Self-* Software Architectures and Component Middleware in Pervasive Environments

George Edwards[1] Chiyoung Seo[1] Daniel Popescu[1] Sam Malek[2] Nenad Medvidovic[1]

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

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

ABSTRACT

Software systems that execute in embedded and pervasive environments are frequently required to be self-monitoring, self-adapting, and self-healing. However, supporting these *self-** capabilities in pervasive environments creates a number of unique engineering challenges. This paper first describes the challenges that we believe to be the most significant based on our experience developing real-world pervasive software applications with self-* capabilities. We then discuss each challenge in the context of four strategies commonly employed in self-* systems: dynamic software update, service discovery, transparent replication, and logical mobility. Finally, we explain how each strategy is implemented in our architectural middleware platform, Prism-MW.

1. INTRODUCTION

As software engineers have developed new technologies for managing the ever-increasing complexity of designing and implementing modern-day software systems, it has become apparent that there is an equally pressing need for mechanisms that automate and simplify the management and modification of distributed systems after they are deployed, *i.e.*, during run-time. For example, safety-critical embedded systems are expected to be robust in the presence of failures that may occur throughout the computing stack — that is, in the hardware, network, operating system, middleware, and application layers. This implies that such a system is self-monitoring, self-diagnosing, self-adapting, and self-healing. Systems with the capability to perform these types of transparent and autonomic dynamic adaptations during run-time in response to changes in the operating environment, non-functional requirements, or functional specification (*e.g.*, via reconfiguration and/or redeployment of software components) have been consequently termed *self-* systems* [7].

Self-* capabilities are particularly essential for pervasive systems, which are frequently embedded and mobile [15]. Pervasive systems are expected to operate in frequently changing contexts [1]. For example, a pervasive system may have many users with different tasks and goals, and the functionality of the system may be required to adjust appropriately. Similarly, a pervasive system may transition from one location to another during execution, and find that certain computing resources (software services, hardware devices, *etc.*) are no longer available. In response, the system must discover and access new resources that provide the required services. Moreover, it is highly desirable that pervasive systems remain available even while run-time adaptations are occurring; that is, software updates, evolution, and maintenance should not cause system downtime.

Not surprisingly, the most promising strategies for realizing these run-time capabilities are essentially the same as those that have been successfully applied to the most difficult and complex aspects of system construction and development: *software architectures* [3, 14] and *component middleware* [2, 13, 18]. Software architecture codifies designs and patterns that enable the flexibility and adaptability required by self-* systems so that they may be effectively reused. Furthermore, the use of explicit components and connectors furnishes the system designer with units that may be easily manipulated during run-time adaptation. Component middleware provides a complementary set of benefits. Middleware platforms can encapsulate common algorithms and protocols needed to implement the functions of self-* systems (context monitoring, service discovery, *etc.*) in such a way as to insulate application designers from this complexity and reuse proven solutions.

Although software architecture and component middleware provide a solid foundation for implementing self-management in pervasive systems, numerous research and development challenges remain. Section 2 enumerates and discusses the challenges we believe to be the most significant based on our experiences engineering self-* pervasive systems over the past decade. Section 3 proposes several candidate solutions that are the subject of our ongoing research and are being evaluated by a third-party collaborator in the context of industrial-scale pervasive applications. Section 4 concludes the paper with a summary of our insights and positions on the future of research in this area.

2. CHALLENGES

The development of software for pervasive environments poses special challenges to programmers, engineers, and architects. When self-* capabilities are demanded, these challenges become even more complex and nuanced. Through collaborations with industry partners (principally Bosch Research and Technology Center), we have been designing and implementing real-world pervasive software applications with self-* capabilities — as well as the development tools and run-time infrastructure that support those applications — over the last several years. As a result of this experience, we have identified several important engineering problems that arise at the intersection of self-* systems and pervasive systems, which are discussed in this section. Additionally, the specific relationship of software architecture and middleware to each challenge is described.

2.1. Resource Utilization

As noted in Section 1, pervasive systems are nearly always embedded and are frequently mobile. As a result, pervasive systems are generally resource-constrained, which demands a high level of efficient resource utilization. Therefore, the self-* capabilities of pervasive systems are subject to these constraints. How-

ever, many self-* functions and services (such as system health monitoring and data replication management), are relatively expensive in terms of computational resources such as processing, memory, and especially network bandwidth. Obviously, this results in a significant conflict.

