application services, QoS dimensions, etc.). However, a widely applicable solution to this problem is challenged by the following: (1) A very large number of system parameters influence *QoS dimensions* (e.g., security, availability) of a software system; while it may be possible to identify a subset of *system parameters* (e.g., network bandwidth, frequencies of interactions) that influence the majority of QoS dimensions, it may not be possible to identify all of them; (2) Many services and their corresponding QoS influence the users' satisfaction; (3) Different QoS may be conflicting (i.e., improving one may degrade another); and (4) The space of possible deployment architectures is extremely large.

Therefore, a solution that meets the challenges identified above will need to (1) provide an extensible model that supports inclusion of arbitrary system parameters; (2) support definition of new QoS dimensions using the system parameters; (3) allow users to specify their QoS preferences; and (4) provide broadly applicable algorithms that find a solution (i.e., deployment architecture) which maximizes the users' satisfaction in a reasonable amount of time.

In this paper, we present a tailorable framework that is targeted at the above requirements. The framework relies on the notion of *QoS utility*, which indicates a user's (desired) degree of satisfaction with a system. The framework's objective is to maximize the overall utility, i.e., the cumulative satisfaction with the system by all its users. Given an application scenario, the system architect instantiates (configures) the framework by defining the appropriate system parameters and the QoS of interest. The framework is then populated with the actual data from a distributed application and users' preferences for the QoS dimensions of each application service. Finally, one of the algorithms supplied by the framework is used to find an improved deployment architecture. We provide an evalua-

tion of our approach by demonstrating its two key aspects: (1) tailorability via the framework's instantiation for different application scenarios, and (2) the algorithms' performance and accuracy in improving the users' satisfaction with the system.

We should note that ours is not the first effort of this kind. Several previous works [1,5,8,11] have realized the impact of a system's deployment architecture on its QoS and have explored techniques for finding an improved deployment of the system. However, these solutions are all either motivated by a particular application scenario or based on simplifying assumptions that make them inapplicable to most systems (e.g., single QoS dimension, single user, small system, and/or particular definition of a QoS dimension). Furthermore, since each approach has created its own, limited model of a system and domain-specific algorithms, it is extremely hard to compare and interchange the different approaches. The contribution of our work has been to address these shortcomings by constructing a generic framework that can be tailored to the specific needs of a multitude of scenarios. In the process, we have developed a highly expressive approach to specifying users' QoS preferences in terms of system parameters. We have devised and evaluated several algorithms that operate on the generic model and perform fine-grained trade-off analysis with respect to users' preferences. Our implementation of the framework results in an environment that allows the system architect to bridge system modeling, analysis, simulation, implementation, and deployment activities. Finally, the framework provides the means for comparing different solutions, which in turn paves the way for further research in this area.

The remainder of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents the framework's system model, its instantiation for an example scenario, and the accompanying algorithms. Section 4 discusses our tool support that realizes the framework. Section 5 provides evaluation results. Section 6 presents a discussion of the framework and its usage. Finally, the paper concludes with an outline of future work.

2. Related Work

Numerous researchers have looked at the problem of improving a system's QoS through resource scheduling [7] and resource allocation [9,14]. However, only a few have considered the users' preferences in improving QoS. The most notable of these approaches are Q-RAM [9] and the work done by Poladian et al. [15]. 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, which can be customized based on the available resources.

Several researchers have considered modifying a software system's deployment architecture to improve a specific QoS dimension of the system. Most notable of these are I5 [1], Coign [5], Kichkaylo et al. [8], and our own prior work [11,12]. I5 [1], proposes the use of the binary integer programming model (BIP) for generating an optimal deployment of a software application over a given network. This approach uses minimization of overall remote communication as the only criterion for optimality. Additionally, solving the BIP model is exponentially complex in the number of software components, rendering I5 applicable only to systems with very small numbers of software components and target hosts.

Coign [5] provides a framework for distributed partitioning of COM applications across the network. Coign monitors inter-component communication and then selects a distribution of the application that will minimize communication time, using the lift-to-front minimum-cut graph cutting algorithm. However, Coign can only handle scenarios with two-host client-server applications. Its authors recognize that the problem of distributing an application across three or more hosts is NP hard and do not provide approximative solutions for such cases.

Kichkaylo et al. [8], provide a model, called component placement problem (CPP), for describing a distributed system in terms of network and application properties and constraints, and an AI planning algorithm, called Sekitei, for solving the CPP model. The focus of CPP is to capture a number of different constraints that restrict the solution space of valid deployment architectures. However, CPP does not provide facilities for specifying the goal, i.e., a criterion function that should be maximized or minimized. Therefore, it only searches for a valid deployment that satisfies the constraints, without considering the quality of the found deployment.

In our own prior work [11,12], we devised a set of algorithms for improving a software system's availability by finding an improved deployment architecture. The novelty of our approach was a set of approximative algorithms that scaled well to large distributed software systems with many components and hosts. However, our approach was limited to a predetermined set of system parameters, and a predetermined definition of availability.

None of the above approaches (including our own previous work) considers the system users and their QoS preferences. Furthermore, none of these approaches attempt to improve more than one QoS dimension of interest. Finally, no previous work has considered users' QoS preferences at the granularity of the application-level services. Instead, the entire distributed software system is treated as one service with one user, and a particular QoS dimension serves as the only QoS objective.

3. Framework

In this section we describe the framework model, the framework's instantiation, and the accompanying generic algorithms.

