Adaptable Architectural Middleware for Programming-in-the-Small-and-Many

Marija Mikic-Rakic and Nenad Medvidovic

Computer Science Department, University of Southern California
Los Angeles, CA 90089-0781 USA
{marija,neno}@usc.edu

Abstract. A recent emergence of small, resource-constrained, and highly-mobile computing platforms presents numerous new challenges for software developers. We refer to development in this new setting as programming-in-the-small-and-many (Prism). This paper provides a description and evaluation of a middleware intended to support software architecture-based development in the Prism setting. The middleware, called Prism-MW, provides highly efficient and scalable implementation-level support for the key aspects of Prism application architectures. Additionally, Prism-MW is easily extensible to support different application requirements suitable for the Prism setting. Prism-MW is accompanied with design, analysis, deployment, and run-time monitoring tool support. It has been applied in a number of applications and used as an educational tool in a graduate-level embedded systems course. Recently, Prism-MW has been successfully evaluated by a major industrial organization for use in one of their key distributed embedded systems. Our experience with the middleware indicates that the principles of architecture-based software development can be successfully, and flexibly, applied in the Prism setting.

1 Introduction

The software systems of today are rapidly growing in size, complexity, amount of distribution, and numbers of users. We have recently witnessed a rapid increase in the speed and capacity of hardware, a decrease in its cost, the emergence of the Internet as a critical resource, and a proliferation of hand-held consumer electronics devices. In turn, this has resulted in an increased demand for software applications, outpacing our ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them. One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment monitoring, freeway-traffic management, damage surveys in times of natural disaster, and so on. Such scenarios present daunting technical challenges: effective understanding of existing or prospective software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data, numbers of data types, and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute on "small" devices, characterized by highly constrained resources such as limited power, low network bandwidth, slow CPU speed, limited memory, and small display size. We refer to the development of software systems in the described setting as *pro-*

M. Endler and D. Schmidt (Eds.): Middleware 2003, LNCS 2672, pp. 162–181, 2003.

gramming-in-the-small-and-many (Prism), in order to distinguish it from the commonly adopted software engineering paradigm of programming-in-the-large (PitL) [6].

Recent studies [11,16,33] have shown that a promising approach to developing software systems in the Prism setting is to employ the principles of software architectures. *Software architectures* provide abstractions for representing the structure, behavior, and key properties of a software system [29]. They are described in terms of software *components* (computational elements) [36], software *connectors* (interaction elements) [19], and their *configurations* (also referred to as *topologies*) [18].

Software architectures provide design-level models and guidelines for composing software systems. For these models and guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation [15,28]. This is particularly important in the Prism setting: Prism systems may be highly distributed, decentralized, mobile, and long-lived, increasing the risk of architectural drift [25] unless there is a clear relationship between the architecture and its implementation.

This paper describes the design and evaluation of *Prism-MW*, a middleware developed to support the implementation of software architectures in the Prism setting. We say that the middleware is *architectural* because it provides programming language-level constructs for implementing software architecture-level concepts such as component, connector, configuration, and event. This allows software developers to directly transfer architectural decisions into implementations, thus distinguishing Prism-MW from existing middleware solutions.

Another key contribution of Prism-MW is its highly modular design that employs an extensive separation of concerns. This results in a middleware that is flexible, efficient, scalable, and extensible. The middleware is *flexible* in its support for independent selection, variation, and composition of implementation-level concerns. The middleware is *efficient* in its size, speed, and overhead added to an application. The middleware is *scalable* in the numbers of components, connectors, events, execution threads, and hardware devices. Finally, the middleware is easily *extensible* to support new development concerns and situations in the Prism setting.

These properties of Prism-MW have been successfully evaluated using a series of example applications, benchmark tests performed both within our group and by external users, and a large-scale feasibility study conducted in collaboration with an industrial organization. At the same time, our evaluations of Prism-MW have suggested several areas of improvement, including an entirely novel approach to designing architectural middleware. We intend to explore these issues in our future work.

The rest of the paper is organized as follows. Section 2 presents our objectives for Prism-MW. Section 3 presents the design and implementation of Prism-MW's core capabilities, and evaluates them with respect to the objectives. Section 4 discusses the extensibility of Prism-MW and presents several specific extensions completed to date. Section 5 describes our tool support. Section 6 presents additional evaluation of the middleware conducted in collaboration with external users. The paper concludes with overviews of related and future work.

2 Middleware Objectives

As discussed above, there are a number of significant challenges faced by software developers in the Prism setting. We believe those challenges to fall within four general

categories, comprising our objectives for Prism-MW. Three of the four objectives directly derive from the "Prism" acronym: support for *pr*ogramming (i.e., development) of Prism applications on *m*any *s*mall computing platforms. We consider these three to be the core objectives. The fourth objective reflects the variation and constant evolution of development situations in the Prism setting. Each objective is further discussed below.

- *Architectural abstractions* – A key observation guiding this research is that an effective way of supporting the development of Prism applications is to explicitly focus on software architectures [11,16,33]. *Pr*ism-MW should thus provide direct implementation-level support for architectural abstractions (components, connectors, communication events, and so on).
- *Efficiency* – Pri*s*m-MW should impose minimal overhead on an application's execution. Our current goal is to enable efficient execution of applications on platforms with varying characteristics (e.g., speed, capacity, network bandwidth). The ultimate goal is to extend this support to include efficient access to and sharing of hardware resources (e.g., battery, peripheral devices).
- *Scalability* – Pris*m*-MW should be scalable in order to effectively manage the large numbers of devices, execution threads, components, connectors, and communication events present in Prism systems.
- *Extensibility* – There are several additional capabilities that may be required for different (classes of) Prism applications. These include awareness, mobility, dynamic reconfigurability, security, real-time support, and delivery guarantees [2,3,7,11,12,24]. Prism-MW should be easily extensible to provide support for (arbitrary combinations of) these capabilities.

3 Middleware Core

In this section we discuss the design, implementation, and evaluation of Prism-MW's core capabilities. The discussion is organized around the three core objectives.

3.1 Architectural Abstractions

Prism-MW's core supports architectural abstractions by providing classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. These abstractions enable direct mapping between an architecture and its implementation. Figure 1 shows the class design view of Prism-MW. The shaded classes constitute the middleware core, with dark gray classes being relevant to the application developer. Our goal was to keep the core compact, reflected in the fact that it contains only eight classes and six interfaces. Furthermore, the design of the core (and the entire middleware) is highly modular: the only dependencies among classes are via interfaces and inheritance; the only exception is the *Architecture* class, which contains multiple *Bricks* for reasons that are explained below.

