Architecture-Driven Software Mobility in Support of QoS Requirements

Marija Mikic-Rakic
Google Inc
1333 2nd Street
Santa Monica, CA, 90401 U.S.A.
marija@google.com

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

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

ABSTRACT

Over the past decade researchers have shown that software architecture provides an appropriate level of granularity for assessing a system's Quality of Service (QoS) properties (e.g., latency). Similarly, many previous works have adopted an architecture-centric approach to reason about the runtime adaptation, including component mobility, of software systems. However, the relationship between software architecture, QoS, and mobility is not clearly understood. In this paper, we present a framework that takes an explicit software architecture perspective for assessing the system's QoS properties, and improving it through architectural mobility. We describe the implementation of the framework, as well as some of the remaining challenges that frame our ongoing work.

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 interaction), and configurations (specific arrangements of components and connectors) [13]. 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 as quality of service (QoS). Different dimensions of QoS are latency, availability, durability, reliability, security, fault-tolerance, etc.

The QoS provided by a distributed software system depends on many system parameters, such as network bandwidth, reliability of links, frequencies of component interactions, etc. At the same time, many distributed systems are challenged by the fluctuations in those parameters due to the frequent mobility of the hardware and changes in the system's execution context. For example, host mobility may result in different connection properties for that host, which may impact QoS such as availability and latency.

Software architecture has been applied with some success in facilitating mobility [3,4]. The resulting techniques enable a system's evolution while minimizing architectural drift, as units of mobility are explicit architectural constructs (software components and connectors). More recently, software architecture has also been leveraged to facilitate QoS satisfaction [5,10]. However, what remains unclear is the nature of the aggregate relationship of software architecture, mobility, and QoS. Our hypothesis is that these three areas can be effectively combined by a uniform framework for designing, analyzing implementing, deploying, monitoring, and evolving distributed, mobile software systems, such that their desired QoS is achieved and architectural drift avoided throughout the system's lifetime. In support of this hypothesis, our approach comprises:

- *Architectural modeling*, both at the level of software components, connectors and their configurations, as well as the relevant facets of the hardware platforms on which the software elements will execute. Additionally, we support modeling of the QoS requirements in terms of the relevant parameters of the architectural constructs (both hardware and software).
- *Analysis* to determine the system's optimal deployment (mapping of the software architecture onto the hardware configuration) based on the QoS requirements and the distributed system's hardware and software properties.
- *Implementation* in terms of explicit architectural constructs provided at the level of source code.
- *Deployment* of the architecture-based software implementation onto the hardware configuration, as determined during analysis.
- *Runtime monitoring* of system properties that influence QoS.
- *Runtime evolution* mechanisms that react to the changes in a system's properties in an attempt to preserve or improve its QoS.

Together these techniques comprise an extensive, flexible framework for architecture-driven software system mobility.

The key observation underlying our framework is that mobility at the architectural level can be treated as a special case of a change to the system's *deployment architecture* (i.e., allocation of the system's software components to its hardware hosts). The deployment architecture of a software system has a significant impact on its QoS. 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. Therefore, a redeployment of the software system via migration of its components may be necessary to improve its QoS. In this paper, we provide an overview of the framework, and our experiences with applying it in order to improve deployment architectures of distributed software systems. We also discuss some of the remaining challenges that frame our ongoing work.

The rest of the paper is organized as follows. Section 2 outlines our framework for studying the relationship between architecture, mobility, and QoS. Section 3 presents an overview of our experience to date with the framework, while Section 4 relates this approach to existing work. We conclude the paper with the discussion of challenges that are framing our ongoing work.

2. MOBILITY FRAMEWORK

Our framework for supporting system mobility takes an explicit software architectural perspective.The framework provides facilities for architectural modeling, analysis, implementation, deployment, monitoring, and runtime evolution via redeployment. In this section, we discuss all of the framework's principal elements.

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
   $qType: Q \rightarrow \begin{cases} -1 & \text{if it is desireable to minimize this QoS} \\ 1 & \text{if it is desireable to maximize this QoS} \end{cases}$
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.

