Software Deployment Architecture and Quality-of-Service in Pervasive Environments

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA, 90089-0781 U.S.A.
neno@usc.edu

Sam Malek
Computer Science Department
George Mason University
Fairfax, VA 22030-4444 U.S.A.
smalek@gmu.edu

ABSTRACT

Over the past several years we have investigated two problems related to the domain of highly distributed, mobile, resource constrained, embedded, and pervasive environments: software deployment and quality of service (QoS). We have done so with the explicit focus on the role played by software architecture in deployment, and on its relationship to QoS. In the process, we have amassed a body of knowledge and experience, and assembled a suite of solutions for targeting different facets of the interplay among software architecture, deployment, and QoS. At the same time, the area we are addressing has proven to be multi-faceted and very complex, constantly presenting new challenges. In this paper we outline the contours of the problem of QoS in architecture-based deployment, our strategy for addressing it, and the challenges that remain. We view this as an important (and fruitful) area of research.

1. INTRODUCTION

Software architecture is a collection of models that capture a software system's principal design decisions in the form of components (foci of system computation and data management), connectors (foci of component interaction), and configurations (specific arrangements of components and connectors intended to solve specific problems) [9]. Architecture realizes a system's functional requirements, that is, the services the system is meant to provide to its users. Additionally, architecture must ensure the level of quality at which those services are to be delivered, referred to in short as *quality of service* or *QoS*. Different dimensions of QoS are latency, availability, durability, reliability, security, fault-tolerance, survivability, dependability, scalability, heterogeneity, and so on.

In the domain of pervasive systems, of particular interest is a specific facet of a software system's architecture — its *deployment* [2]. Simply put, a system's deployment architecture is the allocation of the system's software components (and connectors) to its hardware hosts. Deployment architecture is particularly important in pervasive environments because a system will typically comprise many different, heterogeneous, mobile, and possibly mutable, execution platforms during its lifetime. Figure 1 shows an example such system for illustration: a particular instance of the MIDAS family of wireless sensor network-based systems developed in collaboration between the Bosch Research and Technology Center and our research group. The interested reader can find the deployment architecture of this instance of MIDAS in [7].

Previous research [1][4], including our own [5][8], has shown that a software system's deployment architecture can have a significant effect on the system's non-functional properties, i.e., it will affect the system's delivered QoS. This is further evident in pervasive systems [7], which are often long-lived, and execute in highly heterogeneous and unpredictable computing environments. Consider a pervasive system depicted in the two diagrams in Figure 2 for instance. These are two different deployment architectures for a subset of the Emergency Deployment System (EDS) [6] running on a variety of hand-held devices. Each deployment architecture depicted in Figure 2 comprises 17 software components distributed across five hardware hosts. The solid lines connecting the components represent their interaction paths, while the dotted lines connecting the hosts represent network connectivity. The two diagrams in the figure depict the same system, but with several components repositioned across the hosts.

Even though the actual instances of (the subset of) EDS corresponding to these two diagrams provide the same functionality (i.e., the same services) to the system's end users, they do so with different QoS levels for the system's different services. Thus, for example, both hosts 1 and 2 in Figure 2b will be running more components than their counterparts in Figure 2a. This means that they will likely consume more energy and run out of battery power more quickly (affecting many of

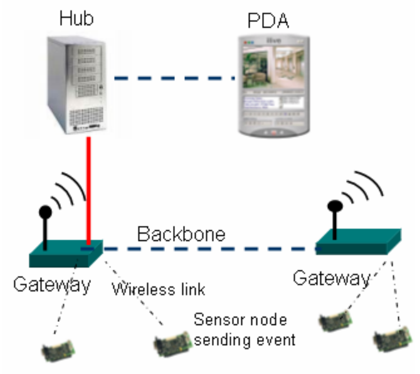


Figure 1. An example wireless sensor network application deployed across a number of heterogeneous, mobile and/or resource constrained platforms.

ESSPE '07, September 4, 2007, Dubrovnik, Croatia
Copyright 2007 ACM ISBN 978-1-59593-798-8/07/09...$5.00

the above properties, such as durability, availability, survivability, and so on). Furthermore, the number of (energy costly) interactions taking place across the network link between the hosts 1 and 2 will likely be significantly larger in Figure 2b than in Figure 2a. This means that the latency of the involved services will increase significantly, while the system's durability will, again, decrease.

