Reconceptualizing a Family of Heterogeneous Embedded Systems via Explicit Architectural Support

Sam Malek[1] Chiyoung Seo[1] Sharmila Ravula[2] Brad Petrus[3] Nenad Medvidovic[1]

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

[2]Bosch Rsrch & Tech. Center
4009 Miranda Avenue
Palo Alto, CA 94304, U.S.A.
sharmila.ravula@rtc.bosch.com

[3]Bosch Rsrch & Tech. Center
Two NorthShore Center, Suite 320
Pittsburgh, PA 15212, U.S.A.
brad.petrus@rtc.bosch.com

Abstract

It has been widely advocated that software architecture provides an effective set of abstractions for engineering (families of) complex software systems. However, architectural concepts are seldom supported directly at the level of system implementation. In embedded environments in particular, developers are often forced to rely on low-level programming languages. While this is conducive to fine-grain control over the system, it does not lend itself to addressing larger issues such as ensuring architectural integrity or managing an application family. In this paper we describe our experience with fundamentally altering the manner in which a family of embedded applications is designed, analyzed, implemented, deployed, and evolved using explicit architectural constructs. We discuss our strategy, the challenges we faced in the course of our project, the lessons learned in the process, and several open issues that remain unresolved.

1. Introduction

Wireless sensor network (WSN) systems are fast becoming pervasive in a variety of domains including medical, defense, industrial automation, navigation, and civil engineering. These systems are associated with a number of advantages, such as low cost of installation and maintenance, replacement of physical sensor nodes, easy reconfigurability, and so on. At the same time, they face significant challenges, including requirements for scalability, fault-tolerance, performance (e.g., response time), availability, and dependability. Additionally, the software applications deployed on WSNs are faced with extreme constraints in terms of available power, memory, processor or controller speed, network throughput, and so on.

These issues are reflective of WSN applications being investigated at Bosch for use in areas such as monitoring, tracking, and control. The WSNs used in these applications may need to be integrated with legacy wired networks, other embedded devices, and mobile networks that include PDAs and cell-phones for user notification. This suggests that an effective underlying infrastructure used in the development, deployment, and execution of these applications has to be generic enough for use in multiple application domains, and flexible enough to offer easy customization for specific application requirements and heterogeneous operating environments. In other words, the distributed systems deployed using WSNs require a shared application substrate in the form of a *middleware platform* [5].

At the same time, conventional software engineering wisdom suggests that a most promising approach to addressing the challenges of developing complex software systems, such as those discussed above, is to employ the principles of *software architecture* [15]. Software architectures provide abstractions for representing the structure, behavior, and key properties of a software system. They are described in terms of software *components* (computational elements), *connectors* (interaction elements), and their *configurations*. A given software architectural *style* (e.g., publish-subscribe, peer-to-peer, pipe-and-filter, client-server) further refines a vocabulary of component and connector types and a set of constraints on how instances of those types may be combined in a system.

In practice, however, while software developers have embraced architectural abstractions as powerful design-level tools, they are typically forced to realize those abstractions using a different set of implementation-level tools. For example, engineers may prefer to think of systems in terms of components, connectors, and styles, but usually have to implement those systems using low-level constructs such as methods, arrays, pointers, and so on [9]. This is particularly prevalent in embedded systems, in which C/C++ has remained the *lingua franca*. Even the state-of-the-art middleware solutions for embedded systems development (e.g., CORBA Orbix [14], TAO [1]) still lack the necessary implementation-level facilities for key elements of software architecture. For example, explicit architecture-level connectors are usually distributed (and thus "lost") across different implementation-level modules as combinations of method calls, shared memory,

network sockets, and other facilities supported in the middleware [11]. Another example is that of application-level architectural styles (e.g., pub-sub or peer-to-peer), which are often at best mimicked, and at worst ignored, by the styles (e.g., client-server) assumed by the middleware.

For this reason, in the past the researchers from the software architecture group at USC have argued that traditional middleware solutions are not the best candidates for architecture-based software development [9]. Instead, we have shown that *architectural middleware*—a middleware platform that provides native implementation-level constructs for the key architectural abstractions—can be employed effectively in a number of domains, including embedded systems [7][10].

Applying an adaptable architectural middleware platform to WSN-based applications thus seemed like a natural choice, and we decided to undertake the challenge collaboratively between Bosch and USC. The resulting experience yielded some outcomes we had not anticipated, as it forced us to reconceptualize how WSN-based applications are built most effectively. Similarly, we also had to rethink several assumptions and design choices made originally in the development of Prism-MW. In the process, we have drawn several lessons we believe to be more broadly applicable to software development in the WSN arena, the role of software architecture in heterogeneous embedded systems, and the nature of architectural middleware.

The remainder of the paper is organized as follows. Section 2 outlines a set of architectural challenges and requirements for developing distributed embedded systems. Section 3 then discusses the related work in terms of those requirements. Section 4 provides an overview of the approach we took in this project. Section 5 describes and evaluates our experience in engineering a family of WSN applications using our approach. Finally, a discussion of the lessons we have learned and a brief conclusion round out the paper.