2.1. Architecture Modeling and Visualization

To be able to analyse a mobile software system at runtime, one needs to model not only the system's software architecture, but also the system's context, which may include the hardware, and network characteristics. Each of these elements may be associated with arbitrary parameters. The selection of a set of parameters to be modeled depends on the criteria (i.e., QoS objectives) that a system's deployment architecture should satisfy. For example, if minimizing latency is one of the objectives, the model should include parameters such as physical network link delays and bandwidth. Finally, the system users' usage of the functionality (i.e., *services*) provided by the system, and the users' QoS preferences (i.e. *utility*) for those services may change over time. Therefore, we also need to model the system's services, users, and users' QoS preferences. As illustrated in Figure 1, 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 function *nParam* that maps each parameter to a value.
4. A set *I* of logical interaction links between software components in the distributed system, with the associated parameters (e.g., frequency of component 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 (e.g., "find the best route to the disaster area").
6. A set *DepSpace* of all possible deployment mappings.
7. A set *Q* of QoS dimensions, and a function *qValue* that quantifies a dimension (e.g., security) for a given service in the current deployment. 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. Relative importance of different users is determined by two threshold values: *MinRate* and *MaxUtil*.
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.
10. Using the *loc* function, deployment of any component can be restricted to a subset of hosts. Using the *colloc* function, constraints on collocating components can be specified.

Note that some elements of the 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

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. $\forall c1 \in C \quad \forall c2 \in C \quad if \ (colloc(c1, c2) = 1) \Rightarrow (H_{c1} = H_{c2})$
   $if \ (colloc(c1, c2) = -1) \Rightarrow (H_{c1} \neq H_{c_i})$
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.

distributed application scenarios. We leverage a tool described below to specify these loosely defined elements of the model.

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 of our analysis support, discussed below, is to find a (new) deployment architecture that maximizes *overallUtil* and meets all of the specified constraints.

DeSi [12] is an environment that supports specification, manipulation, and visualization of deployment architectures for large-scale, highly distributed systems. DeSi's modeling capabilities are formally described in Figure 1. Its screenshot is shown in Figure 3a. An architect is able to enter desired system parameters into DeSi's model, and to manipulate those parameters and study their effects. For example, the architect is able to specify architectural elements (e.g., components, hosts), parameters (e.g., network bandwidth, host memory), and parameter values. The architect may also specify constraints (e.g., denoting a subset of components that may not be collocated on the same host). DeSi also graphically displays the system's monitored data, deployment architecture, and analysis results.

2.2. Architecture Analysis

The problem of determining and maintaining a good deployment of a software system on a set of mobile hosts is an instance of multi-dimensional optimization problems, characterized by many QoS dimensions, users and user preferences, and constraints that influence the objective. Our goal has been to devise reusable algorithms that provide highly accurate results across application scenarios. An in-depth study of the generally applicable strategies resulted in four algorithms, where each algorithm is suitable to a particular class of systems or mobility scenarios. This allows the architect to run the algorithm that is most appropriate in the given context.

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 [15]) are best characterized as generic techniques for dealing with multi-dimensional optimization problems. These techniques are accompanied by widely used algorithms and solvers. We tailored 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. Below we provide a brief discussion of all four techniques, with an analysis of their algorithmic complexity as well as their inherent trade-offs. A more detailed description of the algorithms can be found in [8].

MINLP: 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 (e.g., the combined required component memory cannot exceed the available memory on a host). Finally, we need to define the objective function (e.g., maximize availability, minimize latency). Unfortunately, there is no known algorithm for solving a MINLP problem optimally [15]. Furthermore, for problems with non-convex functions (such as ours), MINLP solvers are not guaranteed to find and converge on an approximate solution [15]. For these reasons, we needed to investigate other algorithms.

MIP: A well-known technique for transforming the MINLP problems into MIP is to add new "auxiliary" variables [15]. This transformation significantly increases the complexity of the original problem. While MIP problems can be solved optimally in principle, doing so is computationally expensive even for small problems. By leveraging appropriate heuristics, it is possible to cut down the search space, but the MIP algorithm still remains computationally very expensive. It may 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. 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.