To illustrate this problem more clearly, consider a pervasive application, such as a smart home, in which numerous sensors collect data about the environment and transmit it to hubs where it is logged, aggregated, and analyzed. In such as system, where a large amount of sensor data is being exchanged over a wireless network, network bandwidth becomes an extremely scarce resource. Now consider the consequences of supporting a self-* capability, namely, automatic recovery from hub failure, in this environment. First, hubs must interact to monitor each other's status and health. Second, services running on the hubs must be replicated to ensure that there is no single point of failure; again, this likely requires interaction over the wireless network to keep replicated resources synchronized. Depending on the specific application characteristics, these additional sources of network traffic overhead may be significant, and in some cases, prohibitive.

Software architecture and middleware have significant impacts on resource utilization concerns. Software architecture provides developers with abstractions that enable the construction of high-level design models that can be analyzed with respect to resource utilization early in the development cycle. Middleware services encapsulate highly efficient algorithms and design patterns that are generally superior to the *ad-hoc* solutions implemented by application programmers. On the other hand, middleware adds an additional layer of abstraction to the computing stack, which incurs additional overhead.

2.2. Real-Time Concerns

Pervasive systems must continually interact with the physical world, which implies some level of real-time requirement. The performance concerns related to pervasive systems become even more pronounced when requirements for self-* capabilities exist. First, if software components are updated or modified dynamically, the system must verify that the changes will not result in violations to real-time constraints. Second, the system cannot degrade the level of functionality or performance provided while self-adaptation is taking place. Both of these considerations may require that the system's configuration (*e.g.*, the allocation of threads or network bandwidth to components) be altered either temporarily (while updates are being effected) or semi-permanently (until another dynamic adaptation takes place).

To demonstrate this complex situation, we return to the smart-home scenario described in Subsection 2.1. To accommodate real-time requirements along with automatic failure recovery, the *active* model of replication [5] must be used. With active replication, requests to software services are transmitted to all replicas, which each process the request independently and reply to it. This allows a request which caused a failure in one or more replicas to exhibit the same end-to-end latency as a request which did not cause a failure. In contrast, in the *passive* model of replication, requests are transmitted to only a primary provider of a service, which then propagates updates — usually lazily — to replicas. When a failure occurs in the primary service provider, latency increases temporarily while a backup is promoted. However, it is easy to see that active replication incurs relatively high network overheads, (every request is transmitted to every replica and responded to), exacerbating the resource utilization problem. Similarly, with fewer replicas providing a service, we may need to increase the priority of threads executing the service in order to achieve acceptable performance.

Software architectures and middleware are commonly employed in the development of real-time systems, including pervasive, self-* systems. Architectural models are necessary to determine whether a given design will fulfill real-time requirements so that problems can be addressed before significant resources are expended on implementation. Middleware is experiencing rapid adoption in the real-time domain, especially in large-scale systems, because it provides the ability to configure and ensure many quality-of-service properties. For example, real-time middleware platforms implement management capabilities for priority-based thread allocation and message queuing.

2.3. Heterogeneity

A high degree of heterogeneity in a computing environment tends to encourage brittle system implementations. When application components must support particular protocols, target non-standard or proprietary virtual machines, and drive specialized hardware devices, they quickly lose portability, flexibility, and modifiability. Unfortunately, this is exactly the setting in which most pervasive applications operate. First, as pervasive applications attempt to integrate seamlessly into their environments, they tend to have unique user interfaces and displays [17]. Second, pervasive applications commonly rely on networking protocols and hardware other than the ubiquitous "TCP/IP over Ethernet". For example, they may use infrared networking or the Controller Area Network (CAN) protocol [4], common in the automotive domain. Aside from user interfaces and network technologies, there are numerous other examples of extreme heterogeneity in pervasive systems. Of course, the portability, flexibility, and modifiability that are difficult to achieve under these conditions are, at the same time, requisites for self-* systems.

Again we illustrate this problem through a simple example. Suppose the smart home application described above includes a set of wearable devices with control and management functionality. Additionally, assume that the system hubs are connected to the Internet, and periodically check for, download, and install software updates. The software developers can, consequently, enhance or upgrade the capabilities on one of the hubs relatively easily. However, providing access to these new features via the mobile, wearable user interface presents a major problem: the device was only designed to handle certain functionality. Thus, the ability of the system to self-upgrade and self-evolve is severely hampered.

