Tailoring an Architectural Middleware Platform to a Heterogeneous Embedded Environment

Sam Malek Chiyoung Seo Nenad Medvidovic

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

Abstract

Embedded systems are rapidly growing in size, complexity, distribution, and heterogeneity. As a result, the traditional practice of developing one-off embedded applications that are often rigid and unmanageable is no longer acceptable. Recent studies have suggested that an effective approach to developing software systems in this domain is to employ the principles of software architecture. However, for software architectural concepts to be truly useful in a development setting, they must be accompanied by support for their implementation and evolution. This has motivated our work over the past several years on an architectural middleware, called Prism-MW, that provides implementation-level support for the development of software systems in terms of the software architectural constructs (e.g., components, connectors). Prism-MW was initially developed in Java and used in several domains. Recently, as part of an on-going project, we were required to implement Prism-MW in ANSI C++. This experience proved to be more challenging than we initially anticipated, mainly due to the inherent heterogeneity of the computing substrate. As a result of this experience, we had to reconsider some of our earlier assumptions of what constitutes an architectural middleware and its role in the software development process. In this paper, we provide an overview of our experience and the lessons we have learned along the way.

1. Introduction

Over the past several years, our research group has conducted work in the area of *architectural middleware* [6,9,10,12]. We define architectural middleware as a middleware platform that provides implementation-level constructs for key architectural abstractions [13,15]: components, connectors, ports, events, styles, and so forth. The objective behind architectural middleware is clear: software developers have embraced these architectural abstractions as powerful design-level tools, but are typically forced by existing technologies to realize them using a different set of implementation-level tools. Thus, while engineers may prefer to think of systems in terms of high-level abstractions such as components, connectors, styles, and so on, they are often forced to implement them in terms of low-level constructs such as methods, arrays, linked lists, and so on.

A programming language (PL) or middleware platform may provide support beyond such low-level constructs, but typically they are insufficient to fully realize architecture-level decisions [6]. For illustration, consider the following examples.

- A software *component* may in fact be implemented as a middleware-level *component* (though perhaps with different properties, such as lack of support for required interfaces), or simply as a cluster of PL-level *classes*.
- A communication *port* on a component may be implemented as an explicit PL-level *interface* (again, perhaps with different properties from those intended in the architecture-level port, such as bi-directionality) or simply as an ungrouped set of *method declarations*.
- An *event* may be supported directly in a middleware as a *message*, or it may be mimicked via a *method call* or even *shared memory*.
- A *connector* may be distributed (and thus "lost") across different implementation-level modules as a combination of *method calls, shared memory, network sockets,* and other facilities.
- Finally, a given *architectural style* (e.g., publish-subscribe or peer-to-peer) may not be supported at all, but rather at best mimicked, and at worst ignored, by the style (e.g., client-server) assumed by the middleware.

Ample experience demonstrates that software engineers have clearly "made do" with the existing middleware facilities, building many successful systems. However, an argument can also be made that it is precisely this "abstraction chasm" between design and implementation that is a significant cause of *architectural erosion* [13]. Architectural erosion denotes the (frequent) situation in which a system's implementation departs from, or even invalidates, key architectural design decisions.

Our hypothesis has been that we can significantly reduce architectural erosion by bridging the abstraction chasm and, in turn, that we can bridge the chasm by providing an architectural middleware platform. As a proof-of-concept, we developed such a platform, called Prism-MW [6,9,10,12]. Prism-MW was initially implemented, evaluated, and used primarily in Java (both JVM and KVM). However, different subsets of it were also implemented in Embedded Visual C++ (EVC++), Python, C#, and Brew. The Java version in particular was evaluated extensively in the laboratory setting for a number of properties of interest, including efficiency, scalability, dynamic adaptability, and portability [6,12]. Prism-MW was used in several domains: desktop applications, handheld computing, data grids, and embedded systems [6,8,10].

Based on this experience, we felt confident in our understanding of architectural middleware and in Prism-MW's applicability to additional domains. So, when engineers from the Bosch Research and Technology Center (RTC) approached us with the idea of using our work in one of their heterogeneous, distributed embedded systems, we were optimistic that Prism-MW was the right solution. After all, we "only" had to reimplement our design in ANSI C++, and we already had the EVC++ implementation as a baseline.