2. Requirements

Early on in the process of formulating our collaborative project, we engaged in a requirements elicitation activity. The requirements were driven by one of Bosch's families of embedded applications, called MIDAS. MIDAS is composed of a large number of sensors, gateways, hubs, and PDAs that are connected wirelessly in the manner shown in Figure 1. The sensors are used to monitor the environment around them. They communicate their status to one another and to the gateways via events. 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 to the hubs. Hubs are used to evaluate and visualize the sensor data for human users. They also provide an interface through which a user can send control commands to various sensors and gateways. Hubs may also be configured to propagate the appropriate sensor data to PDAs, which are then used by the mobile users of the system.

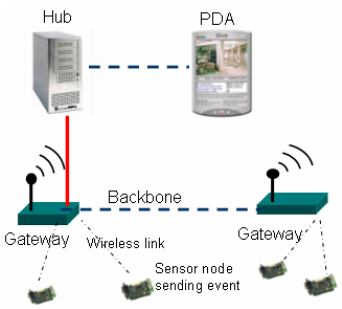

Figure 1. MIDAS

We identified a set of eleven key requirements driving the middleware, as detailed below. We should note that the first five requirements are non-functional or quality attributes; the next four deal with system development support; finally, the last two requirements impose a software architectural perspective on the middleware.

R1: Resource Consumption. Since we are concerned with highly resource-constrained embedded systems, it is important that the middleware be efficient in its use of the resources required to deploy it. In particular, the code space and memory required by the middleware should be minimized to realize cost-savings in terms of FLASH or EPROM hardware. Also, the communications data overhead should be minimized to support communication over low-bandwidth channels with minimal latency. Furthermore, the middleware should also provide the appropriate facilities for managing system resources.

R2: Performance. Application types that will be deployed using the middleware include time-critical data transfers from static and mobile sensors. The factors affecting the middleware performance include performance of the hardware on which the application/middleware is executed, of the network, and of the application code. In light of these parameters, the middleware should provide latency guarantees and support priority-based scheduling.

R3: Scalability. The middleware is intended to be deployed in systems that could scale from a control unit monitoring 10-15 nodes to a multi-control unit, multi-hub network with hundreds of sensors and tens of control units. The middleware itself could be deployed on a sensor with extremely limited computing resources or a powerful desktop platform. All such variations should be effectively supported by the middleware.

R4: Heterogeneity. Since the middleware is intended to be deployed on embedded systems in many domains, it is important that it be able to support multiple OSs and platforms. For example, the middleware may be deployed on embedded systems running VxWorks and eCos. It is also conceivable that an alternative OS may become available during a product's lifetime, in which case it should be possible to redeploy the middleware and the applications developed on top of it to that OS with little effort.

R5: Fault-Tolerance. Many distributed, embedded applications operate in safety-critical or regulated environments, in which a given event must be delivered to its recipients under all conditions (e.g., medical emergency). The middleware platform must support fault-tolerance using fail-over techniques: if a host providing a given service fails, then another host in the system should quickly be enabled as the provider of the same service. The computational state of the newly enabled service must be the same as the last known state of the failed service.

R6: System Modeling and Analysis. The ability to explicitly model and analyze a distributed system is especially important in the case of large systems comprising hundreds of hosts and providing hundreds of software services. The middleware must support analysis of a distributed application both prior to deployment and during runtime, in order to select the location of various system services that will minimize the communication latencies and battery consumption, maximize performance and service availability, and so on.

R7: Deployment. We require the middleware platform to provide facilities for component download and deployment. This is especially important for WSN systems, such as MIDAS, that lack a convenient I/O interface (e.g., monitor, disk drive, or keyboard) that could be used for download and installation of software. Additionally, if a new (version of a) component has been developed, it would be advantageous to have the ability to deploy it without having to remove the target host from the system. Given the nature of our target computing environment, component deployment is to be supported on multiple platforms and in multiple programming languages (PLs).

R8: Service Discovery. In large distributed systems, service providers and clients may join and leave the system at arbitrary times. Therefore, the middleware must support the ability of a client to discover and/or invoke a service without prior knowledge of its physical location.

R9: Monitoring. It is important to be able to non-intrusively monitor the activity of a distributed, long-lived application for properties such as event rate, resource consumption, device load, amount of battery power remaining, and so on. For example, if the network traffic at a given host is particularly high, some of the local services may be migrated to a less congested host.

R10: Architecture-Based Development. The middleware should support architecture-based development since WSN application designs are increasingly expressed in architectural terms, including components, connectors, ports, and events. This requirement eases the translation from high-level design to implementation.

R11: Multiple Architecture Styles. Applications in the WSN domain typically employ several architectural styles. For instance, it is not unusual for different parts of a single application to be developed using both pub-sub and client-server styles. Direct support for heterogeneous styles is thus required of the middleware platform.