Software architecture addresses the problem of heterogeneity by providing mechanisms for creating flexible and modifiable systems. For example, product-line architectures are composed of reusable components that can be deployed in different configurations without significant modification. In this way, platform-dependent architectural elements are encapsulated and can be substituted as required without impacting core application functionality. Middleware greatly simplifies software development for heterogeneous environments by creating an abstraction layer between application components and low-level operating systems, network protocols, and hardware.

2.4. Scale

Pervasive systems may include a large number (potentially tens of thousands) of distributed components or subsystems. This scale of participants is especially challenging for self-* systems because (1) changes in the system's environment, users, and requirements may become extremely frequent (even continual), (2) self-adaptation may be required in many components or subsystems simultaneously, and (3) adaptations may take relatively longer to complete. Self-monitoring of the overall system state, configuration, and architecture becomes hard in the presence of this high level of dynamism. Without accurate self-monitoring, many other self-* capabilities (such as self-tuning and self-healing) are difficult or impossible. Additionally, pervasive systems may vary widely in the size and complexity of their individual subcomponents, which impacts many self-* activities. For example, if components may migrate from one host to another, the costs associated with the transfer will be greater for a large component than a small component. Similarly, a complex component is likely to include far more state information than a relatively simple component; dynamically updating the configuration or maintaining synchronization between replicas will therefore also incur a higher cost.

In a scenario in which thousands of smart homes in a city communicate with a security monitoring system, self-* capabilities may be limited due to the number of participants. Some smart homes could execute a self-adaptation that impacts thousands of other smart homes, which in turn initiate their own self-adaptation. If not managed carefully, a pattern in which smart-homes are continually adapting to the changes of other participants could develop. Hence, solutions for scalable self-* procedures are needed.

Software architectures aid the development of large-scale systems by defining patterns of composition and interaction, or architectural styles, that scale well. Architectural styles constrain the ways that components can interact with each other and the runtime infrastructure in order to promote scalability. For example, peer-to-peer architectures have been shown to scale very well in practice. Middleware is important for large-scale systems because it simplifies application configuration management and provides functionality, such as event filtering in publish-subscribe services [6], that reduces overheads.

2.5. Decentralization

Many of the envisioned benefits of pervasive systems result from an expectation of decentralization. The software components of pervasive systems, rather than being organized and controlled in a top-down manner, will discover and interact with external services and resources in an *ad-hoc* fashion. However, this has important consequences for self-* systems. In a decentralized setting, individual components generally do not have access to a global view or representation of the architecture or configuration of the system. This makes it extremely difficult to predict how adaptations in one part of the system could impact the functionality or quality-of-service (QoS) observed in other parts of the system. In other words, without a single controlling entity, the coordination of adaptations among individual components or subsystems becomes a major design consideration, with cross-cutting impacts on numerous functional and non-functional properties.

In the smart home example, decentralization may become a factor if the home software wishes to interact with devices (electronics, appliances, vehicles, etc.) that are not under its management. For example, the home might wish to provide services to or access services on automobiles that are periodically located in the garage. The home does not know what vehicles may appear or what the internal software architectures of those vehicles are, and it does not have the ability or authority to adapt or modify the vehicle software. Therefore, any self-adaptation performed by either the smart home or the vehicle could have unforeseen consequences on their ability to effectively utilize each other's resources.

Decentralization poses problems for the standard approach to software architecture. Constructing analyzable and intuitive models of these highly dynamic and unpredictable systems is still an area of active research. Middleware, on the other hand, is already a crucial element in decentralized systems because it provides many required services, such as transparent resource discovery.

2.6. Safety-Criticality

Pervasive systems have the potential to become increasingly safety-critical in the future. However, safety-criticality can pose a problem for self-* systems because adaptation and evolution can result in an unsafe state. For example, components that are deemed to be safety-critical are subjected to specific, rigorous standards during coding, testing, and integration so that they may be certified by safety authorities. Also, safety-critical systems cannot experience downtime during self-adaptation. Thus, the update of these components during run-time will likely be precluded. On the other hand, dynamic adaptation can also be utilized to improve the dependability of safety-critical pervasive systems. For example, a system could dynamically suspend non-critical services in order to preserve resources for critical services when some resources have failed.