Unfortunately, things did not go that smoothly. The application domain and computing environment (including the PL) to which we

were porting Prism-MW turned out to be very different from those we had faced in the past. A lot of the facilities on which we had come to rely, and which we got "for free" in Java (and even EVC++) and on the platforms we had employed (e.g., Compaq's PocketPC), were no longer present. In this new setting, we had to custom-build many of those facilities (e.g., data serialization, socket-based communication), and develop several low-level utilities (e.g., memory management, thread synchronization) to abstract away the heterogeneity of the computing substrate. Even then, porting Prism-MW posed a number of added challenges, at least in part due to the inherent trade-off between the two desired characteristics of a middleware for the embedded systems domain: supporting heterogeneous target platforms and achieving efficiency.

This experience forced us to reassess what an architectural middleware is, what facilities it must provide, and which of those facilities are "more architectural" vs. "more middleware". As a result, we have developed a much more nuanced understanding of Prism-MW, and believe that the insights we have gained can also help us in formulating a generally applicable reference architecture for architectural middleware platforms. As a side benefit, we can now better relate Prism-MW, and other similar technologies, to traditional middleware platforms, as well as to related infrastructure concepts such as software libraries and frameworks.

The remainder of the paper is organized as follows. Section 2 discusses the related work in the areas of architecture-based development and middleware for embedded systems. Section 3 provides a brief overview of Prism-MW. Section 4 presents our experience in developing a family of sensor network applications on top of Prism-MW in collaboration with Bosch, with a particular focus on the changes this imposed on the middleware itself. Finally, Section 5 discusses the lessons we have learned from this experience.

2. Related Work

At a very high-level the related literature can be classified into two categories: technologies specifically targeted at supporting implementation of software architectures, and middleware platforms for embedded systems. Below we provide an overview of the most notable solutions from both categories. We should note that the first category (architectural middleware) only encompasses two technologies, offering another demonstration of how rarely researchers have looked into this problem.

ArchJava [1] is an extension to Java that unifies software architecture with implementation, ensuring that the implementation conforms to architectural constraints. ArchJava currently has several limitations that would likely limit its applicability in the embedded computing setting: communication between ArchJava components is achieved solely via method calls; ArchJava is only applicable to applications running in a single address space; it is currently limited to Java; and its efficiency has not yet been assessed.

Aura [16] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Aura is thus only applicable to certain classes of applications in the embedded setting. Similarly to Prism-MW, Aura has explicit, first-class connectors.

Orbix/E [3] is a lightweight CORBA ORB optimized for embedded applications. It is designed for rapid development and deployment support in both C++ and Java. Orbix/E has a relatively small memory footprint, which enables its use in memory constrained applications. Orbix/E provides the ability to choose a subset of features for a given application in order to optimize its size and speed.

ACE [14] is an object-oriented framework that implements many core patterns for concurrent communication software. The patterns and components in the ACE framework have been applied in the ACE ORB (TAO), which is a CORBA-compliant middleware framework. TAO allows clients to invoke operations on distributed objects without concern for object location, PL, OS platform, communication protocols, or hardware.

JXTA [4] is a set of open protocols that allow any connected device on the network, ranging from cell phones and wireless PDAs to PCs and servers, to communicate and collaborate in a peer-to-peer manner. JXTA peers create a virtual network where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls or on different network transports. JXTA supports multiple platforms and languages, and ensures secure communication of collaborating peers.

Jini network technology [17] is an open architecture that enables developers to create network-centric hardware or software services that are highly adaptive to change. The Jini architecture specifies a way for clients and services to discover each other and to collaborate across the network. When a service joins a Jini network, it advertises itself by publishing an object that implements the service API. A client finds services by looking for an object that supports the API. When the client gets the service's published object, it will download any code it needs in order to communicate with the service, thereby learning how to "talk" to the particular service implementation via the API.

XMIDDLE [7] is a data-sharing middleware for mobile computing. XMIDDLE allows applications to share data that are encoded as XML with other hosts, to have complete access to the shared data when disconnected from the network, and, when possible, to reconcile any changes made with all the hosts sharing the data. The goal is to make sure that eventually all hosts will have a consistent version of the shared data. XMIDDLE is lightweight and fast, and caters to the frequent disconnection behavior that mobile devices exhibit. XMIDDLE also allows applications to influence the reconciliation process.

Lime [5] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Lime is specifically targeted at the complexities of ad-hoc mobile environments. The goal of Lime is to provide the simple Linda model of coordination in mobile environments via tuple spaces.

Finally, MobiPADS [2] is a reflective middleware that supports active deployment of augmented services for mobile computing. MobiPADS supports dynamic adaptation in order to provide flexible configuration of resources and optimize the operations of mobile applications. MobiPADS configurable services (called mobilets) can be augmented to address the changing conditions of a wireless environment (e.g., CPU load, network bandwidth).