3.1. Framework Model

Our primary objective in the design of the model has been to make it practical: enable it to capture realistic distributed system scenarios and avoid making assumptions that will restrict its applicability. For example, we wanted to avoid prescribing a predefined number of system parameters, or particular definitions of QoS dimensions. The framework model provides the minimum skeleton structure required to model a distributed system's deployment and the system parameters that affect that deployment. Each skeleton element of the model can be extended and arbitrarily refined by the system architect. Figure 1 shows the framework's formal model. We model a distributed software system as:

1. A set H of hardware nodes (hosts) with the associated parameters (e.g., available memory or CPU on a host), and a function *hParam* that maps each parameter to a value.
2. A set C of components with the associated parameters (e.g., required memory for component's execution or JVM version), and a function *cParam* that maps each parameter to a value.
3. A set N of physical network links with the associated parameters (e.g., available bandwidth, reliability of links), and a func-

A distributed system is modeled in terms of

1. a set H of hardware nodes, a set HP of host parameters, a function $hParam: H \times HP \rightarrow \Re$
2. a set C of components, a set CP of component parameters, a function $cParam: C \times CP \rightarrow \Re$
3. a set N of network links, a set NP of network link parameters, a function $nParam: N \times NP \rightarrow \Re$
4. a set I of logical links (interactions), a set IP of logical link parameters, a function $iParam: I \times IP \rightarrow \Re$
5. a set S of services, and a function $sParam: S \times \{H \cup C \cup N \cup I\} \times \{HP \cup CP \cup NP \cup IP\} \rightarrow \Re$ of values for service-specific system parameters
6. a set $DepSpace = \{d_1, d_2, ...\}$ of all possible deployment mappings, where $|DepSpace| = |H|^{|C|}$
7. a set Q of quality of services, a function $qValue: S \times Q \times DepSpace \rightarrow \Re$ that quantifies the achieved level of QoS, and $qType: Q \rightarrow \begin{cases} -1 & \text{if it is desireable to minimize this QoS} \\ 1 & \text{if it is desireable to maximize this QoS} \end{cases}$
8. a set U of users, a function $qosRate: U \times S \times Q \rightarrow [MinRate, 1]$ representing the rate of change in a QoS, and a complementary function $qosUtil: U \times S \times Q \rightarrow [0, MaxUtil]$ representing the utility for that rate of change
9. a set PC of parameter constraints, and a function $pcSatisfied: PC \times DepSpace \rightarrow \begin{cases} 1 & \text{if } constr \text{ is satisfied} \\ 0 & \text{if } constr \text{ is not satisfied} \end{cases}$
10. two functions that restrict locations of software components

$$loc: C \times H \rightarrow \begin{cases} 1 & \text{if } c \in C \text{ can be deployed onto } h \in H \\ 0 & \text{if } c \in C \text{ cannot be deployed onto } h \in H \end{cases} \quad colloc: C \times C \rightarrow \begin{cases} 1 & \text{if } c1 \in C \text{ has to be on the same host as } c2 \in C \\ -1 & \text{if } c1 \in C \text{ cannot be on the same host as } c2 \in C \\ 0 & \text{if there are no restrictions} \end{cases}$$

Figure 1. Framework Model.

tion *nParam* that maps each parameter to a value.

4. A set I of logical interaction links between software components in the distributed system, with the associated parameters (e.g., frequency of component interaction, average event size), and a function *iParam* that maps each parameter to a value.
5. A set S of services, and a function *sParam* that provides values for service-specific system parameters. An example service-specific system parameter is the number of component interactions resulting from an invocation of a single service.
6. A set *DepSpace* of all possible deployment mappings.
7. A set Q of QoS dimensions, and a function *qValue* that quantifies a QoS dimension (e.g., security) for a given service (e.g., "find the best route to the disaster area") in the current deployment mapping. Also, a function *qType* that represents the minimization or maximization aspect of the QoS dimension.
8. A set U of users, and two complementary functions *qosRate* and *qosUtil* that denote a user's preference for a QoS dimension of a service. *qosRate* returns the rate of change, while *qosUtil* returns the utility for that rate of change. For example, the user may denote a utility of 0.1 for a change of 0.2 (i.e., 20%) in a particular QoS dimension of a service. Relative importance of different users is determined by two threshold values: *MinRate* and *MaxUtil*. *MinRate* denotes the minimum rate of change a user is allowed to specify, while *MaxUtil* denotes the maximum utility. In general, a smaller value of *MinRate* and a larger value of *MaxUtil* indicates higher importance (or relative influence) of the user in the final solution.
9. A set *PC* of parameter constraints, and a function *pcSatisfied* that, given a constraint and a deployment architecture, returns 1 if the constraint is satisfied and 0 otherwise. For example, if the parameter constraint is "bandwidth satisfaction", the corresponding constraint function may ensure that the total volume of data exchanged across any network link does not exceed that link's bandwidth in a given deployment architecture.
10. Using the *loc* function, deployment of any component can be restricted to a subset of hosts. Using the *colloc* function, constraints on collocations of components can be specified.

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

For brevity we use the following shorthand notations in the remainder of this 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 is a set of components that constitute service s.

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

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

is maximized, and the following conditions are satisfied:

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

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 2. Problem Definition.

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

3.2. Framework Instantiation

To configure the framework model for an application scenario, its loosely defined parts must be specified. We illustrate the framework's instantiation

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

The first step in instantiating the framework is to define the relevant system parameters. Item 1 of Figure 3 shows a list of parameters that we have identified to be of interest for specifying the four QoS dimensions. We should note that additional parameters may be found to be relevant as well. Those parameters can be similarly instantiated in our framework. Once the parameters of interest are specified, the parameter realization functions (e.g., *hParam*, *cParam* of Figure 1) need to be defined. These discrete functions can be defined in many ways: monitoring the system, relying on system engineer's knowledge, extracting from the architectural description, etc. We provide a detailed discussion of this issue in Section 4, where we present our tool support for the framework.