A smart home application is likely to have at least some safety-critical functions. The components that implement these functions, such as fire detection, must be either insulated from dynamic adaptations that take place elsewhere in the system, or the dynamic adaptation must be carefully constrained and limited. For example, if the fire alarm system relies on certain network links and expects a minimum bandwidth allocation and access priority, modifications to any other components utilizing those links must be ensured to not cause oversaturation of the bandwidth or otherwise preempt the fire alarm system.

Software architectures play in an important role in the development of safety-critical systems. Architectural models are often employed during safety certification activities to ensure that non-safety-critical components and subsystems cannot interfere with the function of safety-critical components. Furthermore, architectures provide the basis for model-checking tools that verify that the system cannot reach an unsafe state. Middleware, however, is still used only selectively in safety-critical systems. Most widely-used middleware platforms are not certified for safety (*e.g.*, they were not developed using the stringent coding practices required by safety standards).

3. PROMISING SOLUTION: DYNAMIC ADAPTATION OF ARCHITECTURAL MIDDLEWARE

While standard approaches to software architecture and middleware each address certain aspects of the challenges outlined in Section 2, these approaches must be refined and enhanced in order

to be applicable to self-* systems. This section discusses how we address the challenges within four capabilities — dynamic software update, service discovery, transparent replication, and logical mobility — that are commonly employed in self-* systems. Table 1 shows which challenges primarily impact each self-* capability. For each capability, we first describe its relationship to the challenges listed in the table. We then explain how each capability is supported by leveraging our lightweight middleware platform, Prism-MW [8], which enables architecture-based development of distributed applications in pervasive environments.

Table 1: Addressed Challenges

Capability	Challenge
Dynamic Update	• Real-Time Concerns • Heterogeneity • Scale
Service Discovery	• Decentralization • Scale
Transparent Replication	• Real-Time Concerns • Safety-Criticality • Resource Utilization
Logical Mobility	• Real-Time Concerns • Safety-Criticality • Heterogeneity

3.1. Dynamic Software Update

Since pervasive software systems may need to modify their functionality or quality-of-service characteristics due to changing user requirements or operating environments, they often rely on a dynamic and transparent software update mechanism (*e.g.*, replacing an existing component with a newer version). However, performing software updates at run-time can violate real-time constraints if it degrades the system's functionality or performance even temporarily. In addition, there are many heterogeneous hardware platforms in pervasive domains, some of which do not have convenient I/O interfaces (*e.g.*, CD-ROM, monitor, keyboard, *etc.*) that can be used for updating software. Finally, since pervasive systems may consist of a large number of components distributed over a network, they need an efficient and scalable way of coordinating and controlling software updates.

Unlike many traditional middleware platforms, which support software upgrade at the granularity of executable software images or patches, the software update mechanism in Prism-MW operates at the level of architectural components and connectors. Figure 1 depicts our overall approach to supporting the initial deployment and run-time update of software components. DeSi [9], our interactive deployment and analysis environment, provides the ability to model the system's deployment architecture (*i.e.*, allocation of the system's software components on its hardware hosts), and visualize and assess the architecture. DeSi also allows an engineer to replace the current version of a component with a new version, add a new component to the system, or remove an existing component from the system at run-time. An *Admin* component is a middleware-level component that is responsible for instantiation, addition, upgrade, and removal of components and connectors by interacting with DeSi. For example, if an engineer replaces the current version of a component, *Comp A* (shown in Figure 1) with a newer version using DeSi's management interface, DeSi will then transmit the new component implementation to the *Admin* component running on Host 1. The *Admin* then transfers the old component's state information to the new version and replaces the old component with the new version only when the old component is in an idle state. Similarly, if the *Admin* receives a command from DeSi to remove one of its local components, it waits until the component is idle to perform the operation.

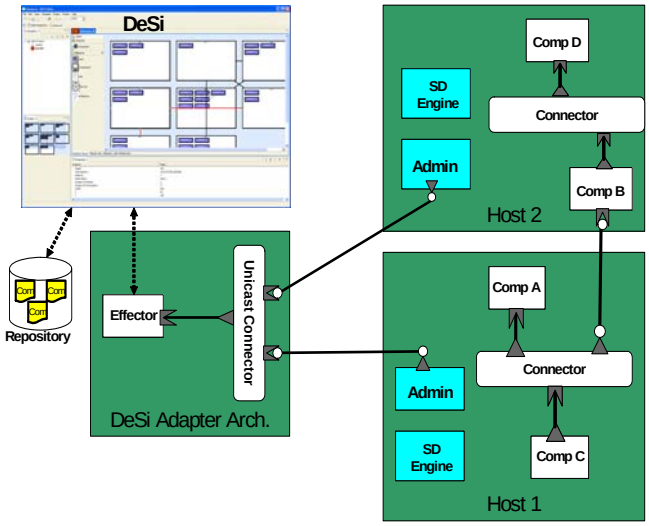

Figure 1. Dynamic software update support at runtime.