3. Related Work

We classify the related literature into (1) technologies targeted at supporting implementation of software architectures and (2) middleware for embedded systems. We provide an overview of the most notable solutions from each category, and relate them to the above requirements. We highlight the requirements each solution *fails* to satisfy, as it suggests our motivation in searching for an alternative.

ArchJava [2] is an extension to Java that provides PL-level constructs for architectural concepts (component, connector, port), and ensures that the implementation conforms to architectural constraints. ArchJava has several limitations with respect to the above requirements (e.g., R1, R3, R4, R8), including its assumption of a single address space. Aura [17] is an architectural style and supporting middleware for ubiquitous computing. Aura's performance and scalability have not been assessed in widely distributed settings (R1, R2, and R3), and it imposes a single style for all applications (R11). Finally, RUNES [4] is a reconfigurable component-based middleware, but it does not support a number of software architectural elements, such as connectors and styles (R10, R11). Furthermore, since RUNES's component framework is tied to the implementation platform, it is not possible to redeploy components among heterogeneous platforms (R4, R7).

None of the middleware platforms surveyed below support requirements R10 and R11. Furthermore, it can be argued that only the first two, CORBA-based solutions, support R6 to some extent. However, these solutions address several of the other requirements, and have certainly influenced our work. Orbix/E [14] is a lightweight CORBA ORB optimized for embedded applications, including its relatively small memory footprint. The ACE ORB (TAO) [1] is a CORBA-compliant middleware framework that allows clients to invoke operations on distributed objects without concern for object location, PL, OS, communication protocol, or hardware. XMIDDLE [8] is a data-sharing middleware for mobile computing, which ensures consistency of the shared data across hosts. XMIDDLE is lightweight and fast, and caters to the frequent disconnections of mobile devices. Lime [6] is a Java-based middleware targeted at ad-hoc mobile environments. It provides a coordination layer for designing applications that exhibit logical and/or physical mobility. Finally, MobiPADS [3] 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 performance of mobile applications.

4. Overview of the Approach

To meet the requirements discussed in Section 2, and to address the shortcomings of the related approaches discussed in Section 3, we extensively customized and integrated three technologies: a cross-platform virtual machine, an architectural middleware, and a distributed system modeling and analysis environment. In this section, we provide an overview of what each of these technologies contributed, and outline how they were integrated to arrive at a comprehensive architecture and tool-suite. A much more detailed discussion of the challenges we encountered in this process and the significant modifications that were made to each of the technologies are provided in Sections 5 and 6. As shown in Figure 2, at the bottom of the resulting integrated architecture is a *virtual machine layer* that allows the middleware to be deployed on heterogeneous OS and hardware 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 constructs are used to implement various *domain-specific computing facilities*.

4.1 Modular Virtual Machine

To cope with the heterogeneity and resource-constrained nature of the hardware platforms frequently encountered in embedded systems, we adapted a domain-specific virtual machine called Modular Virtual Machine (MVM). MVM was designed at Bosch as a configurable family of utilities, including support for threading, IO management, networking, and so forth (see the MVM layer in Figure 2). MVM provides an abstraction layer on top of various OSs (Linux, Windows, eCos) and hardware platforms (Intel x86, KwikByte, and several proprietary sensor platforms). It is composed of three parts: *resource abstractions*, *implementations*, and *factories*. Resource abstractions provide a common API that is leveraged by the higher middleware layers as well as application developers to produce platform-independent code. An example of a resource made available via resource abstraction is a thread. A resource abstraction is realized via its implementation, which may use OS- or hardware-specific libraries. Resource abstractions are managed via their corresponding factories. For example, a thread factory may manage the number of threads that could be created in the system. For a given target host, the executable image of MVM is created by building the MVM source code with the appropriate implementation files included.

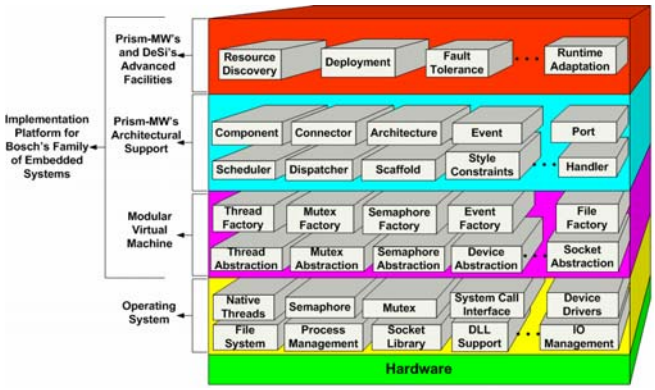

Figure 2. Layers of System Stack.

Given MVM's support for the development of platform-independent source code, it was hoped that MVM would allow for a higher degree of software reuse, as well as support for more advanced capabilities, such as code mobility, runtime adaptation, and so on. However, application modules developed on top of it frequently suffered from unintended dependencies. Furthermore, since MVM lacked support for service discovery, dynamic adaptation, and component-level deployment, software systems developed on top of it were compiled into monolithic, rigid executable images. These shortcomings were a direct by-product of MVM's lack of support for architecture-based development, and ultimately hindered its potential to promote software reuse and adaptability among distributed embedded applications.