While Prism-MW may include features and exhibit characteristics that are similar to those provided by some of the technologies discussed above, unlike any of them it provides native implementation facilities required for software architecture-based development in a manner that is suitable to embedded and resource-constrained systems. We discuss Prism-MW's design and implementation next.

3. Design of Prism-MW

Prism-MW supports architectural abstractions by providing middleware-level modules (e.g., classes) for representing each architectural element, with operations for creating, manipulating, and destroying the element. These abstractions enable direct mapping between an architecture and its implementation. Figure 1 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. Our goal was to keep the core compact, reflected in the fact that it contains only twelve classes (four of which are abstract) and four interfaces. Furthermore, the design of the core (and the entire middleware) is highly modular: we have tried to limit direct dependencies among the classes by using abstract classes, interfaces, and inheritance as discussed below.

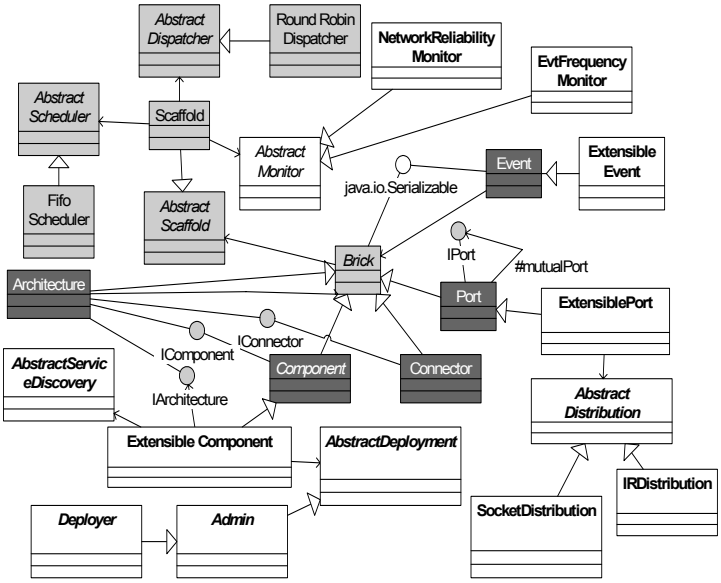

Figure 1. UML class diagram of Prism-MW. Middleware core classes are highlighted.

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, possibly at system runtime. 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 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 reconfigurability.

Each subclass of the *Brick* class has an associated interface. The *IArchitecture* interface exposes a *weld* method for attaching two ports together. The *IComponent* interface exposes *send* and *handle* methods used for exchanging events. *Component* provides the default implementation of *IComponent*'s *send* method: generated request events are placed asynchronously on all of the request ports attached to the component, while generated reply events are placed asynchronously on all of the attached reply ports. The *IConnector* interface provides a *handle* method for routing of events. The *Connector* class provides the default implementation of the *IConnector*'s *handle* method, which forwards all request events to the connector's attached request ports and all reply events to the attached reply ports. We have also provided implementations of different routing policies, including unidirectional broadcast, bidirectional broadcast, and multicast [6]. The *IPort* interface provides the *setMutualPort* method for creating a one-to-one association between two ports.

Finally, Prism-MW's core associates the *Scaffold* class with every *Brick*. *Scaffold* is used to schedule and queue events for delivery (via the *AbstractScheduler* class) and pool execution threads used for event dispatching (via the *AbstractDispatcher* class) in a decoupled manner. Prism-MW's core provides default implementations of *AbstractScheduler* and *AbstractDispatcher*: *FIFOScheduler* and *RoundRobinDispatcher*, respectively.

The novel aspect of our design is that this separation of concerns allows us to independently select the most suitable event scheduling, queuing, and dispatching policies for a given (e.g., embedded) application. Furthermore, it allows us to independently assign different scheduling, queuing, and dispatching policies to each architectural element, and possibly even change these policies at runtime. For example, a single event queue can be instantiated for the

entire architecture; alternatively, a separate event queue can be assigned to each component. Additionally, dispatching and scheduling are decoupled from the *Architecture*, allowing one to easily compose many sub-architectures (each with its own scheduling and dispatching policies) in a single application. *Scaffold* also directly aids architectural awareness (also referred to as reflection) by allowing probing of the runtime behavior of a *Brick* via different implementations of the *AbstractMonitor* class.

Prism-MW's core has been implemented in Java JVM. Subsets of the described functionality have also been implemented in Java KVM, EVC++, C#, and Python; they have been used in example applications and in evaluating Prism-MW. The implementation of the middleware core is quite small (under 900 SLOC), which aids Prism-MW's understandability and ease of use.

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 [6].