A software system's *availability* is commonly defined as the degree to which the system is operational when required for use [6]. Availability is the QoS dimension denoting whether a service is present or ready for immediate use. In the context of distributed environments, where the most common failure is a network failure, we quantify availability as a function of successfully completed inter-component interactions in the system [11,12]. Item 2 of Figure 3 defines availability for a single service s in a given deployment d. A software service's *latency* is commonly defined as the time elapsed between making a request for service and receiving the response [6]. The most common causes of communication delay in a distributed system are the unreliability of network links, low bandwidth, and network transmission delay. Item 3 of Figure 3 defines latency for a service s. Note that for simplicity in our specification of latency we did not consider the computational delay associated with each software component's execution; however, it should be evident that the framework does not prevent one from including a more elaborate and accurate quantification of latency. A major factor in the *security* of distributed systems is the level of encryption (e.g., 128-bit versus 40-bit encryption) capability provided in remote communication [18]. Item 4 of Figure 3 defines communication security for a service s. Finally, *energy consumption* (or battery usage) of each service is determined by the energy required for the transmission of data among hosts plus the energy required for the execution of application logic in each software component for the service. Item 5 of Figure 3 defines energy consumption of a service s. For ease of exposition in this paper we have provided a simplified definition; a more sophisticated energy consumption model can be found in our recent work [17].

The definitions of the four QoS dimensions in Figure 3 are intended to serve primarily as an example of how QoS dimensions are quantified and used in our framework. A more detailed explanation of these QoS dimensions is beyond the scope of this paper. We do not argue that these are the only, "correct", or even most appropriate definitions for these four dimensions. In fact, our framework can accommodate arbitrary definitions of these as well as other QoS dimensions, so long as they are quantifiable.

For illustrating parameter constraints we use the memory available on each host. The constraint in item 6 of Figure 3 specifies that the total size of the components deployed on each host may not be greater than the total available memory on that host. Other constraints, such as "bandwidth satisfaction", are included in the same manner, but have been elided from the figure for brevity.

We also need to populate the set S with the services and the set U with the users of the system (recall Figure 1). Finally, the users' preferences are determined by defining the two functions *qosRate* and *qosUtil*. The users define the values these two functions take based on their preferences, as discussed in more detail in Section 4.

3.3. Framework Algorithms

The above problem is an instance of multi-dimensional optimization problems, characterized by many QoS dimensions, system

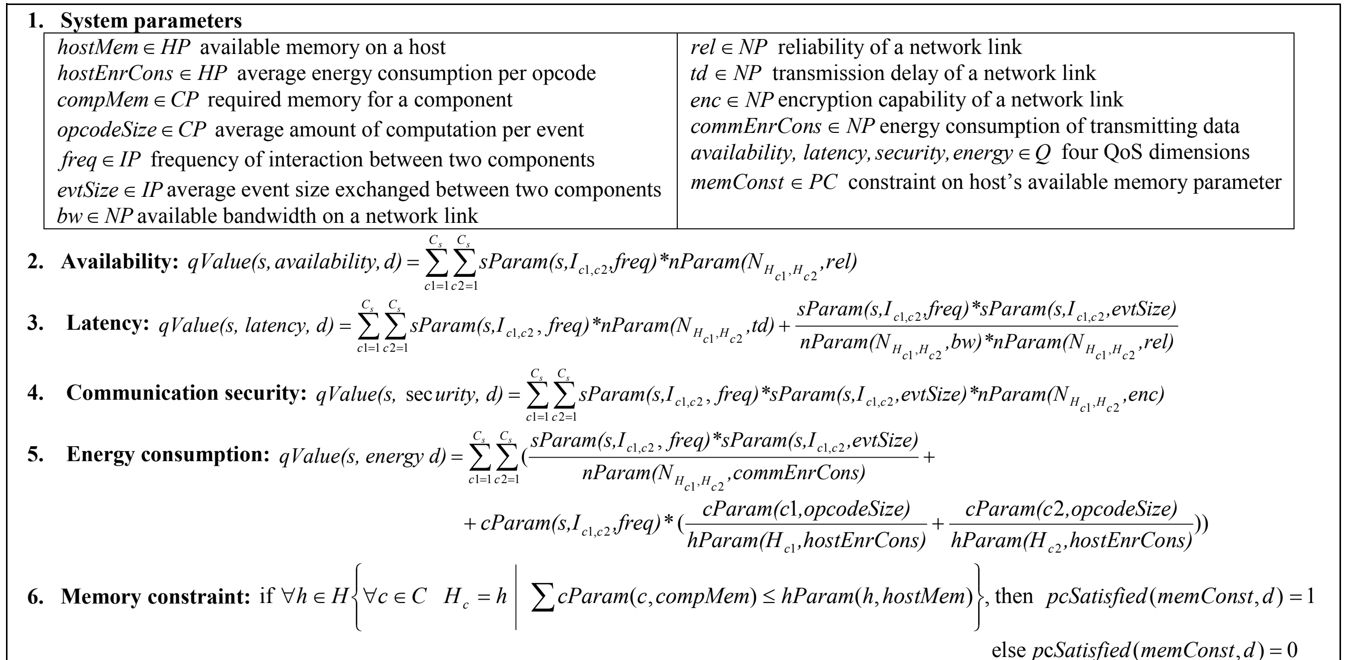


Figure 3. Framework instantiation example.

users and user preferences, and constraints that influence the objective function. Our objective has been to devise reusable algorithms that provide highly accurate results regardless of the application scenario. An in-depth study of the general applicable strategies resulted in four algorithms for solving the described problem, where each algorithm is suitable to a particular class of systems as will be further discussed in Section 6. Unlike previous works that depend on the knowledge of specific system parameters, we have developed a number of novel heuristics for improving the performance and accuracy of our algorithms independently of system parameters. Therefore, regardless of the specific application scenario the system architect simply executes the algorithm most suitable for the system (e.g., based on the size of the system, or stability of system parameters) without any modification.