Greedy: 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 algo-

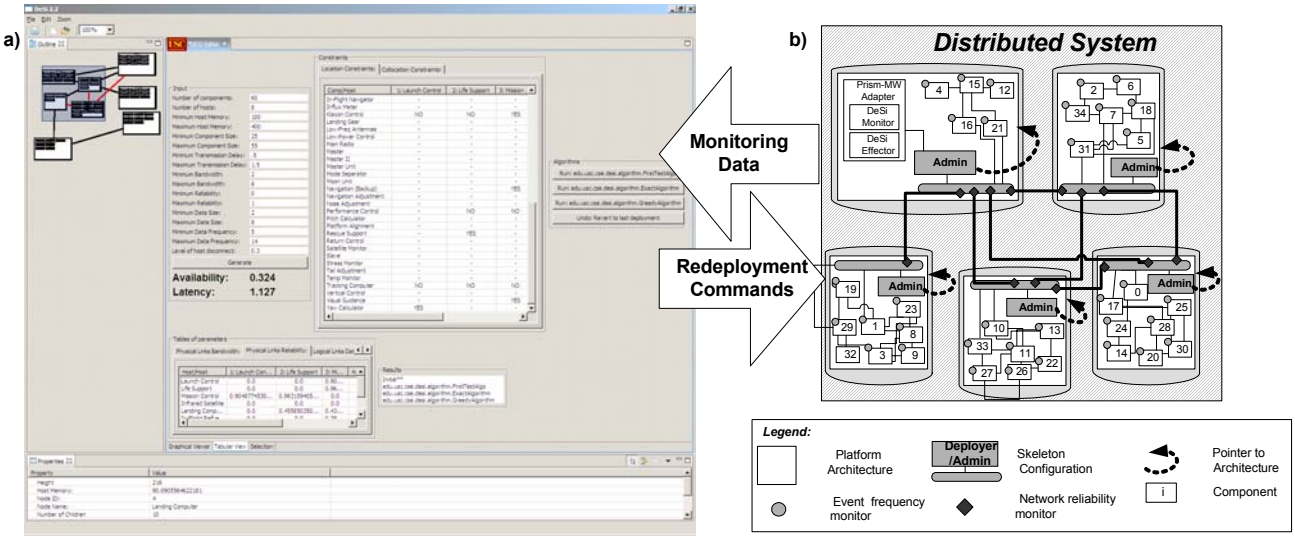

Figure 3. a) DeSi's screenshot. b) a software system running on top of Prism-MW that is being monitored and redeployed.

rithm 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 a 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 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. 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.

Genetic: Another approximative solution to our problem is based on a class of stochastic approaches called genetic algorithms. Genetic algorithms can execute in parallel on multiple processors with no overhead. 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 terminated by a triggering condition or the global optimum has been found. 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 a system's software components to hosts. Mutating an individual corresponds to changing the deployment of a few components in a given system. To evolve populations of individuals, we define a fitness function that evaluates the quality of each new individual. This function returns zero if the individual does not satisfy 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.

2.3. Architecture Implementation

We support the implementation of a software architecture via an architectural middleware platform, called Prism-MW [9]. Prism-MW provides classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. These abstractions enable direct mapping between an architecture and its implementation. Figure 4 shows the class design view of Prism-MW. We will now briefly describe this design.

Brick is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection.

A distributed application is implemented as a set of interacting *Architecture* objects. *Event*s are used to capture communication in an architecture and are exchanged via *Port*s.

*Component*s perform computations in an architecture and maintain their own internal state. Each component can have an arbitrary number of attached ports. When a component generates an event, it places copies of that event on its appropriate ports. Components may interact either directly (through ports) or via connectors. *Connector*s are used to control the routing of events among their attached components. Like components, each connector can have an arbitrary number of attached ports. Components are attached to connectors by creating a link between a component port and a single connector port. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast).

In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime. This property of components and connectors, coupled with event-based interaction, provides the underpinning of our support for runtime system mobility.

2.4. Architecture-Based System Deployment

The outcome of the architectural analysis step is a suggested deployment architecture (i.e., mapping of software components to hardware hosts) that needs to be effected on top of the implementation platform. Prism-MW allows components to exchange *ExtensibleEvents*, which may contain computational elements (components and connectors) as opposed to data. Additionally, *ExtensibleEvents* implement the *Serializable* interface (as shown in Figure 4), thus allowing their dispatching across address spaces.

In order to deploy the desired set of architectural elements onto a set of target hosts, we assume that a skeleton configuration is preloaded on each host (Figure 3b). The skeleton configuration consists of an *Architecture* object that contains an *Admin* component with a *DistributionEnabledPort* (i.e., an *ExtensiblePort* with the appropriate implementation of *AbstractDistribution* installed on it) attached to it. An *Admin* is an *ExtensibleComponent* with the *Admin* implementation of *AbstractDeployment* installed on it (see Figure 4). Since the *Admin* on each device contains a pointer to its *Architecture* object, it is able to effect runtime changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. *Admins* are able to send and