Since Prism-MW updates a component only when it is idle, it does not incur any degradation of system functionality or require shutdown. This is usually sufficient to satisfy soft-real time constraints; if hard real-time constraints are present, replication must be used along with dynamic update to ensure that quality-of-service is not compromised. Because DeSi distributes software updates over the network, our solution can be employed with pervasive devices that do not have standard I/O interfaces. DeSi automates most of the update process, allowing it to efficiently support the update of a large-scale pervasive system.

3.2. Service Discovery

Service discovery in embedded and pervasive environments is required to support self-* capabilities because (1) a service provider's location may not be known at design-time; (2) a service provider may become unavailable due to hardware, software, or network failures; and (3) the location of a service provider may change at run-time (*i.e.*, via physical or logical mobility). In this domain, discovery of external services at run-time must be supported in a decentralized manner — service lookup should not be performed and managed by a single central server, but instead facilitated via interaction with peer hosts. In addition, since there might be a large number of services in a large-scale software system, service discovery must be performed efficiently.

To support dynamic service discovery, Prism-MW provides a middleware-level component called *SDEngine* (shown in Figure 2). A *SDEngine* on each host maintains a database about all the services provided by the components running on that host, and interacts with other *SDEngines* to process service lookup requests. Prism-MW also provides a middleware-level connector, called *SDConnector* (shown in Figure 2), which is responsible for routing service requests to the appropriate service provider. Service lookup

and invocation is performed by the interactions between *SDConnector*s and *SDEngine*s.

Figure 2 shows a small fragment of a smart home application that illustrates Prism-MW's service discovery support. In this example, the Global Logging Service (GLS) component running on Host 2 records and maintains messages that client components have received from sensors. For example, suppose Client 3 on Host 3 receives an event from a sensor and wants to record the event in persistent storage via the GLS component running on Host 2. Client 3 sends an event that includes the data to be logged to its local *SDConnector*. If the *SDConnector* does not have any location information about the GLS in its routing table, it sends a lookup request to its local *SDEngine*, which then interacts with other *SDEngine*s to determine the GLS location. Once Host 3's *SDEngine* receives the location information, it connects the local *SDConnector* to Host 2's *SDConnector* as shown in Figure 2. Host 3's *SDConnector* then sends the request from Client 3 to the GLS via the connection with Host 2's *SDConnector*. Future invocations of the GLS by Client 3 (or other clients on Host 3) do not require the above lookup steps. For interested readers, a more detailed explanation of Prism-MW's service discovery mechanism is given in [16].

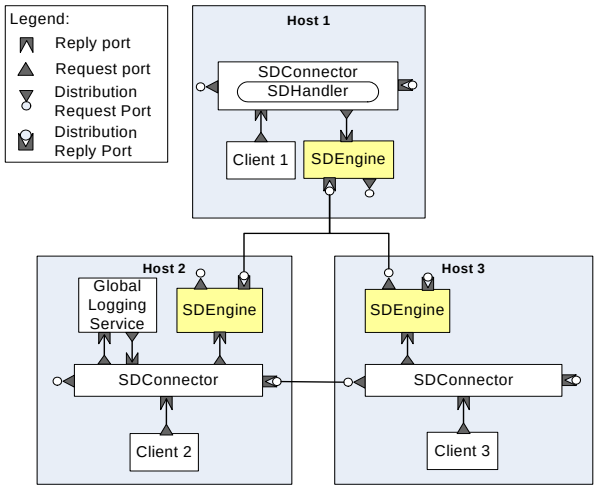

Figure 2. Dynamic service discovery support at runtime.

As described above, our solution processes service lookup requests in an *ad-hoc* and decentralized manner via *SDEngines*, instead of requiring a single central server to manage service location information. Moreover, our approach scales well because the whole system's overhead due to dynamic service discovery is distributed and balanced among the system's constituent hosts.