This example illustrates that determining an "ideal", or even merely "good" deployment architecture for a system is a non-trivial task. This is the case for three reasons:

1. The total number of possible deployments for a given system is exponential. For a system comprising *h* hardware hosts and *c* software components the number of deployments is h^c. This means that, even for moderately sized systems, determining the best deployment will require evaluating a very large number of options.
2. The question of which deployment is "better" does not have a straightforward answer. For example, consider the two deployments in Figure 2: which one is better? In order to answer the question, first we would have to define the "goodness" of a deployment architecture. Even if we can agree that "good" means "that which exhibits shorter latency, longer durability, higher availability, better fault-tolerance, higher scalability, and so on", the specific definitions of those individual QoS dimensions will very likely differ from one developer to another, or one organization to another — a cursory overview of software engineering literature will attest to that. Furthermore, certain QoS dimensions are difficult to quantify. Examples include security, dependability, heterogeneity, and so on.
3. The system may be severely constrained. For example, certain devices may be unable to host certain (combinations of) components either due to insufficient capacity, because they are missing runtime resources, or because of other concerns such as security; certain groups of components may be required to be collocated on the same device; other components may be barred from residing on the same host; and so on. Constraints such as these may be particularly pronounced, and may even change at runtime as the system's execution context changes, in pervasive environments. Even in the case of relatively small systems, it quickly becomes impossible for a human architect to make appropriate deployment decisions.

In the remainder of this paper, we will first consider one critical challenge in modeling a deployment architecture and assessing it for a given set of QoS dimensions: the problem's multidimensional nature. We will present a particular solution for dealing with this problem. We will then briefly touch upon the problem's other critical challenge: its inherent complexity, especially in pervasive environments. We will also outline a strategy for dealing with this challenge. We close out the paper with a discussion of a number of significant issues that remain unresolved.

2. DEPLOYMENT AND QOS

Consider a very simple software system, derived from EDS, consisting of one service (*Schedule*) provided by two components (*Modify Resource Map* and *Resource Monitor*) that need to be deployed on two hosts (a laptop and a PDA).[1] The only QoS dimension of interest in this system is latency. These different elements are depicted conceptually in Figure 3a. For this system, four possible deployments are possible, and it would also be relatively easy to measure (or estimate, provided appropriate models and analytical tools are available to the architects) their latencies, as shown in Figure 3b. It is clear that the first deployment has the shortest latency. Thus, it is the optimal deployment.

However, if we expand the problem only slightly, by introducing another QoS dimension—durability (shown in Figure 4a), we get the situation depicted in Figure 4b. If the objective is to minimize the latency and maximize the durability in this system, none of the four deployments can be argued to be optimal. This phenomenon is known as *Pareto Optimal* in multidimensional optimization.

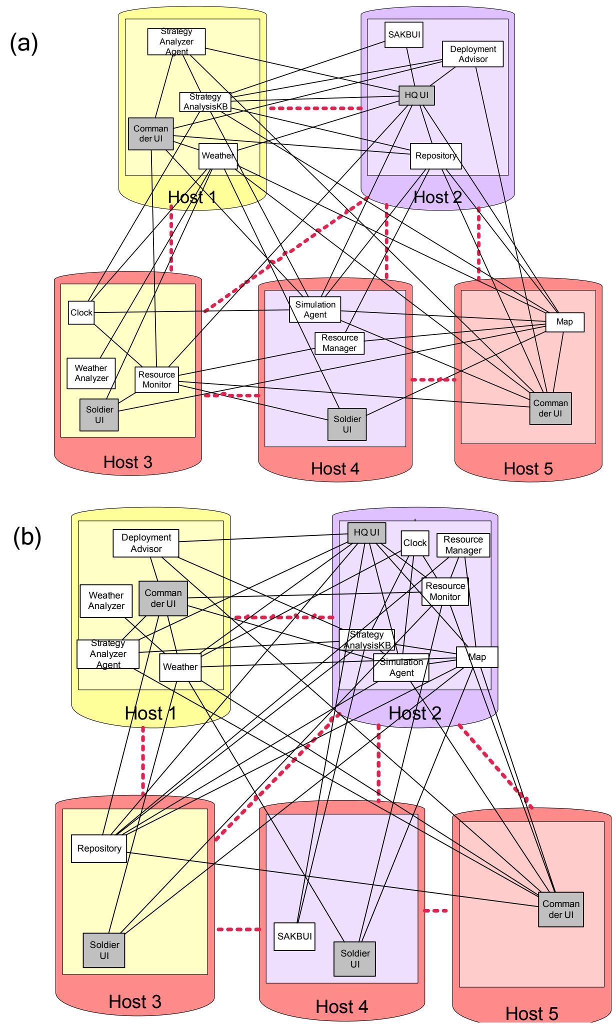


Figure 2. Two candidate deployment architectures for a distributed handheld application.

