Providing Middleware-Level Facilities to Support Architecture-Based Development of Software Systems in Pervasive Environments

Sam Malek[1] Chiyoung Seo[1] Sharmila Ravula[2] Brad Petrus[2] Nenad Medvidovic[1]

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

[2]Bosch Research and Technology Center
4009 Miranda Avenue
Palo Alto, CA, 94304
{sharmila.ravula, brad.petrus}@us.bosch.com

ABSTRACT

Software architecture has been widely advocated as an effective abstraction for modeling, implementing, and evolving complex software systems such as those in distributed, decentralized, heterogeneous, mobile, and pervasive environments. Typically, however, architectural abstractions have not been supported directly at the level of system implementation. Instead, even developers with access to state-of-the-art middleware facilities have had to rely on constructs that are at least in part different from those used in the design of their systems. In this paper we argue that it is possible to provide native and flexible software architectural facilities in a middleware platform geared to pervasive environments. We refer to such a platform as "architectural middleware". In support of our argument, we outline the design, implementation, and our experience with a specific architectural middleware platform, which has been used in solving pervasive computing problems in the classroom as well as two industrial domains. We also demonstrate that middleware-level architectural support can be effective, efficient, scalable, and adaptable.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: Domain-specific architectures, and Languages

Keywords

Prism-MW, Software Architecture, Pervasive Computing

1. INTRODUCTION

Over the past few decades we have witnessed an unrelenting pattern of growth in size and complexity of software systems, which will likely continue well into the foreseeable future. This pattern is further evident in an emerging class of embedded and pervasive software systems that are growing in popularity due to increase in the speed and capacity of hardware, decrease in its cost, emergence of wireless ad hoc networks, proliferation of sensors and handheld computing devices, etc. Studies have shown that a promising approach to resolve the challenges of developing large-scale software systems is to employ the principles of software architectures [1][9]. *Software Architectures* provide abstractions for representing the structure, behavior, and key properties of a software system [6][7]. They are described in terms of software *component*s (computational elements), *connector*s (interaction elements), and their *configurations*. Software *architectural styles* (e.g., publish-subscribe, peer-to-peer, pipe-and-filter, client-server) further refine the vocabulary of component and connector types and propose a set of constraints on how instances of those types may be combined in a system.

For software architectural models to be truly useful in a development setting, they must be accompanied by support for their implementation [4][8]. This is particularly important in the context of pervasive systems: they are often complex, highly distributed, decentralized, heterogeneous, mobile, and long-lived, increasing the risk of architectural drift [6] unless there is a clear relationship between the architecture and its implementation. This suggests that state-of-the-art middleware solutions (e.g., CORBA Orbix, TAO) that lack the implementation-level facilities for key elements of software architecture (e.g., explicit support for software connectors or architectural styles) are not necessarily the best candidates for architecture-based software development.

This paper describes our position, which has emerged from close to ten years of experience with embedded and pervasive environments: we argue that an *architectural middleware*—a middleware platform that provides native implementation-level support for the key architectural abstractions—is better suited than traditional middleware platforms to address the software engineering challenges inherent in developing pervasive systems. In support of this argument we present the design and implementation of Prism-MW, an architectural middleware geared to distributed, mobile, and pervasive environments. We have directly leveraged Prism-MW's architectural focus to provide a number of facilities required in ad hoc and pervasive systems: deployment of software across heterogeneous devices, discovery of resources, runtime analysis, dynamic adaptation and redeployment of software, fault tolerance, etc. While a number of other existing middleware platforms also provide some of these facilities [3], none provide them *both* at the architectural level and in a manner that satisfies the challenges imposed by pervasive environments. Thus far, our experience with Prism-MW in the classroom as well as two industrial domains has been very positive. However, a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MPAC '06, November 27-December 1, 2006 Melbourne, Australia
Copyright 2006 ACM 1-59593-421-9/06/11... $5.00

number of remaining issues remain unresolved, and we discuss them in this paper.

The paper is organized as follows. Section 2 discusses the architectural challenges and requirements that should be addressed in developing and deploying applications in pervasive settings. Section 3 presents an overview of Prism-MW's design. Section 4 discusses the extent to which Prism-MW addresses these requirements in the context of our experiences of applying the middleware on two families of applications representative of pervasive domains. The paper concludes with an overview of several future directions.