3.3. Transparent Replication

In pervasive environments, the services provided by a component can become unavailable due to host or network failures, lack of resources (*e.g.*, battery power), or software errors. Therefore, to provide continuous services, support for failover facilities through the replication of service providers is necessary. However, the replication of software components can incur large resource-usage overheads (*e.g.*, CPU, network, and battery power) because all replicas must maintain a synchronized state. At the same time, replication is used to ensure that failures and software updates to not reduce QoS in real-time systems. Therefore, a service replication solution for pervasive environments must be efficient in resource usage while still enabling fast recovery from failure.

Prism-MW's component replication strategy is based on the *active* replication model (discussed in Section 2.2), as the target domain frequently involves real-time considerations. Figure 3 shows Prism-MW's replication synchronization and failover strategies. We developed a middleware-level connector, called *FTConnector*, on top of Prism-MW. A *FTConnector* delivers service requests to both the primary provider of a service and all backup replicas. Each *FTConnector* is placed between a service provider (*e.g.*, the GLS component on Host 2) and its local *SDConnector* as shown in Figure 3. *SDEngine*s periodically exchange heartbeat messages; each *SDEngine* checks these heartbeat messages to detect when one of its neighbors has experienced a failure. When a *SDEngine* determines that a neighboring host has failed, if it has a backup replica for one or more primary service providers on the failed host, it promotes the backup replica to the primary role and informs all other *SDEngine*s of this promotion. For a more detailed explanation of Prism-MW's transparent replication implementation, refer to our previous work [16] on supporting fault tolerance in pervasive environments.

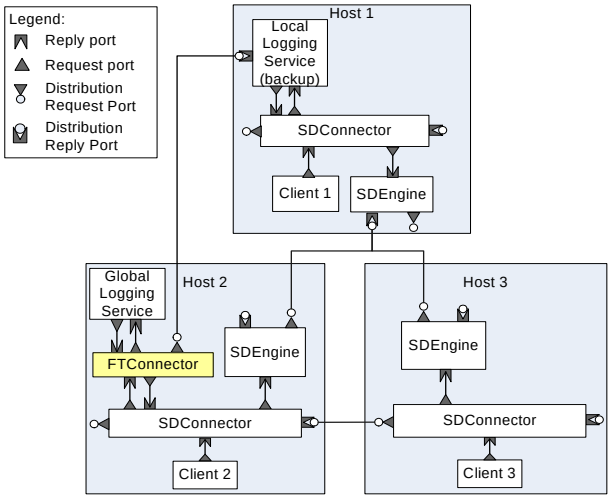

Figure 3. Fault tolerance support in the MIDAS system.

Typically, one of the main design considerations in service replication is guaranteeing that all replicas receive and process requests in the same order. Otherwise, some replicas could end up in an inconsistent state. Previous solutions [11,12] rely on reliable totally-ordered multicast protocols such as Totem [10], which can incur large overheads. In Prism-MW, on the other hand, each request event is first routed to the *FTConnector* associated with a service, and then forwarded to both a primary service provider and its backups. Therefore, all replicas receive the same sequence of request events. This approach results in reduced network and processing overhead for coordination and synchronization, which is crucial in resource-constrained environments.

3.4. Logical Mobility

Logical mobility is the capacity for software components to migrate from host-to-host after initial deployment. In some cases,

engineers cannot determine completely the non-functional properties of a software system or its target environment prior to the system's initial deployment. As a result, the system's initial deployment architecture (*i.e.*, allocation of software components to hardware hosts) may be unsatisfactory within the context of the actual running system. This is of particular concern in embedded and pervasive systems, which are affected by unpredictable movement of target hosts and fluctuations in the quality of wireless network links. Therefore, the system's deployment architecture may need to be altered by redeploying some components at runtime. However, logical mobility can violate real-time constraints if service downtime occurs during redeployment. Moreover, if the system consists of a large number of components, determining an optimal deployment architecture is computationally expensive (the computational complexity increases exponentially with the numbers of components and hosts [9]). Finally, some safety-critical components should be precluded from logical mobility.