Of the four general approaches we have adopted and adapted, two (Mixed-Integer Nonlinear and Mixed Integer Linear Programming, a.k.a. MINLP and MIP [19]) are best characterized as generic techniques developed in operations research to deal with multi-dimensional optimization problems. These techniques are accompanied by widely used algorithms and solvers. We tailor these techniques to target them specifically at our problem and introduce heuristics that 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. In this section, we provide a discussion of all four techniques, with an analysis of their algorithmic complexity as well as their inherent trade-offs.[1]

3.3.1. Mixed-Integer Nonlinear Programming (MINLP)

A linear (or non-linear) programming problem consists of three parts: decision variables, constraint functions, and an objective function. A linear programming problem has a linear objective function, while a non-linear programming problem has a non-linear objective function. In a mixed-integer programming problem [19] the decision variables can only take on integer values within a specified domain. As we will see, since our objective function is a non-linear function, our problem is by default a MINLP problem. While we cannot solve the MINLP representation of our problem optimally, off-the-shelf MINLP solvers can approximate a solution most of the time. In Section 5, we compare the solution of our two approximative algorithms (introduced below) to the solutions produced by these MINLP solvers. Furthermore, in the next section we demonstrate a technique for converting a MINLP problem into a *linear* programming problem, which can be solved optimally.

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

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

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

1. Due to space constraints we are only able to provide a brief background for each of the four strategies. This should allow an informed reader to follow our discussion of the specific enhancements we have introduced.

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

Finally, we need to define the objective function. Below we show the MINLP representation of the objective function for the scenario introduced in Section 3.2. In the interest of brevity, we are only showing availability and its contribution to the objective function; other QoS dimensions are included very similarly.

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

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

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

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

The function *availValue* corresponds to the *qValue* function defined in item 2 of Figure 3. The function *availUtil* calculates a user's utility for the availability of a service. *initAvail* is a constant which represents the availability of the service for the initial deployment of the system. Note that as a result of decision variable multiplication ($x_{c1,h1}*x_{c2,h2}$), the objective function is non-linear.

As mentioned earlier, while there are approximative algorithms for estimating a solution to a MINLP problem, there is no known algorithm for solving it optimally. Furthermore, for problems with non-convex functions (such as ours), MINLP solvers are not guaranteed to find and converge to an improved approximate solution [19]. Finally, given the non-standard techniques for solving MINLP problems, it is hard to determine a complexity bound for these MINLP solvers.[2] We provide an empirical evaluation and comparison of MINLP solvers with other algorithms in Section 5.

3.3.2. Mixed-Integer Linear Programming (MIP)

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

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

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

Using the new variable t and the three new constraints, we can rewrite the *availValue* function (as well as the other QoS functions)

2. All state-of-the-art MINLP solvers are based on confidential algorithms. Most vendors claim that their solvers run in polynomial time [3].

to arrive at an equivalent linear problem:

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

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

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

However, most MIP solvers support specification of the order in which the variables are to be branched by assigning different priorities to the decision variables. Thus, by assigning a higher priority to *x* variables and lower priority to *t* variables, we are able to reduce the complexity of the algorithm significantly, to $O(2^{|H||C|})$. This is because, after solving the problem for the *x* variables, the values of *t* variables trivially follow from the three constraints discussed in the previous paragraph. Finally, the constraint that each software component can be deployed on only one host (recall Section 3.3.1), allows for significant pruning of the branch-and-bound tree, thus reducing the complexity of our problem to $O(|C|^{|H|})$. Fixing some components to selected hosts, or specifying that a pair of components have to be collocated on the same host further reduces the algorithm's complexity [11].

As we will see in Section 5, even after all this reduction, the MIP algorithm remains computationally very expensive. It may still be used in calculating optimal deployments for systems whose characteristics are stable for a very long time. In such cases, it may be beneficial to invest the time required for the MIP algorithm, in order to gain maximum possible overall QoS utility. However, note that even in such cases, running the algorithm may become infeasible very quickly, unless the number of allowed deployments is substantially reduced through locational constraints.

3.3.3. Greedy Algorithm

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

Greedy algorithms are iterative algorithms that incrementally find better solutions. Unlike the previous algorithms that need to finish executing before returning a solution, a greedy algorithm generates a valid and improved solution in each iteration. This is a desirable characteristic for systems where the parameters change frequently and the available time for calculating an improved deployment varies significantly: whenever the algorithm is terminated, it returns either the initial deployment or one that is better than it. In each step of the algorithm, we take a single component *aComp* and estimate the new deployment location for it (i.e., a host) such that the objective function *overallUtil* is maximized. Our strategy is to improve the QoS dimensions of the "most important" services first. The most important service is the service that has the greatest total utility gain as a result of the smallest improvement in its QoS dimensions. The importance of service *s* is calculated via the following formula:

$$svcImp(s)=\sum_{u=0}^{|U|}\sum_{q=0}^{|Q|} qosUtil(u,s,q)/qosRate(u,s,q)$$

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

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

where *d'* is the initial deployment, *d* is the new deployment when *aComp* is deployed on *h*, and *p(S)* is a subset of services in whose provision *aComp* is involved.

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

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

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

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

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

However, since typically only a small subset of components participates in a given service and swappable components are only a small subset of components deployed on the *bestHost*, the average complexity of this algorithm is typically much lower. Finally, similar to the analysis of the MIP algorithm, specification of locational constraints decreases the number of times we calculate *hValue*, thus resulting in further complexity reduction.