3.1.1 Middleware Core's Design

Brick is an abstract class that encapsulates common features of its subclasses (*Architecture, Component*, and *Connector*). The *Architecture* class records the configuration of its constituent components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system runtime. A distributed applica-

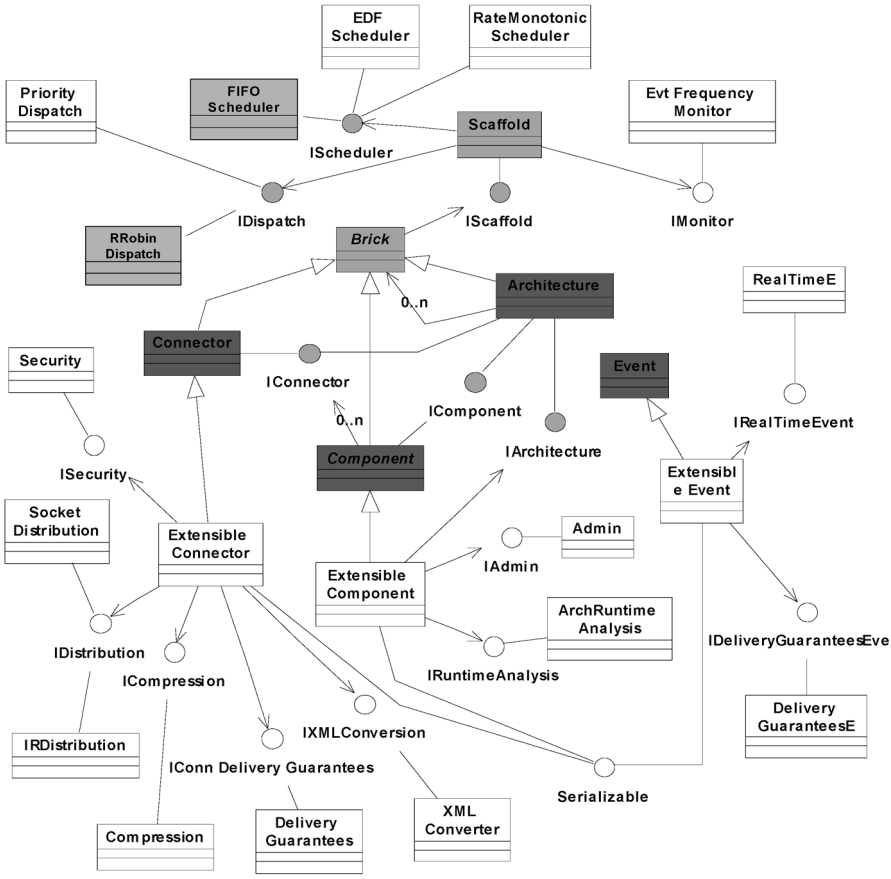

Fig. 1. UML class design view of Prism-MW. Middleware core classes are highlighted.

tion is implemented as a set of interacting *Architecture* objects. *Component*s in an architecture communicate by exchanging *Event*s, which are routed by *Connector*s. In order to support different topologies, each component may be attached to an arbitrary number of connectors. In order to support the needs of dynamically changing applications, each Prism-MW connector is capable of servicing varying numbers of components [21]. This property of connectors, coupled with event-based interaction, has proven to be a highly-effective mechanism for addressing system reconfigurability.

Each subclass of the *Brick* class has an associated interface. The *IArchitecture* interface exposes a *weld* method for attaching components and connectors to one another. The *IComponent* interface exposes *send* and *handle* methods used for exchanging events. We have implemented several versions of this interface to support asynchronous, synchronous unicast, and synchronous multicast of events. The *IConnector* interface provides a *handle* method for routing of events. To address the needs of different applications in the Prism setting, we have implemented two versions of this interface, supporting both symmetric (i.e., peer-to-peer) and asymmetric (i.e., request-response) interaction. Each *Architecture* object implements both *IConnector*

and *IComponent* interfaces, thus allowing construction of complex components and connectors with internal architectures.

Finally, Prism-MW's core associates the *IScaffold* interface with every *Brick*. Scaffolds are used to schedule events for delivery (via the *IScheduler* interface) and pool threads (via the *IDispatch* interface) in a decoupled manner. Prism-MW's core provides default implementations of *IScheduler* and *IDispatch*: FIFO and round-robin, respectively. The novel aspect of our design is that this separation of concerns allows us to select the most suitable event scheduling policy independently of the dispatching policy for a given application. 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. *IScaffold* also directly aids architectural awareness [2] by allowing probing of the runtime behavior of a *Brick*.

To date, Prism-MW's core has been implemented in Java JVM and KVM [35], C++ and Embedded Visual C++ (EVC++). Each implementation of the middleware core is quite small, averaging 1,750 SLOC, which aids Prism-MW's understandability and ease of use.[1]

3.1.2 Using Prism-MW

Prism-MW's core provides the necessary support for developing arbitrarily complex applications, so long as they rely on the provided default facilities (e.g., event scheduling, dispatching, and routing) and stay within a single address space. The first step a developer takes is to subclass from the *Component* class for all components in the architecture and to implement their application-specific methods. The next step is to instantiate the *Architecture* class and define the needed instances of thus created components, and of connectors selected from the reusable connector library.[2] Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class. This process can be partially automated using our tool support described in Section 5.

For illustration, Figure 2 shows a simple usage scenario of the Java version of Prism-MW. The application consists of two components communicating through a single connector. The *DemoArch* class's *main* method instantiates components and connectors and composes (*weld*s) them into a configuration. Figure 2 also demonstrates event-based communication between the two components. *Component A* creates and sends an event, in response to which *Component B* sends a response event. An event need not identify its recipient components; they are uniquely defined by the topology of the architecture and routing policies of the employed connectors [16].

3.2 Efficiency

Since Prism applications frequently run on resource-constrained devices, with low amounts of memory (e.g., 256 KB on the Palm Pilot) and slow processing speed, we have performed several optimizations on Prism-MW's core. While there are common techniques for ensuring efficient implementations of distributed systems, Prism-MW

[1] The interfaces used in Prism-MW's core are directly supported in Java. In C++ and EVC++ they have been implemented using abstract classes with pure virtual functions.

[2] Recall that Prism-MW's core provides several connectors through the implementations of the *IConnector* interface.

presented unique challenges in this regard because of its objective of *directly* supporting architectural abstractions in highly resource-constrained settings. Some of the optimization techniques we applied are novel, while others have been adapted from existing work. A contribution of our work on optimizing Prism-MW lies in their combination: it results in a highly efficient architectural middleware that introduces minimal overhead in terms of dynamic memory usage and shows good performance. In the remainder of this section we describe these optimizations and provide a series of benchmark results that evaluate them.