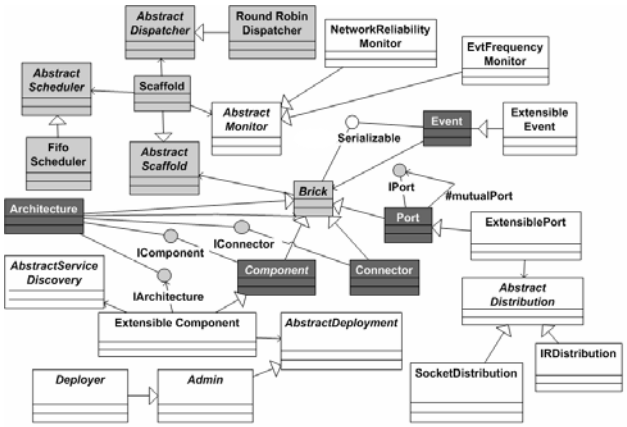

Figure 4. UML class diagram of Prism-MW's design.

receive from any device to which they are connected the *ExtensibleEvent*s that contain application components.

2.5. Mobility

As discussed previously, the nature of modern distributed systems mandates frequent changes in the system parameters and possibly QoS. For these reasons, the system's components may need to be mobile throughout the system's execution. Prism-MW has facilities to support both stateless and stateful mobility of components. The process of stateless migration can be described as follows. The sending *Admin* packages the migrant element into an *ExtensibleEvent*: one parameter in the event is the compiled image of the migrant element itself (e.g., a collection of Java class files); another parameter denotes the intended location of the migrant element in the destination subsystem's configuration. The *Admin* then sends this event to its *DistributionEnabledPort*, which forwards the event to the attached remote *DistributionEnabledPort*s. Each receiving *DistributionEnabledPort* delivers the event to its attached *Admin*, which reconstitutes functional modules (i.e., components and connectors) from the event, and invokes the *IArchitecture*'s *add* and *weld* methods to insert the modules into the local configuration.

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

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

If the serialization-like mechanism is not available, we use the event stimulus-based technique: the compiled image of the architectural element(s) to be migrated is sent across a network using the stateless technique. In addition, each event containing a migrant element is accompanied by a set of application-level events needed to bring the state of the migrant element to a desired point in its execution. Once the migrant architectural element is received at its destination, it is loaded into memory and added to the architecture, but is not attached to the running subsystem. Instead, the migrant element is stimulated by the application-level events sent with it. Any events the migrant element issues in response are not propagated, since the element is detached from the rest of the architecture. Only after the migrant architectural element is brought to the desired state is it welded and enabled to exchange events with the rest of the architecture. While less efficient than the serialization-based scheme, this is a simpler technique, it is PL-independent, and it is natively supported in Prism-MW. At the same time, the memory cost of event stimulus-based technique may be large if the numbers and sizes of events needed to update the state of a component are large.

2.6. Runtime Monitoring and Adaptation

We leverage Prism-MW as well as DeSi to support the run-time monitoring of distributed systems. DeSi provides the ability to model the system's deployment, visualize and assess its architecture, and improve it via one of the deployment improving algorithms.

We have already discussed *Admin*'s role in the deployment and adaptation of TDS. To monitor the various system properties, we leveraged Prism-MW's *AbstractMonitor* class, which is associated through the *Scaffold* with every *Brick* (shown in Figure 4). This allows for autonomous, active monitoring of a *Brick*'s runtime behavior. Once the monitoring data on each device becomes stable, the corresponding *Admin* forwards the data to DeSi where the monitored data is aggregated. Once the *Admins* determine that the monitoring data is stable, they send the data to DeSi, which populates its model. Afterwards, one of the algorithms provided by DeSi is executed for improving the system's QoS via component mobility. Finally, the result is reported back to the individual *Admins*, which coordinate the redeployment of the system between different hosts.

3. EXPERIENCE

We have applied the described framework on two application families developed with external collaborators. The first application family, TDS [9], is in the domain of mobile pervasive systems intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. The second application family is MIDAS [10], a security monitoring distributed application composed of a large number of wirelessly connected sensors, gateways, hubs, and PDAs. In both cases, the applications involved varying numbers of hosts, components, system parameters, and QoS, allowing our collaborators to apply our framework. We summarize the resulting findings about different aspects of the framework below:

- *modeling* – although different QoS were relevant in these two application families, our model was able to flexibly capture the relevant system parameters and desired objective functions.
- *analysis* – for smaller TDS and MIDAS applications MIP and MINLP performed well and gave optimal results; for larger applications, greedy and genetic algorithms were able to improve the average system deployment quality by about 30% [9].
- *implementation, deployment,* and *runtime evolution* – on average Prism-MW introduced less than 4% overhead on dynamic memory consumption and negligible performance overhead (~ 0.5%); note that the time required to effect a redeployment is a function of the number of components to be redeployed and their sizes.
- *monitoring* – our assessment of Prism-MW's monitoring support suggests that monitoring on each host induced as little as 0.1% and no greater than 10% in memory and computation overheads.