4.2 Prism-MW

Prism-MW [7] is an architectural middleware developed at USC. It supports architectural abstractions that enable direct mapping between an architecture and its implementation. Figure 3 shows the class design view of Prism-MW. The shaded classes constitute the middleware *core*, a minimal subset of Prism-MW that enables implementation and execution of architectures in a single address space. We describe the design of Prism-MW in more detail here because we have leveraged it extensively in this project, as will be seen in Sections 5 and 6.

Brick represents an architectural building block and encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components,

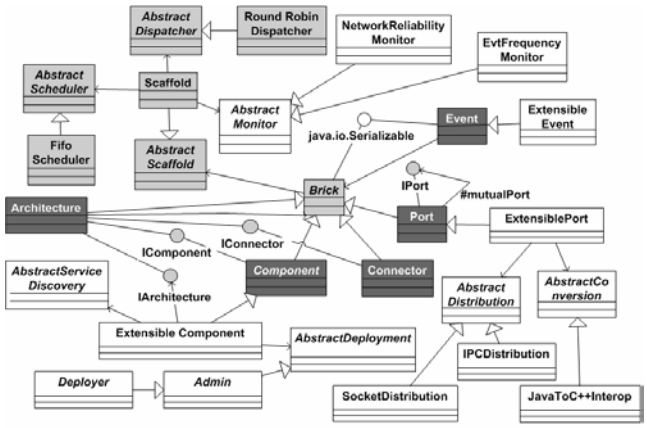

Figure 3. UML class diagram of Prism-MW's design. Middleware core classes are highlighted.

connectors, and ports, and provides facilities for their addition and removal. A distributed application is implemented as a set of interacting *Architecture* objects.

*Event*s are used to capture communication between components. An event consists of a name and payload, i.e., a set of typed parameters that carry data and meta-level information (e.g., sender, time stamp). An event type is either a *request* for a recipient to perform an operation or a *reply* that a sender 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. Each port has a type, which is either *request* or *reply*. Request events are always forwarded from request to reply ports; reply events are forwarded in the opposite direction.

*Component*s perform computations in an architecture and may maintain their own internal state. A component can have any number of attached ports, used for exchanging events. 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, a connector can have any number of attached ports. 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, 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.

Prism-MW's design is intended to be highly extensible while keeping Prism-MW's core unchanged. To that end, the core constructs (e.g., *Component*, *Port*) are subclassed via specialized classes (e.g., *ExtensibleComponent*, *ExtensiblePort*), each of which has a reference to a number of abstract classes (Figure 3). Each *AbstractExtension* class can have multiple implementations, thus enabling selection of the desired functionality inside each instance of a given extensible class.

Prism-MW's support for architecture-based development and its highly flexible nature made it an appropriate solution to many of the requirements discussed in Section 2. However, before Prism-MW could be used in this project, we had to overcome a number of challenges. The existing version of Prism-MW had been developed in Java, and relied on JVM to abstract away the heterogeneity of the computing substrate. On the other hand, for legacy and efficiency reasons most embedded applications are developed in C and, more recently, C++. Therefore, we decided to port Prism-MW to C++. To safeguard Prism-MW from the heterogeneity of the computing substrate (e.g., different C++ runtime libraries or threading semantics on different platforms) we decided to reimplement it on top of MVM. But there still were a number of unanswered questions: What design implications will this have on Prism-MW and MVM? How efficient will a solution developed on top of the resulting middleware be? And so on. In Sections 5 and 6, we will revisit these issues in much more detail.

4.3 DeSi

DeSi [13] is a visual environment that supports specification, analysis, and manipulation of a distributed software system's *deployment architecture* (i.e., allocation of the system's software modules on its hardware hosts). DeSi provides an XML-based meta-modeling language that can be used to specify arbitrary properties for software components and connectors (e.g., maximum memory usage, frequency and volume of exchanged data), as well as hardware devices and their network links (e.g., CPU speed, available memory, network throughput). DeSi completely separates the underlying system model from the various visualization facilities that it provides. This simplifies the development of new customized views that may focus on a particular aspect of the system (an example view of DeSi is shown in Figure 4b). Users can leverage multiple views of the system's model to explore and analyze its various properties. Furthermore, since by default DeSi synchronizes all views with the underlying model, it is a suitable environment for visualizing and monitoring the dynamic aspects of a running system. In addition to its visualization and modeling capabilities, DeSi also provides an API for accessing the system model, which we have leveraged to develop a number of algorithms for analyzing a given system's software architecture and improving its properties (e.g., availability, latency) via redeployment [12].

DeSi's modeling and visualization capabilities made it an appropriate tool for representing the software and hardware architectures of distributed embedded systems such as WSN systems. Its plug-in architecture also facilitated the development of new algorithms for architectural analyses. Our objective was to leverage DeSi's analysis capabilities both at design-time and at runtime, such that we can possibly adapt a running system based on the results of the analysis. However, in order for DeSi's model to be populated with a system's runtime data, and the results of its analysis to be effected on the running system, we needed to integrate DeSi with the implementation substrate (i.e., Prism-MW and MVM). This posed several challenges, on which we will elaborate in the next section.