3.2.1 Initial Implementation

In our initial implementation of Prism-MW's core, each component maintained dynamically allocated queues of its incoming and outgoing events. Each component also owned an internal thread of control that was used to process incoming events and place outgoing events on the queue (as implemented in *IComponent*'s *send* and *handle* methods, respectively). The encompassing *Architecture*'s dispatcher then ensured that the outgoing events are routed to their destinations. Furthermore, the *Architecture*'s implementation of the *IScheduler* interface was trivial since all the scheduling was handled at the individual component level. However, this implementation had several problems, including unacceptable application size and speed. Prism-MW's highly modular design allowed us to significantly improve efficiency by radically altering the manner in which events are exchanged and processed. At the same time, we were able to confine our modifications to the implementations of *IComponent* (specifically, its *send* method), *IScheduler*, and *IDispatch*. These modifications are discussed below.

Architecture initialization

```
class DemoArch {
   static public void main(String argv[]) {
   Architecture arch = new Architecture ("DEMO ");
     // create components here
     ComponentA a = new ComponentA ("A");
     ComponentB b = new ComponentB ("B");

     // create connectors here
     Connector conn = new Connector("Conn");

     // add components and connectors to the architecti
     arch.addComponent(a);
     arch.addComponent(b);
     arch.addConnector(conn);

     // establish the interconnections
     arch.weld(a, conn);
     arch.weld(b, conn);
     arch.start();
   }
}
```

Component A sends an event

```
e = new Event ("Event_a");
e.addParameter("param_1", p1);
send (e);
```

Component B handles the event and sends a response

```
public void handle(event e)
{
   if (e.equals("Event_a")) {
     ...
     event e1= new Event("Response_to_a");
     e1.addParameter("response", resp);
     send(e1);
   }...
}
```

Fig. 2. Illustration of application implementation fragments.

3.2.2 Optimizing for Size and Speed

We observed that a large amount of dynamic system memory usage was a result of the exchange of events among components and connectors. We minimized the required memory for event passing by exchanging read-only events in the same address space

by reference, rather than by copy. We further optimized memory usage by adopting a fixed-sized, circular array for storing all events in a single address space. This reduced overall memory usage by a factor of 20 or more over the initial solution described above [15].

Another modification addressed event processing. A pool of shepherd threads (implemented in Prism-MW core's *RoundRobinDispatcher* class) was introduced to handle events sent by any component in a given address space. The size of the thread pool is parameterized and, hence, adjustable. It should be noted that the concurrency management of the circular array used to implement the event queue slightly impacts the speed of processing by applying a producer-consumer algorithm to keep event production under control, and supply shepherd threads with a constant stream of events to process.

To process an event, a shepherd thread removes the event from the head of the queue. For local communication, the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread (see Figure 3). If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. An alternative design, which required modification of only the *IDispatch* interface's implementation, allows separate threads to be used for dispatching an event from the connector to each intended recipient component (steps 3-6 in Figure 3). This increases parallelism, but also resource consumption, in the architecture. We are currently implementing and evaluating this design.

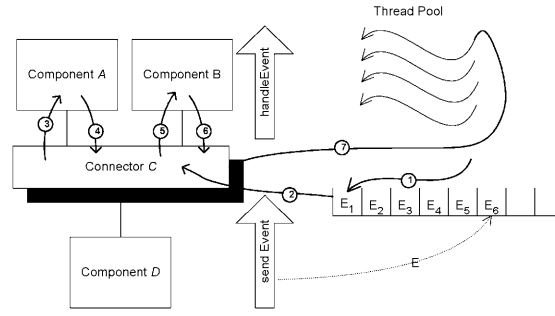

Fig. 3. Event dispatching in Prism-MW for a single address space. Steps (1)-(7) are performed by a single shepherd thread.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: a shepherd thread transports the event from the queue to its recipients via a connector. However, in this case the connector is a specialized *DistributionConnector* (further discussed in Section 4.1), which manages a set of network (e.g., socket or infrared) connections. Thus, instead of routing the event through the components attached to the connector (steps 3-6 in Figure 3), the shepherd thread simply deposits the event on all communication ports managed by the *DistributionConnector*. As the event is propagated across the network, the *DistributionConnector* on each recepient device uses its internal thread to retrieve the incoming event from the communication port and place it on its local event queue.

This solution represents an adaptation of an existing worker thread pool technique [31] that results in several unique benefits:

1. By leveraging explicit architectural topology an event can be routed to multiple destinations using a single shepherd thread. This minimizes resource consumption, since events need not be tagged with their recipients;

2. We further optimize resource consumption by using a single event queue for storing both locally and remotely generated events; and
3. Since Prism-MW does not process local and remote events differently, and all routing is accomplished via the multiple and explicit software connectors, Prism-MW also allows for easy redeployment and redistribution of existing applications onto different hardware topologies.

3.2.3 Evaluation

The above optimizations have resulted in very light-weight Prism-MW implementations that have shown several orders of magnitude in performance improvement over the original implementation described above. More importantly, the performance of Prism-MW is now comparable to solutions using a plain programming language (PL): each Prism-MW event exchange causes five PL-level method invocations and, a comparatively much more expensive, context switch if the architecture is instantiated with more than one shepherd thread (roughly corresponding to steps 1-4 and 7 in Figure 3); analogous functionality would be accomplished in a PL with two invocations and, assuming concurrent processing is desired, a context switch. It should also be noted that it is unlikely that a plain PL could support a number of development situations for which Prism-MW is well suited (e.g., asynchronous event multicast) and due to which it introduces its performance overhead in the first place.

For illustration, we describe the results from one series of evaluations used to measure the size and performance of the Java Prism-MW implementation.[3] The benchmarking applications consisted of n (n = 1, 10, and 50) identical components communicating via a connector with a single component, but not with each other (Figure 3 shows such a scenario for n = 2). The applications used a pool of 10 shepherd threads and a queue of 1000 events (*q_size*). Between 1 and 100,000 simple (parameter-less) events were sent asynchronously by the single component to the n components, resulting in between 1 and 5,000,000 handled events for the three applications. The results of this benchmark are shown in Figure 4.