4. RELATED WORK

There are three general categories of relevant research to our work: architecture-based implementation and evolution, QoS of software architectures, and software mobility technologies. We provide an overview of the most relevant previous works from these areas below, and outline the differences between them and our approach.

Several previous works have investigated the development and evolution of a software system at the architectural level. ArchJava [1] is an extension to Java which ensures that the implementation con-

forms to architectural constraints. However, it lacks explicit support for mobility, beyond what is provided in the Java language. ArchJava also does not have any constructs to support quality assessment of different deployments, or any tools for aiding and optimizing the system's deployment. Aura [14] is an architectural style and supporting middleware for ubiquitous computing with a special focus on context awareness, and context switching. Although Aura supports component mobility and recognizes the importance of QoS in ubiquitous applications, it makes several simplifying assumptions (e.g., that the different QoS are independent from one another, which is clearly not the case for many QoS). Aura is thus only applicable to certain classes of applications in the embedded setting.

Out of techniques for assessing the properties of a software system at the architectural level, most relevant are the types of analysis dealing with the deployment and runtime aspects of a system. I5 [2] proposes the use of binary integer programming for generating a deployment of a distributed application that minimizes the overall remote communication. As such, I5 is computationally expensive and does not provide support for other QoS. I5 also does not deal with runtime reconfiguration, but assumes all the parameters for determining the optimal deployment are stable and known *a priori*. Kichkaylo et al. [6] provide a model for describing a distributed system in terms of the constraints on its deployment, and an AI planning algorithm for solving the model. This approach does not provide approximative solutions for large application scenarios, or any deployment and runtime facilities for effecting the deployment.

Finally, related to our work are the mobility technologies. XMIDDLE [11] is a data-sharing middleware for mobile computing. It allows applications to share data encoded as XML with other hosts, to have complete access to the shared data when disconnected from the network, and to reconcile data inconsistency. Lime [7] is a middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility. Lime is specifically targeted at the complexities of ad-hoc mobile environments. MobiPADS [3] is a middleware that supports active deployment of augmented services for mobile computing. It supports dynamic adaptation in order to support configuration of resources and optimize the operations of mobile applications. While Prism-MW may include features and exhibit characteristics that are similar to those provided by some of the above technologies, unlike any of them it provides native implementation facilities for software architecture-based development and adaptation in a manner that is suitable to mobile systems. Finally, unlike our framework, the above mobility technologies do not explicitly support capturing relevant parameters that affect a system's QoS, nor do they provide analysis facilities to improve system deployment.

5. CONCLUSION

While our experience with the framework has been very positive, there are several remaining challenges to be addressed. One challenge pertains to a class of mobile systems that are pervasive and long lived. These systems require a solution that is adjustable to their continuously changing execution context. This means that, while the system's architects may choose a computationally expensive redeployment strategy initially (e.g., a precise but inefficient redeployment 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, or downgraded QoS, the system will experience in order to migrate a component. In fact, a suboptimal deployment architecture may be preferable if it can be effected more quickly.

Another challenge has to do with the elicitation of users' QoS requirements, especially in an ad hoc mobile environment, where hosts may join or leave the group. For these systems, it is unlikely that architects will have access to all 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 could then be applied in different circumstances.

6. REFERENCES

[1] Aldrich, et al. ArchJava: Connecting Software Architecture to Implementation. *ICSE*, Orlando, Florida, May 2002.

[2] 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.

[3] A.Chan, S. Chuang. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Trans. on Software Engineering, Vol. 29, No.12*, December 2003.

[4] P. Ciancarini, C. Mascolo. Software Architecture and Mobility. *Int'l Workshop on Software Architecture*, Orlando, Florida, Nov. 1998.

[5] P. Clements, et al. Evaluating Software Architectures: Methods and Case Studies, *Addison Wesley*, 2002.

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

[7] LIME http://lime.sourceforge.net/

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

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

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

[11] C. Mascolo, et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Personal and Wireless Comm.*, 2002.

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

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

[14] J. P. Sousa, and D. Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *WICSA*, Montreal, Canada, 2002.

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