5. Experience

The direct motivation for the collaborative project between Bosch and USC was the construction of the MIDAS family of WSN-based applications. Our pursuit of this immediate goal resulted in a significant adaptation

and integration of the three infrastructure tools discussed in the previous section. For exposition purposes, we will describe this experience in terms of an instance of MIDAS's reference architecture, shown in Figure 4a. Furthermore, we will organize our discussion around the requirements introduced in Section 2. Note that we do not explicitly address requirements R6, R9, and R10, since DeSi and Prism-MW natively support them and no significant changes to them were required. Furthermore, several requirements (e.g., R5) are discussed in a different order than in Section 2 because we use facilities built in support of other requirements (e.g., R8) to address them.

5.1 Resource Consumption (R1)

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 in terms of both computational resources and time, with Java we realistically did not have other alternatives as we were limited to JVM's somewhat unpredictable and opaque memory management mechanism. A similar overhead also exists in C++, caused by the (de)allocation of memory on the heap by both Prism-MW and application logic. We were not able to ignore this type of overhead in MIDAS. To address this problem we enhanced the 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 of 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 architectural constructs. For example, a component generates an *Event* via an API exported by the *EventFactory* facility (shown in Figure 2) in the MVM layer, irrespective of whether the *Event* is allocated from the heap or from a memory pool. The total memory footprint of the C++ portion of the MIDAS application shown in Figure 4 was measured to be 3.1 MB, while Prism-MW's memory overhead was 189 KB, or 6%.

5.2 Performance (R2)

MIDAS has stringent real-time requirements, such as transmitting a high-priority event from a sensor to a hub and receiving an acknowledgement back in less than two seconds. Prism-MW's default event processing mechanism [7], which is based on a FIFO implementation of event queue and round robin dispatching of threads (i.e., *FIFOScheduler* and *RoundRobinDispatcher* shown in Figure 3), was not sufficient for guarantying the delivery of such high-priority events. For this reason, we developed an implementation of *AbstractScheduler*, called *PriorityBasedScheduler*, which maintains multiple event queues with different priorities. We also developed an implementation of *AbstractDispatcher*, called *PriorityBasedDispatcher*, which maintains multiple thread pools with different priorities. A thread pool is associated with an event queue of the same priority. Thus a high-priority event is processed (by a high-priority thread) before a lower-priority event. To avoid starvation of low-priority events, *PriorityBasedDispatcher* can be configured to periodically pre-empt the high-priority thread pool and allow the low-priority pool to process an event.

5.3 Scalability (R3)

In our previous work, we have shown that Prism-MW

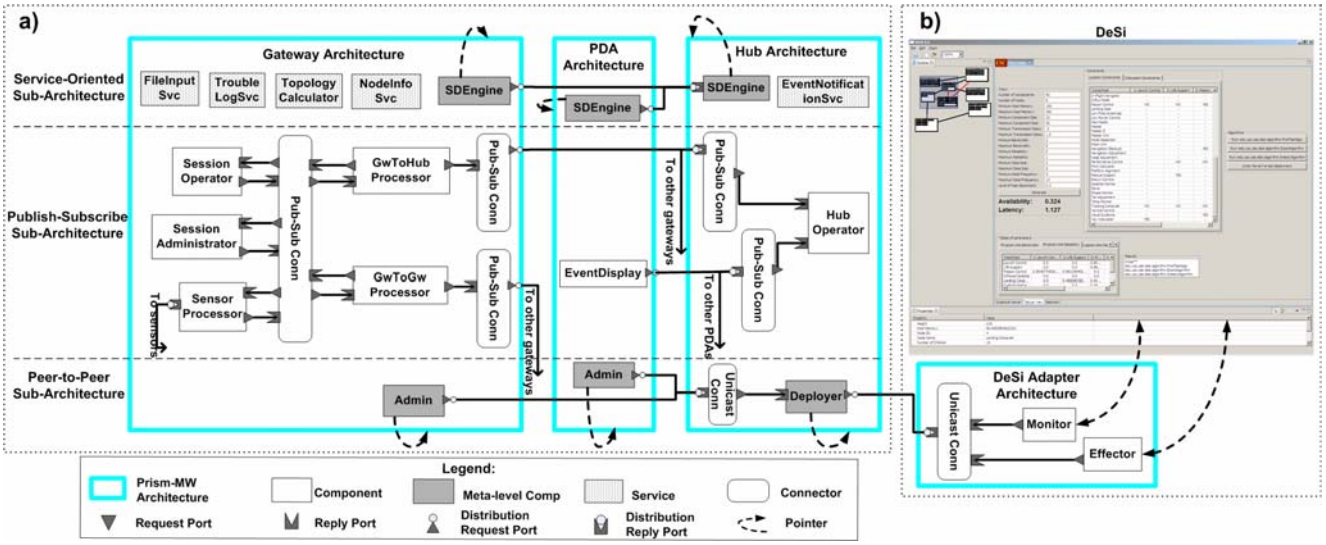

Figure 4. An abridged view of MIDAS's architecture that is monitored, analyzed, and adapted at runtime: a) portions of MIDAS's software architecture, including its three sub-architectures; b) DeSi and its Prism-MW Adapter