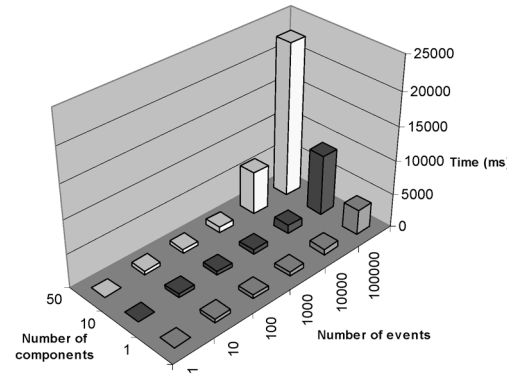

Fig. 4. The results of the performance benchmark.

Memory usage of Prism-MW core (*mw_mem*), recorded at the time of architecture initialization, is 4.6 KB. The overhead of a "base" Prism component (*comp_mem*), without any application-specific methods or state, is 0.8 KB. Memory overhead of creating and sending a single event (*evt_mem*) can be estimated using the following formula, obtained empirically:

evt_mem (in KB) = 0.16 + 0.24 * num_of_parameters

[3] The benchmarks presented throughout the paper were performed on an Intel Pentium III 700 MHz processor with 256 MB of RAM running JDK 1.1.8 on Microsoft Windows 2000.

The formula assumes that the parameters do not contain complex objects, but may contain simple objects (e.g., Java Integer or String).[4] Therefore, for example, the maximum memory overhead (assuming the event queue is full) induced by using Prism-MW in the largest benchmark application described above is approximately

mw_mem + num_comps * comp_mem + q_size * evt_mem =
4.6 + (51 * 0.8) + (1000 * (0.16 +(0.02 * 0))) ≈ 205 KB

3.3 Scalability

Prism-MW's modularity and separation of concerns directly aid its scalability in the numbers of supported devices, components, connectors, threads, and events. Prism-MW's support for large numbers of devices is a consequence of its support for large numbers of connectors. Similarly, its scalability in the number of events is fostered by scalability in the number of threads. The below discussion reflects these relationships.

3.3.1 Connectors and Devices

Unlike the existing middleware platforms (e.g., CORBA [40], LIME [12], .NET [20]), which support a single, implicit connector in a system, Prism-MW supports an arbitrary *number of connectors*. Prism-MW's explicit, flexible connectors allow an architecture to be deployed onto an arbitrary number of hosts, by repeated splitting of the connectors using the technique described in [5]. In a highly degenerate case, this would result in some devices serving only as routers, without containing any components. For this reason, the *number of devices* supported by Prism-MW is unlimited in principle. It should be noted, however, that the deployment choice directly affects efficiency: the performance gain of using the centralized event queue is achieved only if the components are residing in the same address space.

3.3.2 Components

Realistically, the *number of components* on a given device is limited and can be estimated using the following simple formula: n = (M - MS) / ACS, where *M* is the available memory on the device, *MS* is the memory occupied by Prism-MW, and *ACS* is the average component size. Recall from Section 3.2 that the impact of MS and the middleware-induced portion of ACS on the device's memory consumption is very low. We have performed a series of benchmarks in order to assess the behavior of Prism-MW in cases where large numbers of components is used.[5] Figure 5 shows the

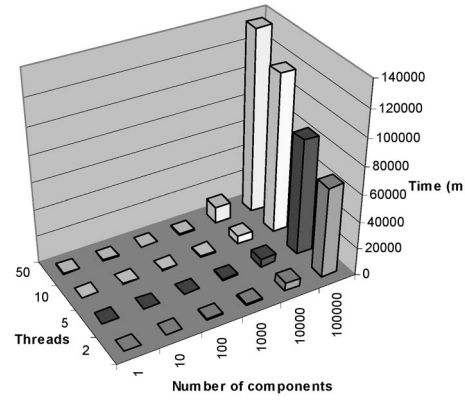

Fig. 5. The results of the scalability benchmark.

[4] In this sense, the measure represents minimum event overhead. Use of complex objects as event parameters is independent of the middleware, but is an application-level decision.

[5] We have run benchmark tests with up to 1,000,000 components.

results of a test in which a varying number of identical components (n) communicated with a single component through a connector. 100 parameter-less events were sent asynchronously by the single component to all other components, resulting in $n*100$ events being handled. We have also performed a benchmark test with two end-point components that communicated using a "chain" consisting of 100,000 components that simply forward incoming events through 100,001 connectors. The total round-trip time for a single event was 2.7 seconds.

3.3.3 Threads and Events

Prism-MW supports as many *threads* as the underlying platform supports. Finally, the *number of events* supported by Prism-MW is not limited by the middleware itself, but by the properties of the underlying hardware platform. This limit can be characterized by the following two parameters: (1) the maximum number of events that can simultaneously be present in a system and (2) the rate of event delivery. The maximum number of events is limited by the available memory on a given host (or set of hosts) and event size (recall Section 3.2), while the rate of event delivery depends on the CPU speed, the number of threads servicing the event queue, the ratio of event production to consumption by the components, and the network bandwidth for events that traverse machine boundaries.

4 Middleware Extensions

The design of Prism-MW's core provides extensive separation of concerns both via its explicit architectural constructs and its pervasive use of interfaces. The design is highly extensible. The unshaded classes and interfaces in Figure 1 show various extensions to the Prism-MW core we have built to date. These include support for architectural awareness, real-time, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility [2,3,7,11,12,24]. In this section we describe our approach to supporting these extensions. Our experience indicates that other extensions can be easily added to the middleware in the same manner.

Our support for extensibility is built around the objective that Prism-MW's core remains unchanged. Instead, the core constructs (*Component*, *Connector*, and *Event*) are subclassed via specialized classes (*ExtensibleComponent*, *ExtensibleConnector*, and *ExtensibleEvent*), each of which composes a number of interfaces. Each interface can have multiple implementations, thus enabling selection of the desired functionality inside each instance of a given *Extensible* class. If an interface is installed in a given class instance, that instance will exhibit the behavior realized inside the interface's implementation. Multiple interfaces may be installed in a single *Extensible* class instance. In that case, the instance will exhibit the combined behavior of the installed interfaces.

Below we describe four classes of extensions supported by Prism-MW, with an explicit focus on the extensions we have completed to date. Further details on these extensions may be found in [21]. With the exception of Prism-MW's support for distribution (see below as well as Sections 5.2 and 6.3), we do not discuss the efficiency aspects of these extensions for two reasons. First, our primary goal to date has been to assess the extensibility of Prism-MW, and we have not optimized our implementations of many of its extensions. Secondly, in most cases our implementations employed

known algorithms and techniques, such that any performance measures would be a function of those algorithms and techniques rather than the inherent properties of Prism-MW.

4.1 Connector Extensions

In order to address different aspects of interaction the *ExtensibleConnector* class composes a number of interfaces that support various interaction services. In turn, each interface can have multiple implementations.