Prism-MW supports logical mobility through the following process. Monitoring facilities implemented in each *Admin* component gather data about the run-time properties and behaviors of software components (*e.g.*, resource utilization patterns) and the computing environment (*e.g.*, network bandwidth fluctuations), and transmit this data to DeSi, which provides system visualizations and populates a deployment model. At this point, one of several efficient optimization algorithms (*e.g.*, greedy, genetic algorithms) provided by DeSi can be selected and executed to improve the system's deployment architecture. Finally, the system's new deployment architecture is transmitted back to the *Admin* components, which coordinate the redeployment by interacting with each other. DeSi also allows an engineer to specify a set of constraints on the system, which are then used as constraint inputs to the optimization algorithms provided.

As *Admin* components only perform the redeployment of a component when it is in an idle mode, our solution produces little downtime and can still satisfy soft real-time constraints. By using efficient approximation algorithms for determining the system's new deployment architecture at runtime, our solution still exhibits good performance for large-scale systems (although it cannot guarantee that it will find an *optimal* deployment architecture [9]). Using system constraints to specify a set of components that are precluded from redeployment easily prohibits the dynamic adaptation of safety-critical components.

4. CONCLUSIONS

This paper described five significant challenges that arise in supporting self-* capabilities in pervasive environments. This paper also demonstrated how a middleware platform can provide four self-* capabilities — dynamic software update, service discovery, transparent replication, and logical mobility — in a pervasive environment. All four capabilities rely on the use of explicit architectural elements in the middleware platform. For example, software components serve as the units on which dynamic update and logical mobility operations execute. Similarly, explicit software connectors are an integral part of our service discovery and transparent replication solutions. Consequently, we believe that software architecture represents a promising approach to addressing the challenges of pervasive systems, and will continue to be a focus of ongoing research in this domain.

5. REFERENCES

[1] Abowd, G.D. Software Engineering Issues for Ubiquitous Computing. In *Proc. of the 21st Intl. Conference on Software Engineering (ICSE '99)*, Los Angeles, CA, 1999.

[2] Castaldi, M., et al. A Lightweight Infrastructure for Reconfiguring Applications. In *Proc. of the 11th Intl. Workshop on Software Configuration Management*, Berlin, 2003.

[3] Cheng, S., et al. Software Architecture-based Adaptation for Pervasive Systems. In *Proc. of the Intl. Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing (ARCS '02)*, April 2002.

[4] Controller Area Network (CAN), http://www.can-cia.org/.

[5] Coulouris, G., Dollimore, J. and Kindberg, T. *Distributed Systems: Concepts and Design*. Addison-Wesley, 2001.

[6] Edwards, G., et al. Integrating Publisher/Subscriber Services in Component Middleware for Distributed Real-time and Embedded Systems. In *Proc. of the 42nd Annual ACM Southeast Conference*, April 2004.

[7] Kramer, J. and Magee, J. Self-Managed Systems: an Architectural Challenge. *2007 Future of Software Engineering*. IEEE Computer Society, 2007.

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

[9] Mikic-Rakic, M., et al. A Tailorable Environment for Assessing the Quality of Deployment Architectures in Highly Distributed Settings. *2nd Int'l. Working Conf. on Component Deployment (CD 2004)*, Edinburgh, Scotland, May 2004.

[10] Moser, L. E., et al. Totem: A fault-tolerant multicast group communication system. *Comms. of the ACM*. April 1996.

[11] Narasimhan, P., et al. State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects. *DSN 2001*, July 2001.

[12] Narasimhan, P., et al., Eternal-A Component-based Framework for Transparent Fault-Tolerant CORBA. *Software Practice and Experience*, Vol. 32, pp. 771-788, 2002.

[13] Norris, B., et al. Middleware for Dynamic Adaptation of Component Applications. In *Proceedings of the IFIP WoCo9 Conference*, July 17-21, 2006.

[14] Oreizy, P., et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, May/June '99.

[15] Satyanarayanan, M. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, August 2001.

[16] Seo, C., et al. Exploring the Role of Software Architecture in Dynamic and Fault Tolerant Pervasive Systems. In *Proc. of the Workshop on Software Engineering of Pervasive Computing Applications, Systems and Environments (SEPCASE)*, MN, May 2007.

[17] Smailagic, A. An evaluation of audio-centric CMU wearable computers. *Mobile Networks and Applications, vol. 4, no. 1*, 1999.

[18] Wang, N., et al. Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications. *IEEE Distributed Systems Online*, July, 2001.

[19] Weiser, M. The Computer for the 21st Century. *Scientific American*, September 1991.