can scale up to hundreds of hardware devices and OS processes, and to architectures comprising thousands of software components and connectors [7]. To satisfy the specific scalability requirements of MIDAS outlined in Section 2, we relied on the modularity and extensibility across different layers of the middleware shown in Figure 2. The configurable nature of the MVM allows us to "strip" it of any functionality that is not required on a target platform. Similarly, recall from Section 4.2 that we are able to reduce Prism-MW's configuration down all the way to its core, which represents the minimum functionality required for deploying Prism-MW. MVM factory facilities were also leveraged to configure the resources used by the application (e.g., maximum number of threads to be used in the system). Therefore, the modularity and extensibility of both MVM and Prism-MW allowed us to customize each deployment of the middleware as well as the application running on it based on the properties of the target platform. We are currently in the process of assessing the limits of our middleware's scalability using variations of the configuration shown in Figure 1.

5.4 Heterogeneity (R4)

Several aspects of MIDAS embody the notion of multiplicity inherent in embedded environments. The devices on which MIDAS has been deployed are of several different types (ARM-based Compaq iPAQ and KwikByte, Intel-based Dell laptops, and several proprietary sensor platforms), and are running four OSs (Windows CE and XP, Linux, and eCos). MIDAS was also developed in two PLs: C++ for applications running on sensors, gateways, and hubs, and Java for applications running on PDAs (recall Figure 1). Furthermore, some of the devices were equipped with wireless network cards, while others only had infrared and serial port capabilities.

The heterogeneity of the Java portion of MIDAS was enabled by the JVM, which abstracts away the variations in the computing substrate. However, this was not the case with the C++ portion of MIDAS. As mentioned earlier and shown in Figure 2, we opted to satisfy this requirement by layering Prism-MW's architectural constructs on top of MVM's system facilities. To further understand the role of MVM, as an example consider the support it provides for threads. In Java, we relied on native thread and thread synchronization mechanisms. However, in C++ developers typically have to use the OS's support for threads, and to rely on OS-level semaphore or mutex libraries for thread synchronization. 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 provided similarly.

MIDAS's multi-lingual nature also introduced a type of heterogeneity that was not abstracted away by the MVM. This was due to the difference in the representation of objects between Java and C++. For example, a Java character is represented as two bytes, while in ANSI C++ it is represented as one byte. To solve this we leveraged the extensible nature of Prism-MW's ports. As shown in Figure 3, we created an implementation of *AbstractConversion*, called *JavaToC++Interop*, that translates an event message between the C++ and Java formats. Similarly, to address the heterogeneity of remote communication, we developed two different implementations of *AbstractDistribution* in Prism-MW: *SocketDistribution* (for devices with wireless cards) and *IPCDistribution* (for devices with infra-red or serial ports). Additionally, we had to enhance the MVM to provide the required platform-specific networking facilities: TCP socket, UDP socket, connection thread, network buffers, and so on.

Our approach to satisfying the heterogeneity requirements of MIDAS proved to be flexible and extensible, as supporting a new OS or hardware platform would require only the addition of host-specific resource implementations in the MVM. The design of the middleware's architectural support (recall Figure 3) remained intact as we ported it from Java to C++. This design also allowed for a clear separation between architectural and system-level concerns. For example, it helped us to categorize a *Port*'s *JavaToC++Interop* extension as an architectural facility (architectural layer of Figure 2), and *endian* conversion on which *JavaToC++Interop* may rely as a system-specific facility (MVM layer of Figure 2). Furthermore, due to the extensive separation of concerns built into Prism-MW, modifying MVM's facilities resulted in only localized changes to Prism-MW. For example, changing the threading API in the MVM only results in changes to *AbstractDispatcher*'s implementation class in the architecture layer of Figure 2. Similarly, changing the interface of network communication abstractions (e.g., socket) in the MVM layer only results in subsequent changes to *AbstractDistribution*'s implementation class.

5.5 Deployment and Adaptation (R7)

We have leveraged DeSi's system modelling and Prism-MW's architectural reflection capabilities to satisfy MIDAS's deployment requirement, as depicted in Figure 4. In order to migrate the desired set of architectural elements onto target hosts, a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW's *Architecture* object that contains an *Admin* component. *Admin* component is an *ExtensibleComponent* with the *Admin* implementation of *AbstractDeployment* installed on it (recall Figure 3 as well as the top layer of Figure 2). As such, the *Admin* component is "architecturally aware": it has a pointer to its *Architecture* and is able to effect runtime changes to it (instantiation, addition, removal, and (dis)connection of components and connectors). Additionally, *Admin* components on different

devices are *peers* capable of exchanging *ExtensibleEvent*s that contain application-level components and connectors.