Figure 1 shows five different interfaces we have implemented thus far. The *IDistribution* interface has been implemented in two classes, one supporting socket-based and the other infrared port-based inter-process communication (IPC). We refer to an *ExtensibleConnector* with instantiated *IDistribution* interface as *DistributionConnector*. A single *DistributionConnector* can be attached to an arbitrary numbers of remote hosts, as well as local components and connectors. Similarly to the "base" Prism-MW connector discussed in Section 3.1.1, a *DistributionConnector* is capable of supporting runtime addition and removal of local components and connectors, as well as remote devices. The base size of the (more frequently used) socket-based *DistributionConnector* is 1.27 KB. In addition to this, each socket connection adds 2.7 KB on average. Finally, the PL's support for IPC introduces additional overhead. In Java this overhead is 9.5 KB for loading the `java.net` package.

The *ISecurity* interface has several implementations that perform combinations of authentication, authorization, encryption, and event integrity. These services are implemented using three major cryptographic algorithms: symmetric key, asymmetric key, and event digest function. The *IConnDeliveryGuarantees* interface supports event delivery guarantees. We have implemented this interface to support at most once, at least once, exactly once, and best effort delivery semantics. In order to support communication across PLs, we have added the *IXMLConversion* interface and implemented XML encoding/decoding of events inside the *XMLConverter* class. Finally, we have added the *ICompression* interface with the goal of minimizing the required network bandwidth for event dispatching. To this end, we have implemented the Huffman coding technique [27] inside the *Compression* class.

Addition of a new interface to the *ExtensibleConnector* requires adding a pointer to the interface and performing method calls on it inside *ExtensibleConnector*'s *handle* method. The change to the *ExtensibleConnector* class is minimal, averaging three new lines of code for each new interface. However, it is important to know the right ordering of method calls to achieve the desired behavior. For example, when combining *ISecurity* and *IXMLConversion* interfaces, *IXMLConversion*'s *convert* method is invoked before *ISecurity*'s *encrypt* method when sending the event; on the receiving end, the *ISecurity*'s *decrypt* method is invoked before *IXMLConversion*'s *reconstitute* method.

The overhead introduced by this solution is that an *ExtensibleConnector* instance may have many *null* pointers, corresponding to interfaces that have not been installed. The values of these pointers will be checked each time the *handle* method is invoked. An alternative solution, which would trade-off the extensibility for efficiency, is to subclass the *Connector* class directly and to have the references only to the desired interfaces. We are planning to implement a tool that would perform this task automatically, given a specification of features that a connector should support.

4.2 Component Extensions

To support various aspects of architectural awareness and middleware-level reflection, we have provided the *ExtensibleComponent* class that composes several interfaces. Additionally, *ExtensibleComponent* contains a reference to *IArchitecture*, allowing its instances to act as meta-level components and to effect runtime changes on the system's architecture. To date, we have augmented the *ExtensibleComponent* class with two interfaces. The *IAdmin* interface is used for performing component deployment and mobility (see Section 5.2). The *IRuntimeAnalysis* interface is used for analyzing the architectural descriptions and assessing proposed architectural changes during the application's execution. We have recently implemented several versions of this interface that encapsulate different subsets of our DRADEL [17] environment (see Section 5.1).

4.3 Event Extensions

To support various facets of event delivery we have provided the *ExtensibleEvent* class that can compose multiple interfaces. To date, we have created three interfaces inside the *ExtensibleEvent* class. The *IDeliveryGuaranteesEvent* interface is used to assign a delivery guarantee policy to an event (i.e., at most once, at least once, exactly once, best effort). This interface is used in tandem with the *IConnDeliveryGuarantees* interface of the *ExtensibleConnector* class. The *IRealTimeEvent* interface is used to assign a real-time deadline to an event. We have implemented this interface to support both aperiodic and periodic real-time events. In support of real-time event delivery we have additionally provided three classes that implement the *IScheduler* and *IDispatch* interfaces, discussed below. Finally, to support communication across PL boundaries the *IXMLRepresentation* interface provides XML-based representation of an event.

4.4 Other Extensions

In addition to the *IDistribution* interface inside the *ExtensibleConnector* class, to support distribution and mobility we have implemented the *Serializable* interface inside each one of the *Extensible* classes. This allows us to send data as well as code across machine boundaries.

In support of real-time event delivery we have provided two additional implementations of the *IScheduler* interface. *EDFScheduler* implements scheduling of aperiodic events based on the earliest-deadline-first algorithm, while *RateMonotonicScheduler* implements scheduling of periodic events.

5 Tool Support

We augment Prism-MW with tools for architectural modeling, analysis, deployment, and run-time monitoring and evolution. These tools themselves have been implemented using Prism-MW. As such, the tools provide additional evaluation of Prism-MW.

5.1 Modeling and Analysis

In adding support for architecture modeling and analysis to Prism-MW, we have integrated xADL 2.0 [4], a highly extensible XML-based architecture description language (ADL) [18], into our existing analysis environment, DRADEL [17]. Our support for architectural description is reasonably general: we model interacting components simply as collections of provided and required services whose semantics are represented in first-order logic. DRADEL's *TopologicalConstraintChecker* and *TypeChecker* components use this information to ensure architectural consistency. While the details of DRADEL have been reported elsewhere [17], DRADEL is relevant in this context because it has been reengineered using Prism-MW: it consists of nine components and four connectors, comprising 13,000 Java SLOC, not counting Prism-MW itself. Furthermore, DRADEL's *CodeGenerator* component generates Prism-MW compatible application skeletons from xADL descriptions, directly aiding the transfer of architectural decisions into application code.

5.2 Deployment and Run-Time Monitoring

Our support for deployment and run-time monitoring directly leverages Prism-MW's services. We have integrated and extended the COTS MS Visio tool to develop Prism-DE, the deployment environment for Prism applications, shown in Figure 6. Prism-DE contains several toolboxes (left side of Figure 6). The top toolbox enables an architect to specify a configuration of hardware devices by dragging their icons onto the canvas and connecting them. The next toolbox enables the specification of processes that will be executing on each device. The remaining toolboxes supply the software components and connectors that may be placed inside the processes. The *Connectors* toolbox is populated with connector types that represent various combinations of *ExtensibleConnector* interface implementations we have built to date. The *Components* toolboxes have to be populated with application components for each new application. This task only requires specifying the location of each component's implementation (either a collection of Java classes or a C++ DLL). Prism-DE actively analyzes the specified configurations, ensuring that each architectural element has a container process and a valid instance name, and that C++ modules are not be in the same process as Java modules. Additionally, Prism-DE contains a pluggable DRADEL *TopologicalConstraintChecker* to ensure conformance of a desired set of topological rules. Our future goal is to integrate DRADEL's entire modeling and analysis capabilities inside Prism-DE.