[1] Note that our notion of service is slightly different from that usually used in the service-oriented computing literature. In this paper, a service denotes a unit of system-level functionality (e.g., a user-observable task) that is provisioned via the collaboration and interaction among several software components (i.e., via multiple component-level services). Our use of this term is not critical to the paper's argument.

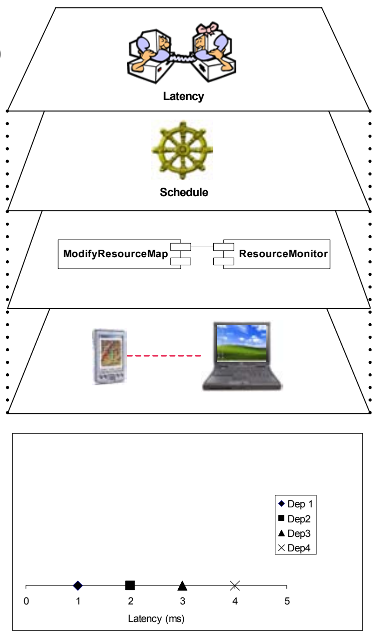


Figure 3. (a) An abstract depiction of a simple deployment problem. (b) The latency values measured for the four possible deployment architectures.

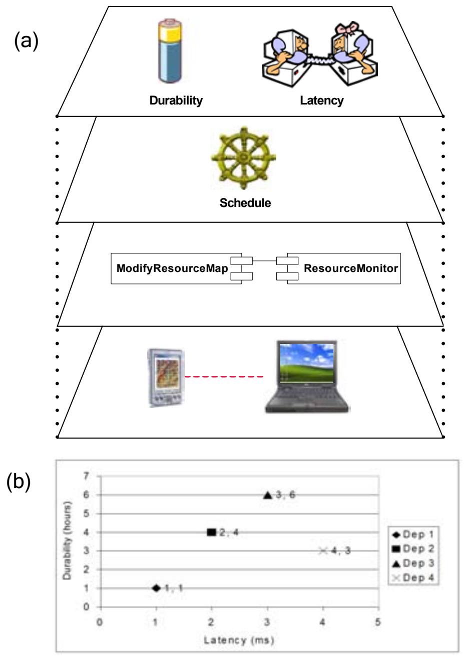


Figure 4. (a) An abstract depiction of a slightly expanded deployment problem. (b) The latency and durability values measured for the four deployment architectures.

A well known technique for dealing with the trade-offs in a multidimensional problem is to leverage a mapping function that arbitrates between conflicting/competing solutions. We can do this in the case of a software system's deployment architecture by explicitly capturing system users who, in turn, express preferences for the different system services (i.e., the *utility* that achieving a given level of quality for that service would have for that user). The utility function allows us to map the multidimensional problem to a uni-dimensional one. Therefore, the accuracy of the analysis performed in this manner depends on our ability to estimate the users' preferences accurately. The users' preferences can take various forms:

- functions describing continuously the utility of a system service;
- specific sets of discrete values;
- vague statements of utility (e.g., "ideally, the latency should be under 1ms; anything over 10ms will be unacceptable");
- "do not care"s for specific services;
- and so on.

Furthermore, the users themselves may be ranked or categorized according to their importance. This would further help architects to decide which deployment architectures to select.

Figure 5 builds on the example from Figure 4 after the introduction of a single user and the user's utility functions for latency and durability. If we consider deployment 2 to be the initial deployment (with the utility set to zero, for simplicity), the other three deployments' utilities can be calculated and compared to it. In this (hypothetical) case, deployment 3 has the highest total utility to the system's user and can be considered to be the optimal deployment for this system.