2. Architectural Implications

Architecting software systems for pervasive and ad-hoc domains poses significant challenges, as the software engineer typically does not know the exact characteristics of the runtime environments in which the software will be deployed. Below we discuss some of the main challenges faced by engineers while developing applications in these domains, and the corresponding middleware requirements for addressing them:

- **Efficiency.** Implementing a software system in the pervasive computing domain is difficult mainly due to its resource-constrained nature (i.e., limited CPU, memory, battery, network bandwidth, etc.). The existing approaches for developing applications on the traditional desktop platforms are often inefficient for these domains. Hence, a middleware geared to this domain should not only provide the appropriate facilities for managing system resources, but it should itself also be very efficient.
- **Analysis.** The analysis of a software system with respect to its properties (e.g., reliability, latency,) at design-time requires the prediction of the system's runtime behavior. However, since a pervasive system's runtime aspects cannot always be accurately predicted by the engineer, design-time analysis may be inaccurate. More precise analysis may only possible at runtime. For supporting runtime analysis, monitoring facilities should be provided by a middleware. Furthermore, since the monitoring activity also uses system resources, it should be both lightweight and adjustable, such that once sufficient runtime data is gathered, the monitoring overhead can be eliminated. In addition, a middleware should be flexible for supporting various dynamic system adaptations (e.g., addition and removal of software components, reconfiguration of system's software architecture) based on the result of runtime analysis.
- **Heterogeneity.** One of the major characteristics of pervasive and ad-hoc environments is the heterogeneity of the hardware platforms and the underlying system software. Heterogeneity of the target platforms impacts most phases of software development: the properties of target hardware devices should be represented during the modeling phase; different versions of the same functionality would need to be developed for and tested on different platforms during the construction phase; and software's platform requirements would need to be considered during the deployment phase. Therefore, a middleware should support an efficient mechanism that abstracts the heterogeneity of target hardware platforms away from the software engineer. This in turn aids software architectural adaptations, as it allows the engineer to focus on higher-level implications of adapting the system. For example, the engineer would only need to determine whether a software component should be migrated from one device to another, and not the low-level system-specific commands for effecting the actual migration.
- **Deployment.** The deployment of a software system onto target platforms is a challenging task. Ideally, a central deployment server would need to communicate with a large number of heterogeneous hosts, coordinate the deployment process with them, and ensure that a software system is deployed and configured correctly. This picture clearly becomes a lot more complex in a decentralized setting, where multiple hosts may play the role of the deployment server. Thus, a middleware running on each host should have a facility that can connect to a deployment server, receive its corresponding components from the server, and install and configure the components according to the overall system's architecture. In turn, the servers themselves would (coordinate to) ensure the architectural integrity [6] of the system.
- **Resource Discovery.** Application software running on one host may not be able to complete its task if some of the resources it requires reside on another host that is unreachable. The most common example of a resource in software architecture-based development is a *service*. A software component usually provides a number of services, which are typically made available to other components via the component's public *interface*s. Hence, a middleware should support a service discovery mechanism that enables a component to register its services, as well as to find, bind, and invoke the services provided by other components.
- **Suitable Architectural Styles.** No previous studies have suggested the appropriate architectural style for applications in pervasive and ad-hoc environments. It is in fact unlikely that a single architectural style would be suitable for every pervasive system. Choosing an appropriate style depends on many factors, such as the interaction patterns among the application's components, or the target environment. In fact, often the engineer will either create a new style (e.g., a hybrid style) or design each part of the system's architecture according to a different style. Thus, a middleware should provide explicit support for multiple architectural styles even within a single application.
- **Fault Tolerance.** Many factors may impact a pervasive system's ability to function correctly. For example, since the computing platforms (e.g., PDA, cellular phones, wearable devices, etc.) that are widely used in pervasive and ad-hoc environments have finite battery lives, a host may "go down" due to battery depletion. Consequently, all the services provided by the components on that host become unavailable to other hosts. To address this issue, a lightweight mechanism for migrating all the services (i.e., components) provided by the "dead" host to its neighbors transparently and in a manner that preserves architectural properties should be supported at the middleware level.