Once a desired software configuration is created in Prism-DE, it can be deployed onto the depicted hardware configuration with a simple button click. In order to deploy the desired architecture on 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 a *DistributionConnector* (recall Section 4.1) and an *ExtensibleComponent* with instantiated *IAdmin* interface (referred to as *AdminComponent* below), that is attached to the connector. The skeleton configuration is extremely lightweight. For example, in our Java implementation, the skeleton uses under 11 KB of dynamic memory.[6] Since Prism-MW itself, the *Architecture* object, and *Distribu-*

[6] This figure does not include the additional overhead discussed in Section 4.1: 9.5 KB needed to load the `java.net` package and 2.7 KB per socket connection.

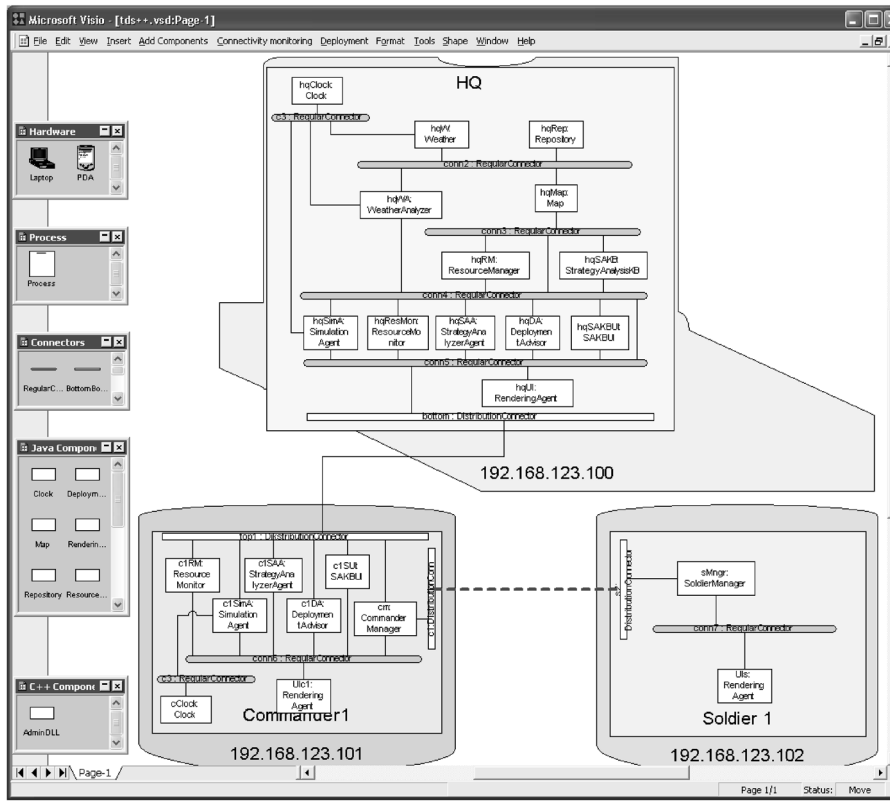

Fig. 6. *The Prism-DE* deployment and run-time monitoring environment.

tionConnector are also used at the application level, the actual memory overhead of our basic deployment support (i.e., the *Admin Component*) is only around 5 KB.

As shown in Figure 1, the *ExtensibleComponent* on each device contains a pointer to its *Architecture* object and is thus able to effect run-time changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors with the help of *DistributionConnector*s. *Admin Component*s are able to send and receive from any device to which they are connected the events that contain application-level components (sent between address spaces using the *Serializable* interface).

Prism-DE supports run-time monitoring of connectivity between application processes. If communication between two *DistributionConnector*s is disabled for any reason (e.g., failed connection or failed container process), that information is propagated via an event to Prism-DE, which, in turn, highlights the disconnection (dotted line in Figure 6). A future enhancement to the run-time monitoring aspects of Prism-DE will include discovery of alternate paths between the disconnected nodes and automatic reconfigurations to enable their continued communication.

6 Further Evaluation and Experience

Over twenty applications have been implemented using Prism-MW to date, involving traditional desktop platforms, PalmOS- and WindowsCE-compatible devices, digital cameras, and motion sensors. Several of these applications were developed in the context of three graduate-level courses at USC. They include distributed digital image capture and processing, map visualization and navigation, location tracking, and instant messaging for hand-held devices. We do not provide additional details of these applications here due to space constraints; several of them are described in [23]. Instead, in this section we focus on experiences resulting from a project involving multiple teams of graduate students and our collaborations with two industrial organizations. The student-led project assessed Prism-MW's ease of use and resulted in different implementations of a dynamic service discovery capability. The first external collaboration resulted in a large-scale military application in support of one organization's specific needs in the ground vehicle domain. The second collaboration resulted in an extensive further evaluation of Prism-MW in the context of the other organization's distributed airborne system.

6.1 Dynamic Service Discovery

Eight teams, each consisting of three graduate students, were tasked with developing a distributed application, called dynamic service discovery (DSD), using the Java implementation of Prism-MW. The application was to be deployed on a set of Compaq iPAQ PDAs running WindowsCE and connected into a wireless LAN. In DSD, each host provides and requires a set of services. The goal of the application is to satisfy the greatest number of service requests in the shortest amount of time given the below requirements. In order to allow the students to focus on the important aspect of the project, the services were simple arithmetic and trigonometric operations provided by Java (e.g., +, –, *sin*, *cos*, and so on).

DSD assumes that the sets of provided and required services will vary across hosts. Furthermore, the sets of provided and required services on each host may change at any time. Each host is connected to and has access to only a subset of other hosts. However, a host may use one of its neighbors as a "relay" to indirectly access the desired host. The connectivity among the hosts may be altered at any time during the application's execution. New hosts may enter the network at any time, while existing hosts may leave and reenter the network at any time.

The eight student teams implemented DSD's requirements by extending the Prism-MW with implementations of IDistribution interface that allows monitoring of the network for new devices and for changes in connectivity among existing devices. Each team also implemented meta-level components in support of the varying set of services requested.