An important heuristic we have introduced in this algorithm is the swapping of components: this significantly decreases the possibility of getting "stuck" in a bad local optimum, alleviating a common shortcoming of greedy strategies. Further enhancements to the algorithm are possible that would result in improving the results at the cost of higher complexity. For example, simulated annealing [16] could be leveraged to explore several solutions and return the best one by conducting a series of additional iterations over our algorithm.

3. This 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.

3.3.4.Genetic Algorithm

We present another approximative solution to our problem that is based on a well-known class of stochastic approaches called genetic algorithms [16]. Unlike the greedy algorithm, the genetic algorithm can easily be extended to execute in parallel on multiple processors with no additional overhead. Furthermore, in contrast with MINLP and greedy algorithms that eventually stop at "good" local optima, a genetic algorithm continues to improve the solution until it is either explicitly terminated by a triggering condition or the global optimal solution has been found. However, the performance and accuracy of the genetic algorithm significantly depend on its mechanism design (i.e., the representation of the problem and the heuristics leveraged in promoting the good properties of individuals). In fact, the genetic algorithm we developed initially, using conventional heuristics typically suggested in literature, significantly under-performed in comparison to the other three algorithms. Instead, we had to devise a novel mechanism specifically tailored at our problem, as discussed below.

In a genetic algorithm, an *individual* represents a solution to the problem. Each individual is composed of a sequence of *gene*s that represent the structure of that solution. A *population* contains a pool of individuals. An individual for the next generation of the population is evolved in three steps: (1) two or more parent individuals are heuristically selected from the population; (2) a new individual is created via a *cross-over* between the parent individuals; and (3) the new individual is *mutate*d 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 4a shows a simple representation of an individual for a problem of 10 components and 4 hosts. Each block of an individual represents a gene and the number in each block corresponds to the host that the component is deployed on. For example, component 1 of Individual 1 is deployed on host 4 (denoted by h4), as are components 4, 5, and 8. The problem with this representation is that the genetic properties of parents are not passed on to future generations as a result of cross-overs. This is because the components that constitute a service are dispersed in the gene sequence of an individual and a cross-over may result in a completely new deployment for the components of that service. For example, assume that in Figure 4a service 1 of Individual 1 and services 2 and 3 of Individual 2 have very good deployments (with respect to users' utility); then as a result of a cross-over, we may create a new individual that has an inferior deployment for all three services: 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 4b shows a mechanism we have developed for the representation of an individual in response to this problem. In this representation, the components of each service are grouped together via a mapping function, represented by the Map sequence. Each block in the Map sequence tells us the location on the gene sequence of an individual to which a component is mapped. For example, component 2 is mapped to block 5 on the gene sequence (denoted by i5). Thus, block 5 of Individual 1 in Figure 4b corresponds to block 2 of Individual 1 in Figure 4a, and both denote the fact that component 2 is deployed on host 1. If the component participates in more than one service, the component is grouped with the components providing the service that is most important. Similarly to the greedy algorithm, the most important service results in the highest utility gain for the smallest improvement in its QoS dimensions, calculated via the *svcImp* function from the previous section. We only allow cross-overs that occur on the borders of services. For example, in Figure 4b, we may perform cross-overs at two locations: the line dividing blocks 4 and 5, or the line dividing blocks 7 and 8 of Individuals 1 and 2. Note that as a result of the cross-over in Figure 4b, we have created an individual that has inherited the deployment of service 1 from *Individual1* and the deployment of services 2 and 3 from *Individual2*.

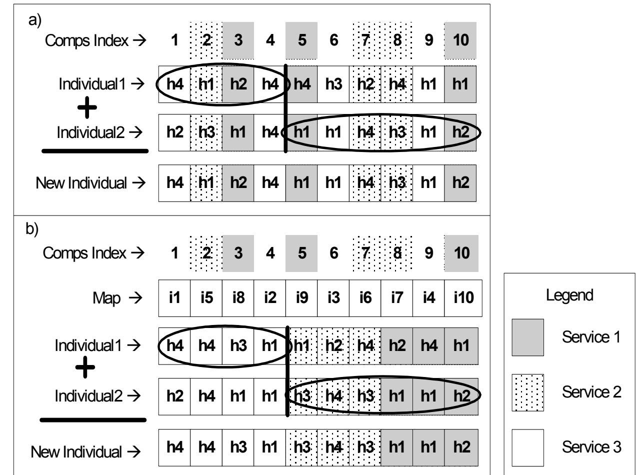


Figure 4. Application of the genetic algorithm for a problem of 10 components and 4 hosts: a) Simple representation. b) Representation based on services.

After the cross-over, the new individual is mutated. This corresponds to changing the deployment of a few components. To evolve populations of individuals, we define a fitness function that evaluates the quality of each new individual. The fitness function returns zero if the individual does not satisfy all the parameter and locational constraints, otherwise it returns the value of *overallUtil* for the deployment that corresponds to the individual. The algorithm improves the quality of a population in each evolutionary iteration by selecting parent individuals with a probability that is directly proportional to their fitness values. Thus, individuals with a high fitness value have a greater chance of getting selected, and increase the chance of passing on their genetic properties to the future generations of the population. Furthermore, we directly copy the individual with the highest fitness value (i.e., perform no cross-over or mutation on it) from each population to the next generation, thus keeping the best individual found in the entire evolutionary search. The worst case complexity of this algorithm is:

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

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

4. Tool Support

To provide an implementation of the framework we leveraged DeSi [13], our visual deployment exploration environment that supports specification, manipulation, visualization, and (re)estima-

tion of deployment architectures for large-scale, highly distributed systems. DeSi exports an API for modifying its deployment model, which can be used to define new system parameters. We leveraged this API to instantiate our framework. DeSi's visualization facilities automatically get updated to reflect the model. DeSi provides a pluggable utility for adding new algorithms that operate on the model, which we used to add the framework's algorithms. We instrumented DeSi to transform its internal model to GAMS [2], which is a popular algebraic optimization language. DeSi was then integrated with state-of-the-art MIP and MINLP solvers [3] for solving the generated GAMS models. DeSi was also extended to allow for specification of arbitrary QoS dimensions and parameter constraints, which are added to DeSi's model structure.