3. Prism-MW

We have attempted to address the above requirements in an architectural middleware platform called Prism-MW. Our experience has shown that an architectural middleware for pervasive computing is composed of three distinct layers which are deployed on top of an OS (shown in Figure 1): at the bottom is a *virtual machine layer* that allows the middleware to be deployed on heterogeneous platforms efficiently; the abstraction facilities provided by the virtual machine are leveraged by the middleware's *architectural constructs* that lay on top of it; finally, these architectural constructs are leveraged to implement various *pervasive computing facilities*. In this section, we describe Prism-MW's architectural support layer which was our primary focus

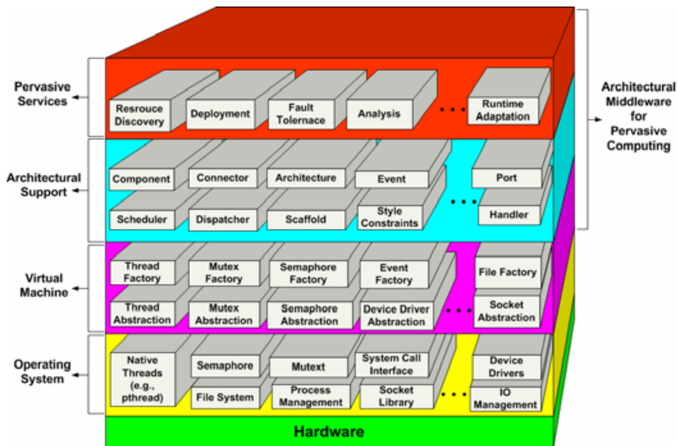

Figure 1. Layers of System Stack.

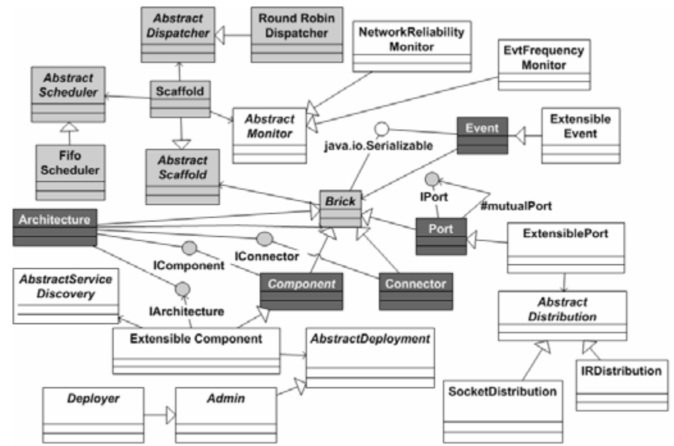

Figure 2. UML class diagram of Prism-MW's design. Middleware core classes are highlighted.

initially, while in Section 4 we discuss the other two layers of the middleware as they gained primacy in the context of our experience.

Prism-MW supports architectural abstractions by providing 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 2 shows the class design view of Prism-MW. The shaded classes constitute the middleware core, which represents a minimal subset of Prism-MW that enables implementation and execution of architectures in a single address space. Only the dark gray classes of Prism-MW's core are directly relevant to the application developer, requiring a minimal effort to master the middleware's basics.

3.1 Architectural Support

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. An event consists of a name and payload. An event's payload includes a set of typed parameters for carrying data and meta-level information (e.g., sender, type, and so on). An event type is either a request for a recipient component to perform an operation or a reply that a sender component has performed an operation.

*Port*s are the loci of interaction in an architecture. A link between two ports is made by *weld*ing them together. A port can be welded to at most one other port. Each Port has a type, which is either *request* or *reply*. An event placed on one port is forwarded to the port linked to it in the following manner: request events are forwarded from request ports to reply ports, while reply events are forwarded in the opposite direction.

*Component*s perform computations in an architecture and may maintain their own internal state. A component is dynamically associated with its application-specific functionality via a reference to the *AbstractImplementation* class. This allows us to perform dynamic changes to a component's application-specific behavior without having to replace the entire component. Each component can have an arbitrary number of attached ports. Components interact with each other by exchanging events via their ports. When a component generates an event, it places copies of that event on each of its ports whose type corresponds to the generated event type. Components may interact either directly (through ports) or via connectors.

*Connector*s are used to control the routing of events among the attached components. Like components, each connector can have an arbitrary number of attached ports. Components attach 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, has proven to be highly effective for addressing system re-configurability.

Finally, Prism-MW provides support for event dispatching, event queuing, architectural monitoring, and reflection facilities that the developer can associate with the system's architecture.

3.2 Extensibility Mechanism

The design of Prism-MW's core provides extensive separation of concerns via its explicit architectural constructs and its use of abstract classes and interfaces. The design is highly extensible. The extensible nature of Prism-MW has enabled us to directly support multiple architectural styles, even within a single application. We do not address this middleware requirement any further due to space constraints, but point the interested reader to [2].