DSD was a reasonably simple application, but one that had some interesting properties representative of the Prism setting. The functionality described above was developed over a ten-week period. While just under one half of the students had exposure to Java prior to starting the project, only one student was somewhat familiar with a previous version of Prism-MW, four students had some experience with component-based software development, three had experience with developing for WindowsCE, and none had any experience with the Compaq iPAQ. The students were asked to estimate

the amounts of time spent on the various aspects of the project, including "*Understanding/Learning Prism-MW*". The teams reported that this aspect of the project required between 8% and 21% of the total project time, with 15% being the average. While many additional case studies are required to draw definitive conclusions, we view these results as indicative of Prism-MW's understandability, particularly in light of the fact that the students were not only asked to use Prism-MW, but also to *modify* it, as described above.

6.2 Military Deployment

Figure 7 depicts the application for distributed military troops deployment and battle simulations (TDS). A computer at *Headquarters* gathers information from the field and displays the current battlefield status: the locations of friendly and enemy troops, vehicles, and obstacles such as mine fields. The headquarters computer is networked via secure links to a set of PDAs used by *Commander*s in the field. The commander PDAs are connected directly to each other and to a large number of *Soldier* PDAs. Each commander is capable of controlling his own part of the battlefield: deploying troops, analyzing the deployment strategy, transferring troops between commanders, and so on. In case the *Headquarters* device fails, a designated *Commander* assumes the role of *Headquarters*. Soldiers can only view the segment of the battlefield in which they are located, receive direct orders from the commanders, and report their status. Figure 6 shows the partial architecture of TDS consisting of single *Headquarters*, *Commander*, and *Soldier* subsystems, while Figure 7 shows one possible deployment with single *Headquarters*, four *Commander*s, and 36 *Soldiers*.

Fig. 7. TDS application.

TDS has provided an effective platform for demonstrating a number of Prism-MW services and assessing its scalability in a real application setting. TDS has been designed, analyzed, implemented, deployed, monitored, and dynamically evolved using the techniques described in this paper. It has been implemented in four dialects of two programming languages: Java JVM and KVM, C++ and EVC++, with on-going plans to integrate it with legacy software implemented in Ada. TDS has been deployed to 105 mobile devices and mobile device emulators running on PCs, with plans for further scaling it up to 1,000 devices. The dynamic size of the application is approximately 1 MB for the *Headquarters* subsystem, 600 KB for each *Commander*, and 90 KB for each *Soldier* subsystem. The devices on which TDS has been deployed are of several different types (Palm Pilot Vx and VIIx, Compaq iPAQ, HP Jornada, NEC

MobilePro, Sun Ultra, PC), running four OSs (PalmOS, WindowsCE, Windows 2000, and Unix). The performance of TDS has been acceptable, easily surpassing user reaction time after the initial delay caused by application deployment. We are currently in the process of designing tests to quantify that performance.

6.3 Airborne System

In order to assess the maturity and suitability of Prism-MW for use in one of their key distributed airborne systems, our second industrial collaborator conducted a series of benchmark tests. The tests were designed to be representative of usage scenarios in the reference system's existing implementation. Once it was established that application speed using Prism-MW was satisfactory,[7] our collaborator became particularly interested in the overhead induced on application size by Prism-MW. One example test involved exchanging 100,000 records of proprietary structure (totalling over 13 MB) between Prism-MW components distributed over a LAN (i.e., using Prism-MW's *DistributionConnector*s discussed in Section 5.2). The base, unoptimized implementation of Prism-MW resulted in a 20% increase of the amount of exchanged data in comparison to the reference implementation. A relatively simple specialization of the *DistributionConnector* class (modifying the implementation of two methods of the *IDistribution* interface), without any other modifications to the middleware, reduced that overhead down to 5%. As a result, our collaborator has deemed Prism-MW "very efficient and flexible" and is planning on adopting it.

7 Related Work

Our work on Prism-MW has been primarily influenced by two research areas: architectural styles and middleware. Architectural styles were discussed in the Introduction. Below we discuss two most closely related approaches in the middleware arena. Additionally, we briefly discuss a preliminary comparison of Prism-MW with several representative middleware solutions.

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 Prism 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 [33] is an architectural style and supporting middleware for ubiquitous computing applications with special focus on user mobility, context awareness, and context switching. Aura is only applicable to certain classes of applications in the Prism setting. Similarly to Prism-MW, Aura has explicit, first-class connectors. Aura also provides a set of components that perform management of tasks, environment monitoring, context observing, and service supplying. This suggests that the Aura style could be successfully supported using Prism-MW augmented with a set of Aura-specific extensions. This would eliminate the need for performing optimizations of Aura's current

[7] We were not appraised of the details of the tests assessing the application speed, only of their outcome.

implementation support, which has to date only been tested on traditional, desktop platforms.

We have performed a preliminary comparison of Prism-MW with several representative middleware solutions with respect to the objectives identified in Section 2. The results of these comparisons are shown in Table 1.[8] TAO and Orbix/E do well in supporting scalability,[9] security, and delivery guarantees, but do so at the expense of the middleware size. Jini, .NET, XMIDDLE, RCSM, and LIME do well in supporting awareness and mobility, while all of them lack support for delivery guarantees. Finally, none of the representative middleware solutions support explicit architectural abstractions, thus clearly distinguishing them from Prism-MW .

8 Conclusions and Future Work

This paper presented the design, implementation, and evaluation of Prism-MW, a middleware targeted at applications in highly distributed, resource constrained, heterogeneous, and mobile settings. The key properties of the middleware are its native, and flexible, support for architectural abstractions, efficiency, scalability, and extensibility. These properties were enabled by Prism-MW's extensive separation of concerns that spans several dimensions:

Table 1: Comparison of existing middleware solutions. **?** denotes unavailable data; ✓✓✓ denotes extensive support; ✓✓ denotes solid support; ✓ denotes some support; empty cells denote no support.

Property		*Orbix/E* [10]	*TAO* [32]	*JXTA* [26]	*.NET* [20]	*JINI* [34]	*XMIDDLE* [14]	*RCSM* [39]	*LIME* [12]	*Prism-MW*
Architectural abstractions										✓✓✓
Efficiency[a]		16,6K	8K	?	?	?	?	?	?	20K
		95KB	0.5MB	?	?	?	156KB	?	?	4.6KB
Scalability		✓✓	✓✓	✓✓	✓✓	✓✓	?	?	?	✓✓✓
Extensibility	*Awareness*		✓	✓	✓✓	✓✓	✓	✓✓	✓✓	✓✓
	Delivery guarantees	✓	✓✓✓							✓✓
	Mobility					✓✓	✓✓✓	✓✓	✓✓✓	✓✓
	Reconfig.		✓✓			✓✓		✓✓	✓✓	✓✓
	Security	✓✓	✓✓	✓✓	✓✓	✓		✓✓		✓✓

a. *Number of events per second (top) and memory usage (bottom).*

- By adopting an explicit architectural perspective, Prism-MW has inherited the separation of computation (handled by components) from interaction (handled by connectors) intrinsic to software architectures.
- Furthermore, Prism-MW's extensive use of interfaces and complete lack of direct dependencies among its classes also allows tailoring implementation-level concerns (e.g., the ability to select different schedulers independently of dispatchers or to compose distribution, XML encoding, and compression facilities for network-based interactions).