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 subclassed via specialized classes (*ExtensibleComponent*, *ExtensibleConnector*, *ExtensiblePort*, *ExtensibleEvent*, and *ExtensibleArchitecture*), each of which has a reference to a number of abstract classes (Figure 1). 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. "Embedding" Prism-MW

Note from the preceding discussion that our view of Prism-MW was "flat": our focus on its many interweaved elements did not distinguish between facilities that were clearly at different levels of abstraction (e.g., low-level mechanisms to ensure efficient delivery of data vs. high-level design mechanisms such as architectural styles). In this section, we describe our experience with developing a family of sensor network applications, called MIDAS, on top of Prism-MW in collaboration with Bosch RTC. This experience caused us to rethink and "stratify" Prism-MW's architecture. We will only briefly describe the details of MIDAS here, and instead will mostly focus on the impact of this project on our architectural middleware.

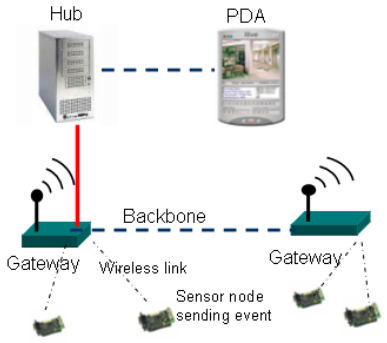

Figure 2. MIDAS system.

MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly in the manner shown in Figure 2. The sensors are used to monitor the environment around them. They communicate their status to one another and to the gateways. The gateway nodes are responsible for managing and coordinating the sensors. Furthermore, the gateways translate, aggregate, and fuse the data received from the sensors, and propagate the appropriate data (e.g., event) to the hubs. Hubs in turn are used to evaluate and visualize the sensor data for human users, as well as to 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.

To facilitate the integration of MIDAS with hardware and software that had been developed previously by Bosch, we were required to develop the portion of MIDAS that runs on the gateways and hubs in C++ and the portion that runs on the PDA in Java. C++ was also selected due to its expected efficiency and low-overhead.

In Section 3 we described Prism-MW's architectural support, which represents the centerpiece of the middleware. However, in the context of the MIDAS project we came to realize 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 several more advanced facilities. In turn, this experience has helped us to identify the scope of an architectural middleware, the services it should provide, and the relationships among those services.

Figure 3 is a graphical depiction of the three layers of an architectural middleware and their internal composition (as well as the OS which acts as the bottom-most layer). The three architectural middleware layers will be further elaborated on throughout the remainder of this paper. In particular, in this section, Subsections 4.1, 4.2, 4.3 primarily refer to the bottom layer, Subsection 4.4 to the middle layer, and Subsections 4.5, 4.6, and 4.7 to the top layer of Prism-MW's architecture.

4.1. Heterogeneity

The JVM provides many common facilities that are used to develop platform-independent code. On the other hand, in standard C++ most of these facilities are not provided. Rather, it is left to the programmer to either develop or reuse them off-the-shelf. Unfortunately, both approaches typically result in a platform-dependent application. For this reason, initially we set out to solve this issue by developing, compiling, and maintaining several versions of Prism-MW, one per each hardware platform and OS. 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 over and over again.

Instead, we opted to develop a domain-specific virtual machine called Modular Virtual Machine (MVM), as depicted in Figure 3. MVM was designed as a pluggable family of utilities that provides an abstraction layer on top of various operating systems (Linux, Windows, eCos) and hardware platforms (Intel x86, KwikByte, and several proprietary sensor platforms). MVM is composed of three parts: resource *abstractions*, *implementations*, and *factories*. Resource abstractions are managed via their corresponding factories and provide a common API that is leveraged by Prism-MW and ap-

plication developers to produce platform-independent code. A resource abstraction is realized via its implementation, which may use OS or hardware specific libraries. For a given target host the executable image of MVM is created by building the MVM source code with the appropriate implementation files included. Note that many of these abstractions correspond to facilities provided by a JVM, which are installed on a target platform as Dynamically Linked Libraries and invoked when Java native methods are executed.

To further understand the role of MVM in the "porting" of Prism-MW from Java to C++ consider the support it provides for threads (shown in Figure 3). In the Java version of Prism-MW, we relied on Java's native thread and thread synchronization mechanisms. However, in C++ software engineers typically have to use the OS's support for threads. Similarly, for thread synchronization they have to rely on the host OS semaphore or mutex libraries. Therefore, to remove this dependency on the OS, we developed thread, mutex, and semaphore abstractions and the corresponding implementations in the MVM layer. Other resource abstractions were also provided similarly.