Our support for extensibility is built around our intent to keep Prism-MW's core unchanged. To that end, the core constructs (*Component*, *Connector*, *Port*, *Event*, and *Architecture*) are sub-classed via specialized classes (*ExtensibleComponent*, *Extensible-Connector*, *ExtensiblePort*, *ExtensibleEvent*, and *ExtensibleArchitecture*), each of which has a reference to a number of abstract classes (Figure 2). Each *AbstractExtension* class can have multiple implementations, thus enabling selection of the desired functionality inside each instance of a given extensible class. If a reference to an *AbstractExtension* class is instantiated in a given extensible class instance, that instance will exhibit the behavior realized inside the implementation of that abstract class. Multiple references to abstract classes may be instantiated in a single extensible class instance. In that case, the instance will exhibit the combined behavior of the installed abstract class implementations.

4. Experience

We illustrate Prism-MW's support for the pervasive computing domain via application families developed in collaboration with

two external software development organizations. The first application family, implemented using the Java version of Prism-MW, is called Troops Deployment Simulation (TDS) and was developed in collaboration with the U.S. Army. It is representative of a large number of mobile pervasive systems that are intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. The second application family, implemented using the C++ version of Prism-MW, has been developed as part of an ongoing collaborative project between the University of Southern California and the Bosch Research and Technology Center.

4.1 Java-based Pervasive Systems

Figure 3. TDS Application.

Figure 3 shows one possible instance of the TDS application family with single *Headquarters*, four *Commander*s, and 36 *Soldiers*. A computer at *Headquarters* gathers information from the field and displays the current battlefield status: the locations of friendly and enemy troops, vehicles, and obstacles such as mine fields. The headquarters computer is networked via secure links to a set of PDAs used by *Commander*s in the field. The commander PDAs are connected directly to each other and to a large number of *Soldier* PDAs. Each commander is capable of controlling his own part of the battlefield: deploying troops, analyzing the deployment strategy, transferring troops between commanders, and so on. In case the *Headquarters* device fails, a designated *Commander* assumes the role of *Headquarters*. Soldiers can only view the segment of the battlefield in which they are located, receive direct orders from the commanders, and report their status. TDS helps to illustrate a number of concepts related to pervasive and ad hoc domains, which we will discuss below.

4.1.1 Heterogeneity

Several aspects of TDS embody the notion of multiplicity inherent in embedded pervasive environments. The devices on which TDS has been deployed are of several different types (Palm Pilot Vx and VIIx, Compaq iPAQ, HP Jornada, NEC MobilePro, Sun Ultra, PC), running four OSs (PalmOS, WindowsCE, Windows XP, and Solaris). As mentioned earlier and shown in Figure 1 the architectural support of the middleware is insulated from the underlying heterogeneity of the operating system and hardware platforms via the virtual machine layer. In the case of TDS, since it was developed on the Java version of Prism-MW, the Java Virtual Machine provided the necessary abstractions. As will be discussed below, supporting heterogeneity in lower level programming languages (e.g., C, C++) is more challenging.

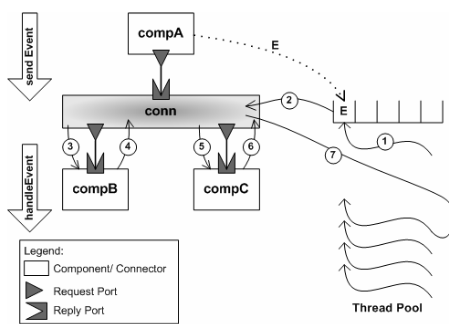

Figure 4. Efficient Event Dispatching in Prism-MW: steps 1-7 are performed by a single shepherd thread, assuming that the connector is sending event E to both recipient components.

Another source of heterogeneity that we had to overcome was the variation in communication protocols. Some of the devices were equipped with wireless network cards, allowing them to interact via TCP/IP, while the others devices were only equipped with the infrared capability. In order to address different aspects of interaction, we leveraged the extensible nature of Prism-MW's ports. The *ExtensiblePort* class has references to a number of abstract classes that support various interaction services. In turn, each abstract class can have multiple implementations. Figure 2 shows some of the many port extensions that we have implemented thus far. In the case of TDS, we leveraged the *AbstractDistribution* class that has been implemented by two concrete classes, one supporting socket-based and the other infrared port-based inter-process communication (IPC), to overcome the communication heterogeneity on the target platforms.