Once the appropriate framework model is specified and instantiated, the model is populated with the proper data about system parameters and user preferences. DeSi provides the option of either generating a random application scenario or acquiring the data from an external source. To acquire the data about system parameters from a real application, DeSi has been integrated with ArchStudio [4] and Prism-MW [10]. ArchStudio is an architecture-based software development environment which, at its core, uses an extensible architecture description language (ADL). DeSi uses ArchStudio to populate its model with system parameter values that are available at design time: initial deployment of the system, available memory on each host, etc. Prism-MW is a middleware platform that enables efficient implementation, deployment, and execution of distributed software systems. Prism-MW provides monitoring facilities that are used by DeSi to populate its model with system parameter values at runtime: frequencies of interaction among components, available bandwidth on each network link, etc.

Calculating an improved deployment architecture is only a part of the larger problem of improving the QoS in a running system. As part of our on-going work we have developed techniques and tools for system monitoring, component deployment and migration, and dynamic system adaptation [10,12,13]. The integration of the framework presented in this paper with our previous work provides a comprehensive solution to user-aware analysis, selection, and effecting of a system's deployment architecture.

5. Evaluation

In this section we evaluate the results of the algorithms discussed in Section 3.3 for several instances of the scenario introduced in Section 3.2. Recall that in this scenario we tailored the framework with four QoS dimensions. We leveraged DeSi's hypothetical deployment generation capability to create the scenario instances. In the generation of deployment scenarios all system parameters are populated with randomly generated data within a specified range, and an initial deployment of the system that satisfies all the constraints is provided. DeSi thereby enabled us to evaluate our approach on a large number of generated examples.

Figure 5 shows the input into DeSi for the generation of example scenarios and benchmarks. The values in Figure 5 represent the allowable ranges for each system parameter. The numbers of hosts, components, services, and users vary across the benchmark tests and are specified in the description of each test. Note that both the framework and DeSi are independent of the unit of data used for each system parameter. For example, in the case of transmission delay, neither the framework nor DeSi depend on the unit of time (*s*, *ms*, etc.). It is up to the system architect to ensure that the right units and appropriate ranges for the data are supplied to DeSi. After the deployment scenario is generated, DeSi simulates users' preferences by generating hypothetical desired rates of change (*qosRate*) and desired utilities (*qosUtil*) for the QoS dimensions of each service. While users may only use and specify QoS preferences for a subset of services, we evaluate our algorithms in the most constrained (and challenging) case, where each user uses all the services and specifies a QoS preference for each service. Unless otherwise specified, the genetic algorithm used in the evaluation was executed with a single population of one hundred individuals, which were evolved one hundred times. Our evaluation focused on five different aspects of our work, as detailed next.

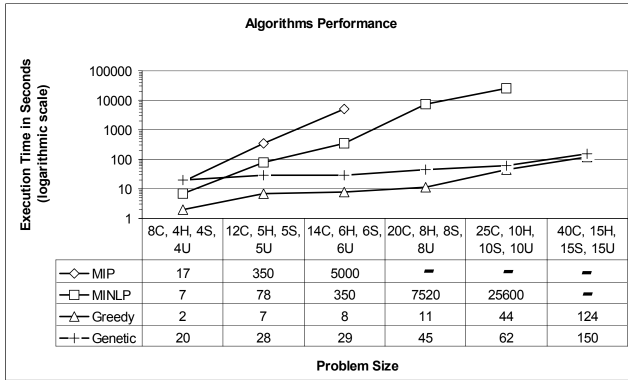
Improving Conflicting QoS Dimensions. Table 1 shows the result of running our algorithms on an example application scenario generated for the input of Figure 5 (with 12 components, 5 hosts, 8 services, and 8 users). The values in the first eight rows correspond to the percentage of improvement over the initial deployment of each service. The ninth row shows the average improvement for each QoS dimension of all the services. Finally, the last row shows the final value of our objective function (*overallUtil)*. The results demonstrate that, given a highly constrained system with conflicting QoS dimensions, the algorithms are capable of significantly improving QoS dimensions of each service. As discussed in Section 3.3.2, the MIP algorithm found the optimal deployment (with the objective value of 64 in this case).[4] The other algorithms also found good approximative solutions, which are

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

Figure 5. Input for DeSi's deployment scenario generation.

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

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



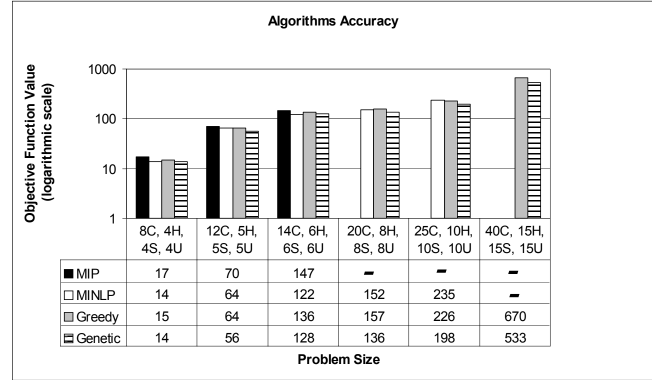


Figure 6. Comparison of the four algorithms' performance and accuracy.

within 20% of the optimal. Although space constraints prevent us from detailing other experiments, as an example we select a benchmark of 20 randomly generated application scenarios, which showed average improvements of 73% for availability, 61% for latency, 51% for security, and 66% for energy consumption. Our results indicate that our algorithms deliver the desired improvements in multiple conflicting QoS dimensions.