[8] The results of performance benchmarks are taken from the available online documentation. The hardware platforms on which these benchmarks were ran are comparable, but the OSs and PLs used are different. However, since both OrbixE and TAO are implemented in C++ running on Linux, we expect that their performance results would not significantly improve when run on Windows2000 using Java (the test platform for Prism-MW).

[9] Recall from Section 3.3 that an aspect of the existing middleware platforms which hampers their scalability is their support for only one software connector.

- The middleware also separates an application's conceptual architecture from its implementation: each component may be implemented in multiple PLs; those implementations are fully interchangeable if *ExtensibleConnectors* with the appropriate implementations of the *IXMLConversion* interface are used.
- Finally, the Prism-DE environment enables the complete separation of an application's architecture from its deployment.

In turn, this separation of concerns across multiple dimensions enables easy selection and tailoring of the exact middleware features needed for each development situation in the Prism setting.

While our experience thus far has been very positive, a number of pertinent issues remain unexplored. One such issue is the role Prism-MW may play in supporting different architectural styles (e.g., client-server, push-based, peer-to-peer) [25,29], perhaps even in the same application. We are also in the process of further evaluating Prism-MW by applying it in the mobile robotics domain in collaboration with USC's Center for Robotics and Embedded Systems. Our future work will span issues such as adding configuration management support to Prism-MW and automatically generating an optimized version of Prism-MW given a desired set of features (i.e., eliminating the need to store and check interface pointers even when they are not used in a given Prism-MW class implementation). Another alternative we are considering to address the latter problem is to parameterize Prism-MW's variation points instead of using interfaces. We are not aware of any comparable attempts at parameterizing middleware to this extent, and consider this to be an interesting research challenge.

References

1. J.Aldrich, C. Chambers, D. Notkin. ArchJava: Connecting Software Architecture to Implementation. International Conference on Software Engineering 2002, Orlando, Florida, May 2002.
2. L. Capra, W. Emmerich and C. Mascolo. Middleware for Mobile Computing. UCL Research Note RN/30/01.
3. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems, 19(3), August 2001.
4. E. Dashofy, A. Hoek, and R. N. Taylor. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. International Conference on Software Engineering 2002, Orlando, Florida, May 2002.
5. E. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. International Conference on Software Engineering'99, Los Angeles, May 1999.
6. F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. IEEE Transactions on Software Engineering, June 1976.
7. W. Emmerich. Software Engineering and Middleware: A Roadmap. In The Future of Software Engineering, ACM Press 2000.
8. R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D Thesis, UCI, June 2000.
9. M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. Joint European Software Engineering Conference (ESEC) and Foundations of Software Engineering (FSE) '99, September 1999.
10. IONA Orbix/E Datasheet. http://www.iona.com/whitepapers/orbix-e-DS.pdf
11. E. A. Lee. Embedded Software. Revised from UCB/ERL Memorandum M01/26, UC Berkeley, CA, November 1, 2001.
12. LIME http://lime.sourceforge.net/

13. T. Lindholm and F. Yellin. The Java Virtual Machine Specification. 2nd Edition Java Series. Addison Wesley 1999.
14. C. Mascolo et. al. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. To appear in Personal and Wireless Communications, Kluwer.
15. N. Medvidovic, N. R. Mehta, M. Mikic-Rakic: A Family of Software Architecture Implementation Frameworks. The Working IEEE/IFIP Conference on Software Architecture 2002, Montreal, Canada, August 2002.
16. N. Medvidovic and M. Mikic-Rakic. Architectural Support for Programming-in-the-Many. TR USC-CSE-2001-506.
17. N. Medvidovic, et al. A Language and Environment for Architecture-Based Software Development and Evolution. International Conference on Software Engineering '99, Los Angeles, CA, May 1999.
18. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, vol. 26, no. 1, pages 70-93 (January 2000). Reprinted in Rational Developer Network: Seminal Papers on Software Architecture. Rational Software Corporation, (July 2001).
19. N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. International Conference on Software Engineering (ICSE 2000), pages 178-187, Limerick, Ireland, June 4-11, 2000.
20. Microsoft.NET. http://www.microsoft.com/net/
21. M. Mikic-Rakic and N. Medvidovic. A Connector-Aware Middleware for Distributed Deployment and Mobility. ICDCS Workshop on Mobile Computing Middleware, Rhode Island, May, 2003.
22. M. Mikic-Rakic and N. Medvidovic. Middleware for Software Architecture-Based Development in Distributed, Mobile, and Resource-Constrained Environments. TR USC-CSE-2002-501.
23. M. Mikic-Rakic and N. Medvidovic. Software Architecture-Based Development Support for Ubiquitous Systems. TR USC-CSE-2002-508.
24. P. Oreizy, et al. An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems and Their Applications, 14(3), May/June 1999.
25. D. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes, October 1992.
26. Project JXTA. http://www.jxta.org/
27. D. Salomon. Data Compression: The Complete Reference. Springer Verlag, December 1997.
28. M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 21(4), April 1995.
29. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
30. D. Schmidt. ACE. http://www.cs.wustl.edu/~schmidt/ACE-documentation.html
31. D. Schmidt et. al. Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers. Kluwer Journal of Realtime Systems, Volume 21, Number 2, 2001.
32. D. Schmidt. TAO. http://www.cs.wustl.edu/~schmidt/TAO.html
33. J. P. Sousa, and D. Garlan: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. The Working IEEE/IFIP Conference on Software Architecture 2002 2002, Montreal, Canada, August 2002.
34. Sun Microsystems. JINI(TM) Network technology. http://wwws.sun.com/software/jini/
35. Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm.
36. C. Szyperski. Component Software – Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, 1998
37. R.N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, June 1996.
38. The x-kernel Protocol Framework. http://www.cs.arizona.edu/xkernel/
39. S. S. Yau and F. Karim, Context-Sensitive Middleware for Real-time Software in Ubiquitous Computing Environments. Proceedings of the International Symposium on Object-oriented Real-time distributed Computing 2001, Magdeburg, Germany.
40. A Discussion of the Object Management Architecture (OMA) Guide, OMG, 1997.