The above approach proved to be flexible and extensible, as supporting a new OS or hardware platform would require only the development of host-specific resource implementations in MVM. This design also allowed for a clear separation of architectural constructs from the system-level constructs (as shown in Figure 3). Also note that the design of the middleware's architectural support (shown in Figure 1) remained intact as we ported it from Java to C++. This was due to the extensive separation of concern built into Prism-MW that allowed for a natural layering of the architectural constructs on top of the lower level system constructs. For example, changing the threading API at the virtual machine layer only results in subsequent changes in the *AbstractDispatcher*'s implementation class at the architecture-layer. Similarly, changing the interface of network communication abstractions (e.g., socket) at the virtual machine layer only results in subsequent changes in the *AbstractDistribution*'s implementation class.

4.2. Serialization

By leveraging JVM's object serialization facility, we were able to send and receive event objects as well as software components over the network. Since there is no similar facility in C++, we had to implement our own serialization facility. For this we created a class called *AbstractSerialization* that exports two abstract methods: *toArray*, which returns a given object in a byte array format; and *fromArray*, which given a byte array representation of an object creates an instance of that object. To make an object serializable we simply extend *AbstractSerialization* and provide the appropriate implementation for these two methods. For convenience the virtual machine layer provides the default implementation of serialization for the basic data types (e.g., integer, string) as well as Prism-MW specific data types (e.g., *Component*, *Event*). For any user-defined data type (e.g., application logic associated with a *Component*, application-specific payload attached to an *Event*), the application developer would have to provide the appropriate implementation of *AbstractSerialization*. While this approach is not as convenient as that provided by Java[1] it is more efficient: object serialization in Java requires more information to be included in the serialization stream (e.g., each data type needs to be tagged appropriately).

1. To make an object serializable, in Java the application developer simply implements the *Serializable* interface.

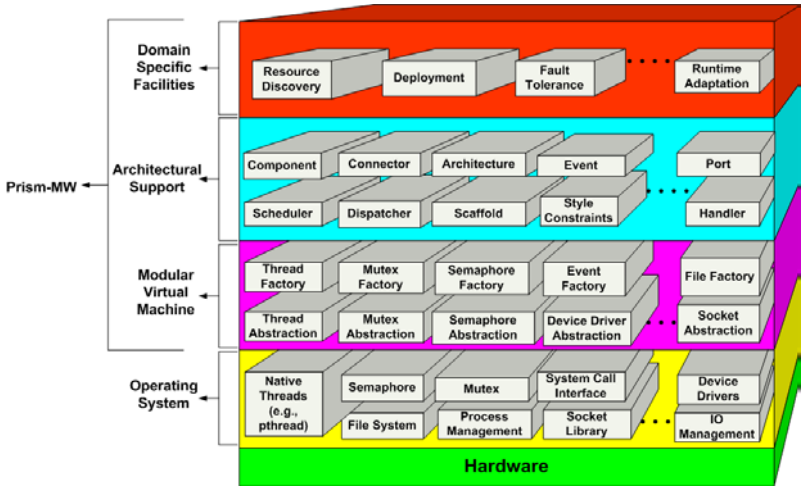

Figure 3. Layers of the Prism-MW architecture.

4.3. Memory Management

In the Java version of Prism-MW, we relied on the JVM to manage the (de)allocation of memory for Java objects at runtime. While this approach incurred an overhead (i.e., wastes both computational resources and time), with Java realistically we did not have any other alternatives as we were limited to JVM's somewhat unpredictable and uncontrollable memory management mechanism. A similar overhead also exists in C++, where the (de)allocation of memory on the heap by both Prism-MW and application logic incurs a significant overhead. We were not able to ignore this type of overhead in MIDAS, since it had stringent latency requirements of transmitting an alarm from a sensor to a hub and receiving an acknowledgement back in less than two seconds. 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 in turn 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 anymore. 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 3) in the virtual machine layer, irrespective of whether the *Event* is allocated from the heap or from a memory pool.

4.4. Programming Language Interoperability

As mentioned in the introduction of Section 4, the MIDAS subsystem running on the PDAs was developed using the Java version of Prism-MW. However, since it had to interact with the C++ application running on gateways and hubs, we faced a new type of heterogeneity at the level of PL that was not abstracted away by the MVM. This was due to the difference between the representation of objects in Java and C++. For example, a Java character is represented as two bytes, but in standard C++ a character is represented as one

byte. To support this type of heterogeneity, we leveraged Prism-MW's extensibility support: we created a new implementation of *AbstractDistribution* called *JavaToC++InteropDistribution* which translates an event message from the C++ message format to Java format and vice versa. This experience showed that an architectural middleware's extensibility and flexibility are essential to cope with the kinds of heterogeneity that are not abstracted away by a virtual machine layer (e.g., application- or PL-level heterogeneity).