However, even this solution quickly runs into problems. Let us consider a relatively small scenario, consisting of three users who are interested in three QoS dimensions in accessing two services provided by three software components deployed on three hosts. This scenario is depicted in Figure 6. In this case, the architect will somehow have to capture 18 utility functions (3 users * 3 QoS dimensions * 2 services) and compare 27 different deployments (3 hosts and 3 components, or 3^3 deployments). Therefore, the problem becomes intractable for a human very quickly.

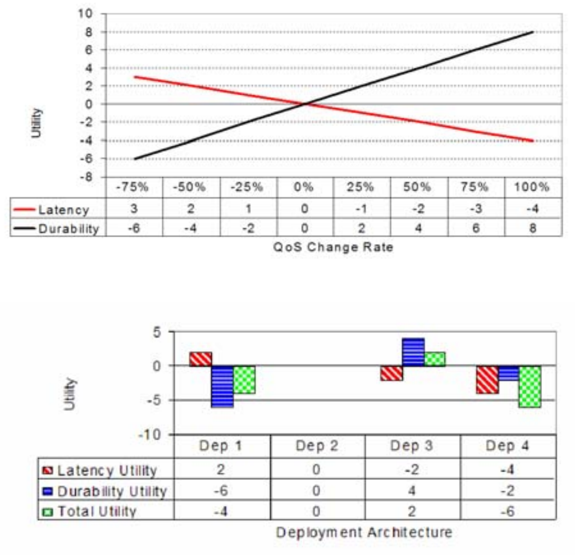
3. FORMALIZATION AND AUTOMATION

One obvious solution to the above situation is to take the problem out of the hands of a human architect and build powerful, comprehensive models of a system's deployment architecture. Formalizing the models would, in turn, enable their automated processing. These models would have to capture, at the least,

- system elements (hardware hosts and network links, software components and connectors) and their myriad parameters;
- functions or other ways of quantifying the system parameters;
- functions or other ways of quantifying the QoS dimensions of interest;
- users and their preferences; and
- constraints on the system elements and/or their parameters.

A partial formalization of the above system elements, with four QoS dimensions, is shown in Figure 7. This formalization is used for illustration only; space constraints prevent us from elaborating on it further. A detailed design can be found in [5]. For any sizeable distributed system, these models will be very large. However, they will be "built once and used forever", meaning that the cost of constructing them for most systems is likely not to be prohibitive.

Once constructed, such a model can be used to define a system's overall utility, i.e., to calculate and compare the quality of the



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

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

Figure 5. Introducing a user and a notion of utility allows the calculation of a single scalar value denoting a deployment architecture's overall quality.

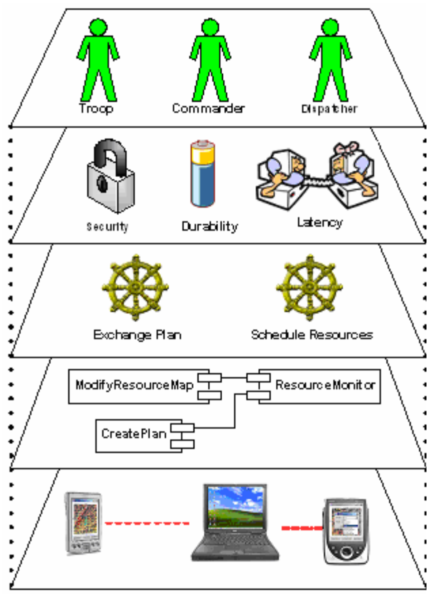


Figure 6. A slightly larger deployment scenario.

system's different deployment architectures. There are known off-the-shelf techniques for doing this (e.g., mixed integer non-linear programming, or MINLP, and mixed integer linear programming, or MIP) [10]. There are also other distributed and decentralized algorithms that can be applied to the problem [8].

4. REMAINING CHALLENGES

While the solution outlined above represents a promising attack on the problem, several challenges must be addressed. One challenge is the sheer size of the problem, in terms of the numbers of components and hosts, and especially in terms of the number of captured system parameters. This may prevent effective visualization of the systems in question, even with the aid of software tools. In turn, this carries the risk that the system will be modeled incorrectly, undermining any decision making ability by the software architects.

Another challenge is that the algorithms mentioned above are heuristic-based since the problem is exponential. This means that, for large systems, it is not possible to ascertain the actual quality of the solution produced by one of these algorithms. Furthermore, some of the algorithms (e.g., both MINLP and MIP) may not be able to guarantee a solution for problems of certain size and complexity.