4.1.2 Efficiency and Scalability

A distributed system implemented in Prism-MW consists of a number of *Architecture* objects, each of which serves as a container for a single subsystem and delimits an address space. *Component*s within and across the different *Architecture* objects interact by exchanging *Event*s. Our original implementation of Prism-MW associated a separate event queue and thread with each architectural element that could send/receive events. While this design was the most intuitive and was sufficient for applications deployed on capacious desktop computers, it proved to be too expensive to be used in TDS. Therefore, we adapted Prism-MW to use a fixed-sized, circular array for storing all events in a single address space (depicted in Figure 4). This allowed us to optimize event processing by introducing a pool of shepherd threads (implemented in Prism-MW's *RoundRobinDispatcher* class) to handle events sent by any component in a given address space. The size of the thread pool and event queue are parameterized and, hence, adjustable.

By default, Prism-MW processes events asynchronously. A shepherd thread removes the event from the head of the queue. The shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread. If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. The new approach to routing events had a number of advantages: (1) By leveraging the explicit architectural topology, an event can be routed to multiple destinations. This minimizes resource consumption, since events need not be tagged with their recipients, nor do the recipients need to explicitly subscribe to events. (2) We further optimize resource consumption by using a single event queue for storing both locally and remotely generated events. (3) Since Prism-MW processes local and remote events uniformly, and all routing is accomplished via the multiple and explicit ports and/or connectors, Prism-MW allows for seamless redeployment and redistribution of existing applications onto different hardware topologies.

TDS was deployed onto 105 mobile devices and mobile device emulators running on PCs, where a total of 245 software components interact via 217 software connectors. The dynamic size of the application is approximately 1 MB for the *Headquarters* subsystem, 600 KB for each *Commander*, and 90 KB for each *Soldier* subsystem, resulting in the total application size of 12.5 MB. In this scenario, the total overhead induced by the middleware on all of the devices was measured to be around 511.5 KB, or 4%.

4.1.3 Deployment

As mentioned earlier, deploying applications in the pervasive environments is a challenging task. Prism-MW components communicate by exchanging application-level events. Prism-MW also allows components to exchange *ExtensibleEvents*, which may contain architectural elements (components and connectors) as opposed to data. Additionally, *ExtensibleEvents* implement the *Serializable* interface (as shown in Figure 2), thus allowing their dispatching across address spaces.

In order to migrate the desired set of architectural elements onto a set of target hosts, we assume that a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW's *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* Component is an *ExtensibleComponent* with the *Admin* implementation of *AbstractDeployment* installed on it (shown in Figure 2). Since the *Admin* Component 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. *Admin* Components are able to send and receive from any device to which they are connected the *ExtensibleEvent*s that contain application components and connectors.

4.2 C++ Based Pervasive Systems

In this section, we describe our experience with developing a distributed software system family, called MIDAS, on top of the C++ version of Prism-MW. MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly in the manner shown in Figure 5. The sensors, which are used to monitor the environment around them, communicate their status to one another and to the gateways. The gateway computers are responsible for managing the sensors. The gateways translate, aggregate, and fuse the data received from the sensors, and propagate the appropriate data (e.g., event) to the hubs. Hubs are used to evaluate and visualize the sensor data for the users, as well as provide an interface through which the user can send control commands to the various sensors and gateways in the system. Hubs may also be configured to propagate the appropriate sensor data to PDAs, which are then used by the mobile users of the system. This application scenario helps to illustrate a number of concepts, several of which are different from those discussed earlier.

4.2.1 Heterogeneity

As mentioned before, the abstractions provided by JVM insulated the Java version of Prism-MW from the underlying heterogeneity of the target platforms. However, since MIDAS was a C++ application, we had to resolve the heterogeneity issue at a much lower-lever than before. In turn, this required us to rethink our view of what an architectural middleware entails and of its overall architecture. This process actually resulted in the view of the architecture of Prism-MW depicted in Figure 1. Initially we set out to develop, compile, and maintain several version of Prism-MW, one per each hardware platform and operating system. However, this approach soon proved to be infeasible: as the number of different versions of Prism-MW kept growing we were faced with developing and exhaustively testing the same feature(s) repeatedly. Instead, we opted to develop a domain specific virtual machine called Modular Virtual Machine (MVM). MVM provided an abstraction layer on top of various operating systems (Linux, Windows, eCos) and hardware platforms (Intel x86, KwikByte, and several other proprietary sensor platforms). Figure 1 shows some of the resource abstractions and factories that we have developed, which are in turn leveraged by the middleware's architectural constructs. This approach proved to be more flexible and convenient, as supporting a new OS or hardware platform would require only the addition of simple abstraction facilities to the virtual machine layer. This design also allowed for a clear separation of architectural constructs from the system-level constructs (as shown in Figure 1). Also note that the design of the middleware's architectural support (shown in Figure 2) remained intact as we ported it from Java to C++. This was due to the extensive separation of concerns built into Prism-MW that allowed for a natural layering of the architectural constructs on top of the lower-level system constructs.