As shown in Figure 4b, we integrated DeSi with Prism-MW, by wrapping DeSi's *Monitor* and *Effector* components via a Prism-MW *Adapter*. This allows us to model a system's deployment architecture in DeSi and associate each component in the model with its corresponding implementation. The deployment may be assessed for properties of interest via the analysis algorithms provided by DeSi [12]. Once a deployment is selected, DeSi's *Effector* sends a sequence of commands (via Prism-MW's *ExtensibleEvent*s) to a centralized *Admin* component, called *Deployer*. *Deployer* in turn propagates the commands to *Admin* components on each host, which then perform the tasks corresponding to the commands (e.g., download component, weld ports, start component).

Very often engineers will not know *a priori* the properties of the target hardware platform, and will make deployment decisions that may turn out to be inappropriate within the context of the actual running system. This is of particular concern in mobile embedded systems such as MIDAS, which are affected by unpredictable movement of target hosts and fluctuations in the quality of wireless network links. To address this issue, we have used Prism-MW's monitoring facilities to observe the system in action, and DeSi to visualize the system and suggest alternative deployments. Figure 4 depicts an example MIDAS application that is monitored and deployed on top of our middleware, and visualized in DeSi. Once the monitoring data on each device is sent by the local *Admin* to the *Deployer*, *Deployer* aggregates and forwards the data to DeSi's *Monitor* component, which in turn populates DeSi's 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 *Admin* components.

5.6 Fault Tolerance (R5)

Currently we do not fully support the fail-over requirement. However, we have provided a utility that will be used as the foundation for completing this task. We have developed a mechanism that replicates components identified as critical either by the system engineers or by DeSi (e.g., components that provide services essential to the processing of high-priority events). DeSi's analysis capabilities are then used to find the optimal allocation of each component's replicas, such that the effects of the failure of (1) the original component, (2) the host on which it is deployed, or (3) a network link, are minimized. Our solution currently does not guarantee that each client of the original component will have uninterrupted access at least to one of its replicas. Furthermore, the event routing and constant state updates to multiple replicas of multiple components in a system may be very costly in terms of resource consumption (R1) and performance (R2).

5.7 Service Discovery (R8)

We have directly leveraged Prism-MW's architectural constructs to satisfy MIDAS's service discovery requirement. In the context of MIDAS a service corresponds roughly to a component interface. We developed an implementation of *AbstractServiceDiscovery* shown in Figure 3 that provides the support for recording and retrieval of services. An *ExtensibleComponent* with an implementation of *AbstractServiceDiscovery* installed on it is called *SDEngine* (shown in Figure 4a and the top layer of Figure 2). *SDEngine* acts as a service discovery agent on its local host. It is "architecturally aware": it can access its architecture to determine the services installed locally. *SDEngine*s then communicate this information across hosts via events. Any remote service requests by application-level components are routed via their local *SDEngine*s, ensuring complete location transparency.

5.8 Multiple Architectural Styles (R11)

As depicted in Figure 4a, MIDAS's reference architecture encompasses three different architectural styles. We briefly discussed the peer-to-peer portion of this architecture in Section 5.5. The pub-sub portion of MIDAS corresponds to the communication backbone that is responsible for routing and processing of sensor data among the various platforms. Unlike the services provided by pub-sub components that are platform-specific, MIDAS applications also require a number of more generic but less frequently used services. To minimize resource utilization, these services are distributed among the platforms and comprise the service-oriented portion of MIDAS. The extensible nature of Prism-MW has enabled us to directly support the three different architectural styles. In the previous section, we discussed how this extensibility was leveraged to provide a service discovery mechanism, which forms the centrepiece of our support for the service-oriented style. Similarly, we implemented support for the pub-sub style by extending and adapting some of Prism-MW's architectural abstractions. For example, a pub-sub connector (shown in Figure 4a) is implemented as an *ExtensibleConnector* that overrides the default routing policy of a basic Prism-MW connector. In MIDAS, the pub-sub sub-architecture can find and invoke services provided by the service-oriented sub-architecture via a handle that Prism-MW provides to *SDEngine*. In fact, from the pub-sub sub-architecture's perspective, services provided by the service-oriented sub-architecture are the same as the facilities provided by the middleware.

6. Discussion

Our experiences with MIDAS, which is both more heterogeneous and had more stringent requirements than other application scenarios to which Prism-MW had been applied, inspired us to reassess some of our earlier design decisions. This in turn has helped us to further understand the nature of architectural middleware. In this section we discuss some of the salient lessons we have learned.

6.1 Design of an Architectural Middleware

Our experience helped us realize that for Prism-MW's architectural facilities to be truly useful in a highly heterogeneous and resource-constrained environment, they need to be complemented with the appropriate low-level system support. This resulted in the architecture depicted in Figure 2. In turn, the separation of system from architectural concerns not only increased the flexibility and extensibility of the middleware, but also gave us more control over resource utilization and system performance.