Since pervasive systems are likely to be long lived and dynamic, they will demand deployment, and *redeployment*, solutions that are able to adjust to the system's changing execution context continuously. This means that, while the system's architects may choose a computationally expensive deployment strategy initially (e.g., a precise but inefficient deployment algorithm), during the system's execution they may be forced to switch to light-weight system monitoring and fast, though less precise, redeployment calculations. Another factor that must be taken into account is the amount of downtime the system will experience in order to redeploy: a deployment architecture that is otherwise inferior may be preferable if it can be effected more quickly.

Yet another challenge deals with how one can obtain the utility quantifications for the system's many services by its many distributed users, especially in a pervasive environment. For many pervasive systems, it is unlikely that architects will have access to all the system users. Even if they did, it is unlikely that the users would be able to articulate their preferences in a manner that is easily captured and/or quantified. This means that it may be necessary to develop techniques to model different classes of users or usage scenarios, and provide suites of deployment architectures (rather than a single solution), which will then be applied in different circumstances.

It should also be recognized that, by reducing a multi-dimensional problem to a uni-dimensional one, some modeling expressiveness and some of the captured information will be inevitably lost. The question is whether that loss of information will happen at the expense of the model's utility and precision. This is difficult to answer since large segments of this problem area remain unexplored and the actual target solution(s) in any given scenario (e.g., the best or even good deployment architecture for a given distributed system) may not be known. It may also be possible to explore other techniques for dealing with multi-dimensional optimization problems, e.g., multi-objective genetic algorithms [3].

It should also be noted that appropriate system deployment (and redeployment) is only one possible approach for ensuring the desired QoS level. Techniques such as component replication will directly aid certain QoS dimensions (e.g., latency, availability, and fault-tolerance). Likewise, data replication, caching, hoarding, and pre-fetching can also significantly improve the manner in which a system delivers its services. It is possible, in principle, to use these techniques in tandem with deployment (e.g., by appropriately expanding a system model such as that depicted in Figure 7). However, the exact effect of each of these potential solutions on deployment architecture would need to be studied further.

The final issue we wish to highlight is in many ways the pre-condition to any effective solution relating a software system's architecture (i.e., model) on the one hand and the system itself (i.e., implementation) on the other. In most large, real-world software systems, the relationship between architecture and implementation is a complex one, further complicated by

A distributed system is modeled in terms of

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

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

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 a host's available memory parameter*
$bw \in NP$ available bandwidth on a network link	

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

3. Latency:

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

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

5. Energy consumption:

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

6. Memory constraint:

$$if\ \forall h \in H \left\{ \forall c \in C \quad H_c = h \ \middle|\ \sum cParam(c, compMem) \le hParam(h, hostMem) \right\}, then\ \ pcSatisfied(memConst, d) = 1\ else\ pcSatisfied(memConst, d) = 0$$

Figure 7. A partial deployment architecture model, used for illustration.

architectural drift and erosion [9]: each architecture-level design decision may be realized by many implementation-level modules; furthermore, each implementation-level module may participate in realizing multiple architecture-level decisions. The solutions we have outlined above do not try to artificially oversimplify the relationship between the system's model and the system itself. However, we have assumed that this relationship is established early on during a system's lifecycle, and carefully managed thereafter. We believe that without such an assumption, no guarantees about a system's QoS can be made from any of its models (including that of its deployment architecture).

5. 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] A. Carzaniga, et. al. A Characterization Framework for Software Deployment Technologies. Tech. Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, 1998.

[3] K. Deb, et al. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Trans. on Evolutionary Computation*, Vol. 6, No 2, 2002.

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

[5] S. Malek. A User-Centric Approach for Improving a Distributed Software System's Deployment Architecture. Ph.D. Dissertation, USC, May 2007.

[6] S. Malek, et. al.. A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems. *IEEE Trans. on Software Engineering*, March 2005.

[7] S. Malek, et. al. Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support. In *Proceedings of ICSE 2007*, Minneapolis, MN, May 2007 .

[8] M. Mikic-Rakic, et. al. Improving Availability in Large, Distributed, Component-Based Systems via Redeployment. In proceedings of the *3rd Int'l. Working Conference on Component Deployment*, Grenoble, France, Nov. 2005

[9] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17:4, October 1992.

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