4.5. Deployment

One of the greatest challenges faced by the engineers of heterogeneous embedded systems is the lack of available tools to support the deployment and verification of software. This problem is further exacerbated in highly distributed and pervasive systems such as MIDAS, which often lack a convenient I/O interface (e.g., monitor, disk drive, keyboard) that could be used for download and installation of software. Therefore, support for software deployment is an essential feature of a middleware geared to this domain. Unlike many traditional middleware platforms, which support software installation at the granularity of executable software image or patch, an architectural middleware can provide support for deployment at the more appropriate level of architectural components and connectors. In developing our deployment capability we have leveraged Prism-MW's support for architectural reflection, as further detailed below.

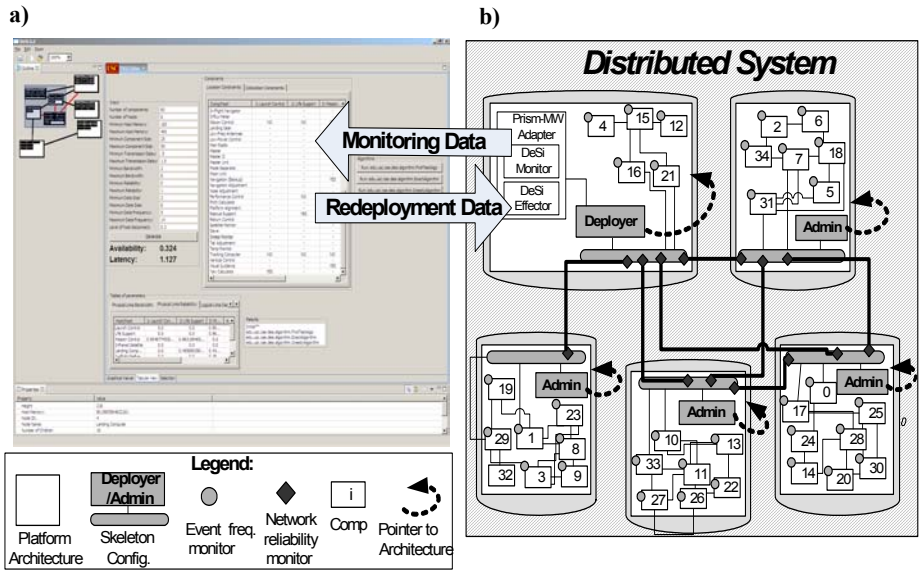

Figure 4. Deployment and runtime analysis suport: a) DeSi's tabular view of system's deployment data, b) a system running on top of Prism-MW that is monitored and managed by meta-level components.

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. 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 1). 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.

For component deployment we needed a mechanism to load the components at runtime. However, unlike Java, C++ does not provide a mechanism for dynamic class loading. Therefore, to address this shortcoming, we first compiled each component's implementation either as a Dynamic Link Library (DLL) for Windows or as a Shared Library for Linux. We then used the file serialization facility implemented in the C++ version of Prism-MW for serializing the DLL or Shared Library file into byte-array format. Finally, the byte-array was transmitted over the network to its target host, deserialized, and stored as a DLL or a Shared Library locally. When the software system is started, the application's implementation encapsulated within the DLL or Shared Library is bound to the middleware facilities.

4.6. Runtime Analysis and Adaptation

Very often engineers do not know *a priori* the properties of the target hardware platform, and early on make decisions that may not be appropriate within the context of the actual running system. This is of particular concern in a mobile embedded system such as MIDAS that is affected by unpredictable movement of target hosts (e.g., PDA users) and fluctuations in the quality of wireless network links (e.g., bandwidth, reliability). In the case of MIDAS, we realized that different deployments of the software components onto hardware platforms (i.e., different *deployment architectures*) 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's architectural facilities as well as our interactive deployment analysis environment, called DeSi [11] (shown in Figure 4a). DeSi provides the ability to model the system's deployment architecture, visualize and assess the architecture, and improve it via one of its redeployment algorithms.[2]

Figure 4b depicts an example distributed system of 5 hosts and 35 components that is monitored and deployed on top of Prism-MW. We have already discussed *Admin*'s role in the deployment and dynamic adaptation of a system's architecture. To monitor the various system properties, we leveraged Prism-MW's *AbstractMonitor* class, which is associated through the *Scaffold* with every *Brick* (recall Figure 1). This allows for autonomous, active monitor-

2. While the architectural analysis issues captured by the redeployment algorithms have been a major focus of our research to date, we are unable to elaborate on them in this paper due to space constraints.

ing 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* component, called *Deployer*, for aggregating the monitored data. As shown in Figure 4b, we integrated DeSi with Prism-MW, by wrapping DeSi's *Monitor* and *Effector* components via a Prism-MW *Adapter*. Once the *Deployer* component has received the monitoring data from all the *Admin* components, it sends the data to DeSi, which populates its model. At that point, one of the algorithms provided by DeSi is selected and executed for improving the system's deployment architecture. Finally, the result is reported back to the *Deployer*, which coordinates the redeployment of the system with the help of the *Admin* components.