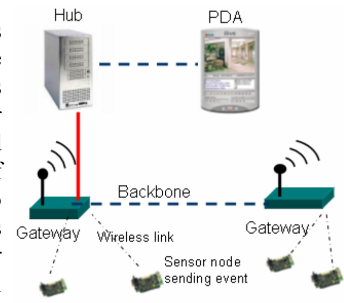

Figure 5. MIDAS System.

4.2.2 Resource Discovery and Fault Tolerance

MIDAS platforms could become unavailable for many reasons: network disconnection, hardware and software failures, and so on. Therefore, there was a need for a facility that supports recovery from such scenarios by (re)discovering the orphan sensors (i.e., sensors that have lost their connection to a gateway) or (re)discovering services that resided on a gateway. As shown in the top layer of Figure 1 and discussed below, we leveraged Prism-MW's architectural constructs to implement resource discovery. In the context of MIDAS a *service* corresponds roughly to a component *interface*. We developed an implementation of *AbstractServiceDiscovery* that provides the support for recording and retrieval of services (as shown in Figure 2). An *ExtensibleComponent* with an implementation of *AbstractServiceDiscovery* installed on it acts as a service discovery agent on the host on which it resides. The service discovery component can leverage *ExtensibleComponent*'s pointer to the architecture to determine the services installed on the local host. Service discovery components leverage *DistributionEnabledPort*s to communicate with other service discovery components (e.g., send service query or update events). Supporting service discovery via Prism-MW's architectural constructs thus provides location transparency at the level of architecture.

4.2.3 Efficiency and Scalability

In the Java version of Prism-MW, we relied on the JVM to manage the creation and removal of Java objects. While this approach incurred an overhead of creating objects dynamically in the heap at runtime, we were able to ignore this overhead in applications without strict real-time requirements such as TDS. However, MIDAS had stringent latency requirements of transmitting an alarm from a sensor to a hub and receiving an acknowledgement back in less than two seconds. Therefore, in this application, we were not able to ignore such inefficiencies. To solve this problem we enhanced MVM by developing a memory management facility based on a memory pooling technique, which pre-allocates various C++ objects (e.g., event, mutex, semaphore, etc.) from the heap when the middleware starts up. This allowed us to efficiently access the pool when an object with a particular type was required, and release it back to the pool when it was not needed any longer. We were thereby able to reduce the overhead of memory allocation to a simple pointer operation.

To insulate the architectural layer from the idiosyncrasies of the underlying memory management facility, we created a number of factory facilities that manage the (de)allocation of the architectural constructs. For example, a *Component* generates an

Event via an API exported by the event factory facility (shown in Figure 1) in the virtual machine layer, irrespective of whether the *Event* is allocated from the heap or from a memory pool. The total memory footprint of the application running on a gateway was 3.1 MB, while Prism-MW's overhead was measured to be 189 KB, or 6%.

4.2.4 Runtime Analysis and Dynamic Adaptation

As mentioned earlier, engineers may not know a priori the properties of the target hardware platforms, and early on make decisions that may not be appropriate for the actual running system. We came across this in the case of MIDAS. It became evident that different deployments of MIDAS had a significant impact on the resulting quality of service provided by the system. However, the engineers did not have sufficient knowledge of runtime properties that could be used to determine a good deployment of the system. To solve this problem we leveraged Prism-MW as well as our interactive deployment analysis environment called DeSi [5] (shown in Figure 6a). 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.