It also became apparent 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. We already discussed some of those above: deployment, runtime analysis, adaptation, resource discovery, and so on. The common design decision behind these services has been to realize them using the architectural constructs provided by Prism-MW. This approach has a number of advantages. First, it helps to keep the middleware's core small and efficient. Second, it allows us to "recursively" reap 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 5.7) with a different implementation of it. Finally, the architectural basis of our solution allows for efficient monitoring and adaptation of the system via Prism-MW's "architectural awareness" capability.

6.2 Flexibility and Extensibility

Another observation is that there are sources of heterogeneity other than those of the underlying hardware and system software. In Section 5.4 we discussed an instance of PL-level heterogeneity. 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.

6.3 Efficiency vs. Configuration Complexity

Recall from Section 5.1 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 changes on resource usage. This level of control is 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 event queue; number of pre-allocated system resources (semaphore, mutex, file, DLL); number of pre-allocated architectural constructs (*Component*, *Connector*, *Port*); size of memory buffer used by the network sockets; and size of pre-allocated memory pool used by application-level variables. On the other hand, the Java version of Prism-MW has only two configuration "knobs": sizes of event queue and thread pool. Of course, as mentioned earlier, the Java version of Prism-MW also incurs a large overhead due to the dynamic allocation of resources. It is also unpredictable, which makes it harder to estimate and control an application's resource usage at the level of architecture.

This indicates a clear trade-off between resource utilization control and 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 application and/or the domain.

6.4 System Validation

One of the greatest challenges we have faced in the MIDAS project to date has been the validation of an application on its target platforms. It became clear early on that manual testing, debugging, and installation of the software is infeasible. Every time a bug was fixed, an updated version of the software had to be installed manually on the various devices, resulting in an extremely time consuming and redundant task. Advanced facilities, such as deployment, runtime monitoring, and analysis, proved to be essential as they automated many steps in the process.

Another reason that validation in this domain is difficult can be attributed to the fact that a virtual machine may not be able to abstract away completely the behavioral variations in the computing substrates. For example, we initially developed and tested the software targeted for the MIDAS gateways on top of Windows. We relied on

our middleware's MVM layer to insulate us from OS-level variations such as different APIs or libraries. 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 located the source of failure in the application logic, which was performing two consecutive *lock* operations on the same mutex by the same thread. We could not recreate the failure in Windows because Windows allows this, while Linux prevents it by throwing an exception.

This example also demonstrates that, as we provide more facilities in a middleware solution, it becomes harder to validate applications developed on top it. In fact, as we already hinted in the previous section, another culprit in making it harder to validate 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 fail. 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.

6.5 Advanced Facilities

Many embedded systems are long-lived and pervasive. As a result, they are constantly evolving in response to the changing environment around them. Architectural middleware for this domain should thus not only provide support for the implementation of a system in terms of its architectural elements, but also facilities that minimize the potential for architectural erosion [15] after the initial deployment. Our middleware's deployment and analysis tools are good examples of facilities that can be used to keep an architectural model synchronized with the actual system. For such tools to be useful, they must be able to represent the dynamic nature of software architectures, analyze their properties "on the fly", and configure the running system based on the results of the analysis. Our experience suggests that middleware-level architectural facilities, such as those provided in Prism-MW's, can be effective enablers of such advanced capabilities.

7. Conclusions

In this paper we have described our experience drawn from an on-going collaborative project between Bosch and USC. The novelty of our work lies in the fact that we have leveraged explicit architectural constructs in the design, analysis, implementation, deployment, and evolution of a family of embedded applications. While, our strategy has proven to be sound, several open issues remain unresolved and are a topic of our future work.

8. Acknowledgments

This material is based upon work sponsored by Bosch. The work was also sponsored by the National Science Foundation under Grant number ITR-0312780. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF. The authors wish to thank the anonymous reviewers for their helpful comments. Finally, the authors wish to express their gratitude to Christoph Stoermer for his support and guidance on this project.

9. References

[1] TAO. http://www.cs.wustl.edu/~schmidt/ACE-documentation.html

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

[3] A.Chan et al. MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing. *IEEE Transactions on Software Engineering*, Vol. 29, No.12, Dec. 03.

[4] P. Costa et al. The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems. *Int'l. Symp. on Personal Indoor and Mobile Radio Communications*, Berlin, Sep. 05.

[5] W. Emmerich. Engineering Distributed Objects. John Wiley & Sons, Chichester, UK, 2000.

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

[7] 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.

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

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

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

[11] N. R. Mehta et al. Towards a Taxonomy of Software Connectors. *ICSE*, Limerick, Ireland, June, 2000

[12] M. Mikic-Rakic, S. Malek et al. Improving Availability in Large, Distributed Component-Based Systems via Redeployment. *Int'l. Conf. on Component Deployment*, Grenoble, France, Nov. 2005.

[13] 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, May 2004.

[14] Orbix/E. www.iona.com/whitepapers/orbix-e-DS.pdf

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

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

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