4.7. Resource Discovery

MIDAS gateways and sensors may become unavailable or unreachable for many reasons: network disconnection, hardware and software failure, or running out of battery power. 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 reside on a gateway. As shown in Figure 3 and discussed further 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 1). 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 (recall Section 3.1) to determine the services installed on the local host. Service discovery components leverage *DistributionEnabledPort*s to communicate with other service discovery components via *Event*s. Supporting service discovery via Prism-MW's architectural constructs thus provides location transparency at the level of architecture.

5. Lessons Learned

Our experiences with MIDAS, which is both more heterogeneous and had more stringent requirements than other application scenario to which Prism-MW had been applied, inspired us to reassess some of our earlier assumptions and design decisions. In the process, this has helped us to further understand the nature of architectural middleware. In this section we discuss some of the more salient lessons we have learned, which, in turn, we have used to formulate the high-level architecture for an architectural middleware platform shown in Figure 3 and discussed above.

5.1. Design of an Architectural Middleware

The initial design of Prism-MW leveraged many abstraction facilities provided by JVM. However, as a result of implementing MIDAS primarily in C++, it became clear that some of our initial assumptions were not generally applicable. We realized that an architectural middleware is composed of three distinct layers of functionality, as shown in Figure 3: at the very 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 advanced *domain-specific facilities*.

We also realized that there are a number of advanced facilities that should be supported by an architectural middleware in the embedded systems domain. We already discussed some of those in the context of the MIDAS project: deployment, runtime analysis, adaptation, and resource discovery. The common guiding design decision behind these services has been their implementation using the architectural constructs provided by Prism-MW. This approach has a number of advantages. First of all, it helps to keep the middleware's core small and efficient. Furthermore, it allows us to reap all the benefits of using an architectural middleware for these facilities as well. For example, we can modify a distributed system's service discovery mechanism, by dynamically swapping the service discovery component (recall Section 4.7) with another implementation of it. Finally, the technique allows for efficient monitoring and adaptation of the system via the architectural awareness capability provided by Prism-MW.

5.2. Flexibility and Extensibility

Another important observation is that there are sources of heterogeneity other than those of the underlying hardware and system software. An example of this, discussed in Section 4.4, dealt with heterogeneity at the level of PL. Similar sources of heterogeneity can also be found in other aspects of a middleware. For example, there are different protocols for establishing trust and determining group membership among hosts in an ad-hoc environment. Therefore, while a virtual machine layer such as MVM can abstract away the heterogeneity of the hardware and system software, it is not sufficient by itself. Rather, the middleware should be flexible and extensible, such that heterogeneity at the level of application can also be resolved by adapting and extending each of the three middleware layers appropriately.

5.3. Efficiency versus Configuration Complexity

Recall from Section 4.1 that MVM's resource factories were leveraged to manage the utilization of system resources. In fact, since all of the architectural constructs are treated as resources and are pre-allocated from the memory pool, we are able to estimate a system's resource consumption from its software architectural models (even at design-time). This in turn allows us to analyze and inspect the impact of architectural change on resource usage. This level of control over resources is extremely important in resource-constrained systems. However, it also has a drawback, as it increases the complexity of system configuration. For example, consider some of the configuration parameters required in the C++ version of Prism-MW:

- size of the event queue;
- number of pre-allocated system resources: semaphore, mutex, file, DLL, etc.;
- number of pre-allocated architectural constructs: Component, Connector, Port, etc.;
- size of the memory buffer used by the network sockets; and
- size of pre-allocated memory pool used by the application-level variables.

On the other hand, in the Java version of Prism-MW, there are only two configuration "knobs": size of event queue and thread pool. However, as mentioned earlier, the Java version of Prism-MW incurs an overhead due to the dynamic allocation of resources. Furthermore, it is also highly unpredictable, which makes it harder to estimate and control an application's resource usage at the level of architecture.