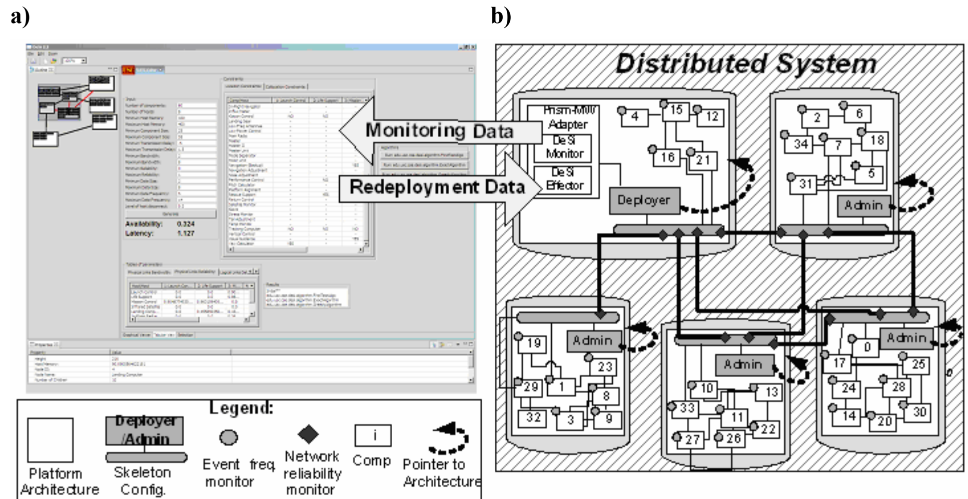

Figure 6. Runtime analysis and adaptation approach: a) DeSi's view of system, b) a system running on top of Prism-MW that is monitored and managed by meta-level components.

Figure 6b depicts an example distributed system that is monitored and deployed on top of Prism-MW. We have already discussed *Admin*'s role in the deployment and adaptation of TDS. To support deployment in the C++ version of Prism-MW, we took the same approach as that described in Section 4.1.3. 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 2). 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 a centralized *Admin*, which is called *Deployer*, for aggregating the monitored data. As shown in Figure 6b, we integrated DeSi with Prism-MW, by wrapping DeSi's *Monitor* and *Effector* components via a Prism-MW *Adapter*. Once the *Deployer* component determines that the monitoring data is stable, it sends the data to DeSi, which populates its model. Afterwards, one of the algorithms provided by DeSi is executed for improving the system's deployment. Finally, the result is reported back to the *Deployer*, which coordinates the redeployment of the system with the help of the *Admin* components.

5. Conclusion

In this paper, we presented the design of Prism-MW, an architectural middleware geared to the pervasive computing domain. We discussed our experiences in the development of two families of applications on top of Prism-MW, and provided an overview of the middleware's evolution as we adapted it to address the challenges we came across. In the process our experiences strongly suggest that it is possible to design a middleware that is applicable to the pervasive computing domain, while preserving its inherent support for architecture-based software development. We realized that for these architectural facilities to be truly useful in a highly heterogeneous and resource constrained environment, they would need to be complemented with the appropriate low-level system support. Furthermore, it became clear that to fully reap the benefits of developing a software system using the architectural facilities provided by Prism-MW, the middleware should be accompanied with advanced facilities that are targeted at the specific challenges posed by the pervasive computing domain.

While our experience has been positive, there are a number of open issues. We are developing several new advanced facilities, including "live update" of software components, fault-tolerance support by maintaining backups of data, and dynamic adaptation to decrease a system's energy consumption. We expect this to further enrich our understanding of architectural middleware as well as our appreciation of its appropriate role in pervasive environments.

6. Acknowledgement

This material is based upon work sponsored in part by the National Science Foundation under Grant number ITR-0312780 and by Bosch.

7. REFERENCES

[1] E. A. Lee. Embedded Software. *Advances in Computers (Marvin V. Zelkowitz, ed.), Academic Press*, London, 2002.

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

[3] C. Mascolo, G. P. Picco, et al. Survey of Middleware for Networked Embedded Systems. Project Report: http://www.ist-runes.org/docs/deliverables/D5_01.pdf

[4] N. Medvidovic, et al. A Family of Software Architecture Implementation Frameworks. *Working Conference on Software Architecture*, Montreal, Canada, Aug. 2002.

[5] M. Mikic-Rakic, S. Malek, 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.

[6] D. E. Perry, et al. Foundations for the Study of Software Architecture. *Software Engineering Notes*, Oct. 1992.

[7] M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.

[8] M. Shaw, et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, April 1995.

[9] J. Sousa, et al. Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Working Conf. on Software Architecture,* Montreal, Aug. 2002.