Sensitivity to Users' Preferences. Space constraints prevent us from showing the preference tables for each of the 8 postulated users and the 4 QoS dimensions of each of the 8 services. However, recall from Section 3.1 that the importance of a QoS dimension to a user (which we call *QoS importance*) is determined by the ratio of the user's *qosUtil* to *qosRate* for that dimension. Thus, QoS dimensions of services that on average have higher importance to the users typically show a greater degree of improvement. 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, which show a greater improvement of these dimensions for service 4 than their average improvement (e.g., in MIP's solution, availability of service 4 is improved by 215% and security by 302%; the average respective improvement of these two dimensions for all services was 86% and 66%). Note that, for this same reason, a few QoS dimensions of some services have degraded in quality, as reflected in the negative percentage numbers. These were not very important to the users and had to be degraded for improving other, more important QoS dimensions.

We select as another illustration a benchmark of 20 randomly generated application scenarios, which showed average QoS improvements of 89% for services for which the users specified a QoS importance of 1 or more, and 34% for services for which the user specified a QoS importance of less than 1. Since the benchmark scenarios were generated for the input of Figure 5, QoS importance could have taken values in the range of 0 to 100. However, a value of 1 represents the expected (statistical average) value that QoS importance could have taken, and divides the data distribution into two halves.[5] In this comparison we have compared the half with higher QoS importance values against the half with lower QoS importance values, which verifies that improvements in QoS are based on users' QoS preferences.

Performance and Accuracy. Figure 6 shows the comparison of the four algorithms in terms of performance (execution time) and accuracy (value of the objective function *overallUtil*). Note that the vertical axis is divided logarithmically, thus the slope of lines may be deceiving. For each data point (shown on the horizontal axis with the number of components, hosts, services, and users), we created ten representative problems and ran the four algorithms on them. The results correspond to the average values attained from these benchmarks. As shown, the high complexity of MIP and MINLP solvers made it infeasible to solve the larger problems. Comparing results of MINLP, greedy, and genetic algorithms against the optimal solution found by the MIP algorithm shows that all three approximative algorithms come within at least 20 percent of the optimal solution. The results also corroborate that the greedy and genetic algorithms are capable of finding solutions that are on par with those found by state-of-the-art MINLP solvers. On the other hand, our greedy and genetic algorithms demonstrate much better performance than both MIP and MINLP solvers, and are scalable to very large problems. The MINLP solvers were unable to find solutions for approximately 20% of larger problems (beyond 20 components and 10 hosts). However, for a meaningful comparison of the benchmark results we are not including problems that could not be solved by the MINLP solvers in Figure 6.

Sensitivity to QoS Dimensions. Figure 7 shows the sensitivity of each algorithm's performance to the number of QoS dimensions. We executed a deployment architecture of 12 components, 5 hosts, 5 services, and 5 users for varying numbers of QoS dimensions.[6] As expected, the performance of all four algorithms is affected by

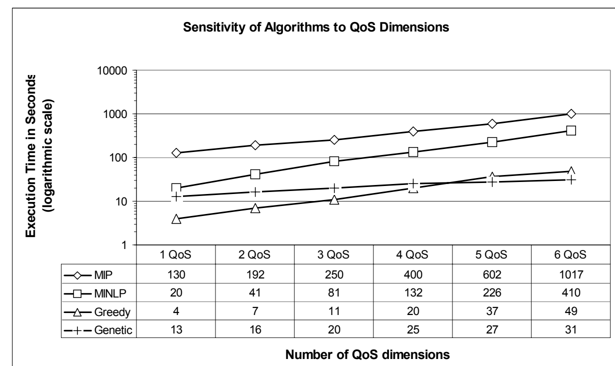


Figure 7. Sensitivity of performance to QoS dimensions.

4. An objective value has no absolute meaning, but is relative to the objective values of other deployments within the same application scenario.
5. Recall from Figure 5 that minRate=0.01 and maxUtil=1. Based on the definitions of Figure 1, qosRate is randomly selected from 0.01 to 1, and qosUtil is randomly selected from 0 to 1. Thus, expected average value for these two variables can be estimated to be 0.5. The expected average value for QoS importance is then equal to qosUtil/qosRate=0.5./0.5=1.

the addition of new dimensions. However, the algorithms show different levels of sensitivity to the addition of new QoS dimensions. The genetic algorithm shows the least amount of degradation in performance. This is expected, since the analysis in Section 3.3, suggests that the complexity of the genetic algorithm increases linearly in the number of QoS dimensions, while the complexity of the greedy algorithm increases polynomially. Even though we do not have access to the proprietary algorithms used by MIP and MINLP solvers, we can see that their performance also depends significantly on the addition of new QoS dimensions

Sensitivity to Heuristics. In Figure 8 we evaluate the heuristics we have introduced in the development of our algorithmic solutions. Figure 8a shows the effect of variable ordering on the performance of the MIP algorithm. As discussed in Section 3.3.2 and shown in the results of Figure 8a, 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 8b compares the greedy algorithm against a version that does not swap components when the parameter constraints on the *bestHost* are not satisfied. As was discussed in Section 3.3.3, by swapping components we decrease the possibility of getting "stuck" in a bad local optimum. The results of Figure 8b 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 8c compares three variations of the genetic algorithm. The first two variations were discussed in Section 3.3.4, where one uses the Map sequence to group components based on service and the other does not. As expected, the results show a significant improvement in accuracy when components are grouped based on services, by up to a factor of 3. The last variation corresponds to the distributed and parallel execution of the genetic algorithm. In this variation we evolved three populations of one hundred individuals in parallel, where the populations shared their top ten individuals after every twenty evolutionary iterations. The results show a small improvement in accuracy (along with the expected significant time savings) over the simple scenario where only one population of individuals was used.