The above discussion indicates that there is a trade-off between resource utilization control and the configuration complexity of a middleware solution. Increased control over resource utilization allows for the development of more efficient systems. On the other hand, increased complexity in a middleware hampers its ease of use and validation. This suggests that developing a "one size fits all" solution is impractical; instead, it is the software engineer's responsibility to determine the appropriate middleware solution based on the characteristics of the given application domain.

5.4. Validation and Verification

One of the greatest challenges we faced in the MIDAS project was validation and verification of an application on the target platforms. Early on it became clear that manual testing, debugging, and installation of software is infeasible. Every time a bug was fixed, an updated version of the software had to be installed manually on the various devices, which resulted in an extremely time consuming and redundant task for the software developers. Advanced facilities, such as deployment and runtime analysis support, proved to be essential as they allowed for rapid deployment, automatic monitoring, and analysis to ensure that the system is functioning correctly.

Another reason that validation and verification in this domain is hard can be attributed to the fact that the underlying virtual machine abstractions may not actually be able to abstract away completely the behavioral variations in the computing substrates. For example, in the MIDAS project, we initially developed and tested the software targeted for the gateways on top of Windows. We relied on the MVM layer to insulate us from OS-level variations such as different API used for thread synchronization. After testing the application on Windows, we ported it to the gateway platforms running Linux (with the Linux version of MVM) for the final evaluation. However, the application kept failing on the gateways. Eventually, we found the source of failure to be the variations among the two OSs' thread synchronization policies: Windows allows for the invocation of two consecutive *lock* operations on the same semaphore, while Linux prevents this by throwing an exception.

This example also demonstrates that, as we provide more facilities in a middleware solution, it becomes harder to evaluate applications developed on top it. In fact, as we already hinted in the previous section, another culprit in making it harder to evaluate applications was the complexity of configuring the C++ version of Prism-MW. For example, since the resources are pre-allocated at system start-up, if at runtime Prism-MW runs out of available resources, it will result in a software failure. This is clearly a trade-off when compared to the Java implementation: Java will dynamically allocate all the resources needed by an application hosted on Prism-MW, but at a performance cost.

5.5. Advanced Facilities

As we already mentioned, a middleware's support for architecture-based development reduces the possibility of architectural erosion during the software construction phase. However, since a large class of embedded systems are long-lived and pervasive, they are constantly evolving to the changing environment around them. Therefore, an architectural middleware for this domain should not only provide support for the implementation of a system in terms of its architectural elements, but also facilities that minimize the potential for software architectural erosion after the initial deployment. The deployment and runtime analysis environment that we discussed earlier are good examples of facilities that can be leveraged to keep the system's software architecture and the actual running system's software architecture in sync. For these tools to be applicable, they have to be able to represent the dynamic nature of software architectures such that the architectures can be analyzed for their properties, and also have the ability to configure a running system based on the results of the analysis. Our experience has showed that by supporting these facilities on top of Prism-MW's architectural facilities, we can directly leverage the reflection and monitoring capabilities provided by the middleware to change the running system.

6. Acknowledgement

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

7. References

[1] J.Aldrich, et. al. ArchJava: Connecting Software Architecture to Implementation. *International Conference on Software Engineering*, Orlando, Florida, May 2002.

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

[3] Orbix/E. http://www.iona.com/whitepapers/orbix-e-DS.pdf

[4] JXTA Project. http://www.jxta.org/

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

[6] S. Malek, et. al. Prism-MW: A Style-Aware Architectural Middleware for Resource Constrained, Distributed Systems. *IEEE Trans. on Software Engineering, 31 (3)*, March 2005.

[7] C. Mascolo et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Personal and Wireless Communications*, Kluwer.

[8] C. Mattmann, et. al. GLIDE: A Grid-based Lightweight Infrastructure for Data-intensive Environments. *European Grid Conference*, Amsterdam, Netherlands, February 2005.

[9] N. Medvidovic and M. Mikic-Rakic. Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility.*Software Engineering and Mobility Workshop* , Toronto, Canada, May 2001.

[10] N. Medvidovic, et. al. Software Architectural Support for Handheld Computing. *IEEE Computer* , September 2003.

[11] M. Mikic-Rakic 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.

[12] M. Mikic-Rakic and et. al. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. *Int'l. Middleware Conference*, Rio De Janeiro, Brazil, June 2003.

[13] D.E. Perry, et.a l. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.

[14] D. Schmidt. ACE. http://www.cs.wustl.edu/~schmidt/ACE-documentation.html

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

[16] J. P. Sousa, et. al. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *Working Conf. on Software Architecture*, Montreal, August 2002.

[17] Sun Microsystems. JINI(TM) Network technology. http://wwws.sun.com/software/jini/