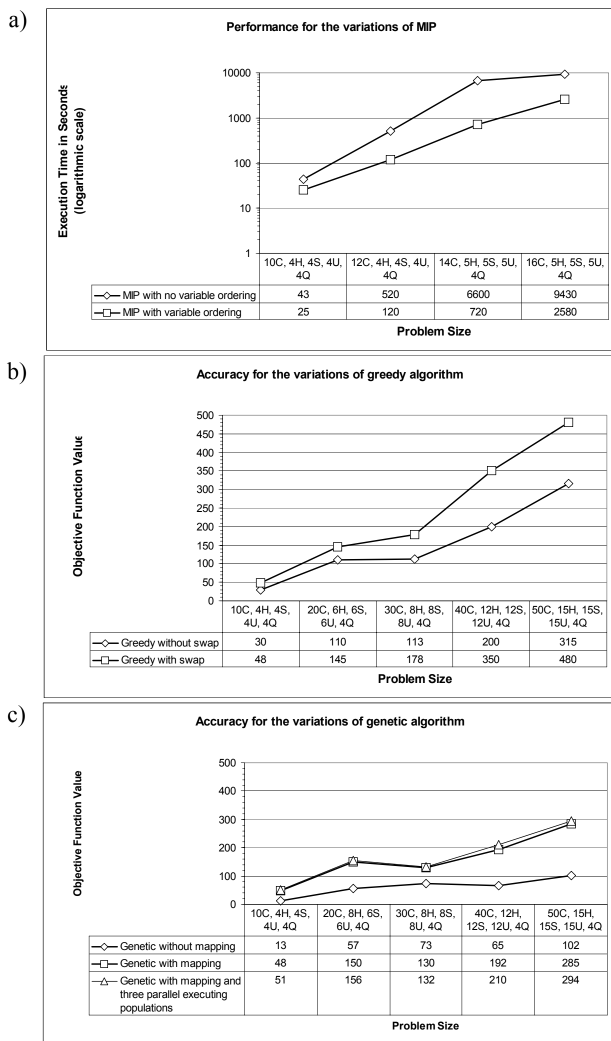


Figure 8. Results of the heuristics used by the algorithms.

6. Recall from Section 3 that our framework allows arbitrarily specified QoS dimensions. This allowed us to simply introduce "dummy" QoS dimensions as needed in the course of our evaluation.

6. Discussion

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

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

lem space remains exponentially large. For any even medium-sized Peer-to-Peer system, the MIP algorithm becomes infeasible and the software engineer has to leverage one of the three approximative algorithms to arrive at a sub-optimal, but significantly improved deployment architecture.

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

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

7. Conclusion and Future Work

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 quality of deployment in a particular scenario, 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. In this paper, we have presented a novel, extensible framework for improving a software system's QoS by finding the best deployment of software components onto the available hardware hosts. The contribution of our approach is a QoS trade-off model and accompanying generic algorithms, whereby given the users' preferences for the desired levels of QoS, the most suitable deployment architecture is found. We demonstrated the tailorability of our solution and its ability to handle trade-offs between QoS dimensions by instantiating it for four representative, conflicting dimensions. We also discussed four generic approaches to solving our multi-dimensional optimization problem, and presented several domain-specific 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. The framework provides a uniform approach to compare algorithmic solutions to this problem, and provides a solid foundation for future research and development of new distribution scenarios and algorithms. This work is part of an integrated solution, in which the data about system parameters are either acquired at design-time (via an ADL or system architect) or at runtime (via monitoring), and an improved deployment architecture is calculated and effected [10,12,13].

While our results have been very positive, a number of pertinent questions remain unexplored. Our future work will span issues such as considering a negative utility for the inconvenience of changing a system's deployment architecture at runtime, and providing autonomic solutions for the selection of the appropriate algorithm(s) based on system characteristics.

8. References

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

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

[3] J. Czyzyk, et. al. The NEOS Server. *IEEE J. Comp. Science and Engineering*, pages 68-75, 1998.

[4] E. Dashofy, et. al. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. *Int'l Conf. on Software Engineering*, Orlando, FL, May 2002.

[5] G. Hunt, et. al. The Coign Automatic Distributed Partitioning System. *Symposium on Operating System Design and Implementation*, New Orleans, Feb. 1999.

[6] IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

[7] M. Jones et. al. CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities. *Symposium on Operating Systems Principles*, 1997.

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

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

[10] S. Malek, et. al. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. on Software Engineering*, Vol. 31, No. 4, March 2005.

[11] M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. *Int'l. Conf. on Component Deployment*, Grenoble, France, 2005

[12] M. Mikic-Rakic, et. al. Support for Disconnected Operation via Architectural Self-Reconfiguration. *Int'l. Conf. on Autonomic Computing*, New York, May 2004.

[13] M. Mikic-Rakic, et. al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *Int'l. Conference on Component Deployment*, Edinburgh, UK, May 2004.

[14] R. Neugebauer, et. al. Congestion Prices as Feedback Signals: An Approach to QoS Management. *ACM SIGOPS European Workshop*, 2000.

[15] V. Poladian et al. Dynamic Configuration of Resource-Aware Services. *ICSE*, 2004.

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

[17] C. Seo, et. al. An Energy Consumption Framework for Distributed Java-Based Software Systems. Submitted to *ACM SIGSOFT 2006 / FSE 14*.

[18] W. Stallings. *Cryptography and Network Security*. Prentice Hall, Englewood Cliffs, NJ, 2003.

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