A Style-Aware Architectural Middleware for Resource-Constrained, Distributed Systems

Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic, *Member*, *IEEE*

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 Prism-MW, a middleware platform intended to support software architecture-based development in the Prism setting. Prism-MW provides efficient and scalable implementation-level support for the key aspects of Prism application architectures, including their architectural styles. Additionally, Prism-MW is extensible to support different application requirements suitable for the Prism setting. Prism-MW has been applied in a number of applications and used as an educational tool in graduate-level software architecture and embedded systems courses. 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.

Index Terms—Software architecture, architectural style, middleware, Prism-MW.

1 Introduction

Software systems are continuously growing in size and complexity. In recent years, they have also increasingly migrated from the traditional, desktop setting to highly distributed, mobile, possibly embedded and pervasive computing environments. Such environments 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 ***pr**ogramming-**i**n-the-**s**mall-and-**m**any* (*Prism*), both for exposition purposes and also in order to distinguish it from the traditional software engineering paradigm of ***p**rogramming-**i**n-**t**he-**l**arge* (*PitL*) [7], which has been primarily targeted at desktop computing.

Software engineering researchers and practitioners have successfully dealt with the increasing complexity of PitL systems by employing the principles of *software architecture*. Software architecture provides design-level models and guidelines for composing the structure, behavior, and key properties of a software system [28]. An architecture is described in terms of software *components* (computational elements) [39], software *connectors* (interaction elements) [21], and their *configurations* (also referred to as *topologies*) [20]. A given system's architecture will adhere to one or more architectural styles. An *architectural style* codifies architectural composition guidelines that are likely to result in software systems with certain desired properties [28]. Examples of widely used styles are event-based, client-server, pipe-and-filter, peer-to-peer, blackboard, etc.

Given the central role software architectures and architectural styles have played in the PitL setting, we expect that their importance will only grow in the even more complex Prism setting. This is corroborated by the preliminary results from several recent studies of software architectural issues in embedded, mobile, and ubiquitous systems [13], [18], [36]. In order for architectural models and stylistic guidelines to be truly useful in any development setting, they must be accompanied by support for their implementation [17], [32]. This is particularly important in the Prism setting: Prism systems may be highly distributed, decentralized, mobile, and long-lived, thus increasing the risk of architectural drift [28] 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. The middleware is tailorable to provide native implementation-level support for arbitrary architectural styles. This allows software developers to transfer directly architectural decisions into implementations, thus distinguishing Prism-MW from

- *S. Malek and N. Medvidovic are with the Department of Computer Science, University of Southern California, 941 W. 37th Pl., Los Angeles, CA 90089-0781. E-mail: {malek, neno}@usc.edu.*
- *M. Mikic-Rakic is with Google Inc., 2644 30th Street, Santa Monica, CA 90405. E-mail: marija@google.com.*

Manuscript received 30 Mar. 2004; revised 18 Feb. 2005; accepted 28 Feb. 2005; published online 20 Apr. 2005.
Recommended for acceptance by J. Kramer.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0053-0304.

existing middleware solutions, including a previously published version of our own work [24].

Another key contribution of Prism-MW is its modular design that employs an extensive separation of concerns. This has resulted in a middleware that is flexible, efficient, scalable, and extensible. The middleware has been assessed for *flexibility* in terms of its support for independent selection, variation, and composition of implementation-level concerns. The middleware has been assessed for *efficiency* in terms of its size, speed, and overhead added to an application. The middleware has been assessed for *scalability* in terms of the numbers of components, connectors, events, execution threads, and hardware devices. Finally, the middleware has been assessed for *extensibility* in terms of support for new development concerns and situations in the Prism setting, including new architectural styles.

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 extensions that have been implemented for a number of architectural styles for distributed systems [10]. At the same time, our experience with and evaluations of Prism-MW have suggested several areas of further study.

The rest of the paper is organized as follows: Section 2 presents our objectives. Section 3 briefly describes an example application used to illustrate the concepts throughout the paper. Section 4 presents the key aspects of Prism-MW's design and implementation. Section 5 then details specific extensions constructed to date, in support of various challenges posed by the Prism setting, while Section 6 discusses Prism-MW's support of different architectural styles. Section 7 evaluates Prism-MW with respect to our objectives. The paper concludes with overviews of related and future work.

2 Objectives

Software development in the Prism setting presents a number of challenges, some of which are unique, while others are "inherited" from PitL. Devices on which Prism applications reside may have limited power, network bandwidth, processor speed, memory, and display size and resolution. Constraints such as these demand highly efficient software systems, in terms of computation, communication, and memory footprint. The trade-offs made to address the scarcity of computing resources directly result in a highly heterogeneous computing environment. The world of Prism is characterized by proprietary operating systems (e.g., PalmOS, Symbian), specialized dialects of existing programming languages (e.g., Sun's Java KVM, Microsoft's Embedded Visual C++, or EVC++), and device-specific data formats (e.g., *prc* for PalmOS, *efs* for Qualcomm's Brew). Modeling, analysis, simulation, and (semi)-automated implementation of software systems are problems on which researchers and practitioners have been working actively for several decades. These problems are only amplified in the highly distributed, heterogeneous, and mobile world of Prism.

Our goal is to investigate the issues and address the challenges highlighted above in the context of Prism-MW. At first blush, a number of the challenges might seem quite familiar to software developers of 30-40 years ago. The resource-constrained nature of the Prism hardware platforms and often-proprietary development infrastructures on those platforms are reminiscent of the development world of the 1960's and 1970's, especially in the arena of embedded systems. At the same time, the sophistication demanded of today's software systems, their comparatively much greater size and complexity, their much wider distribution, their mobility, and their desired interoperability across heterogeneous platforms demand major advances over the solutions that were at the disposal of engineers in the past.

One area from which we believe we can gain a lot of leverage in tackling these challenges is software architecture. Several aspects of architecture-based development (component-based system composition, explicit software connectors, architectural styles, upstream system analysis and simulation, and support for dynamism [20], [21], [27], [28]) appear to make it a good fit for the needs of Prism. Software architecture indeed forms the centerpiece of our approach:

Objective 1. ***Pr***ism-MW should provide native support for designing and implementing *architectural abstractions* (components, connectors, events, and so on). Furthermore, as recent studies (e.g., [1], [18]) have recognized, there is currently a dearth of understanding of which *architectural styles* are suitable for the Prism setting. For this reason, Prism-MW should be configurable to accommodate system development according to the rules of different styles, possibly even in a single application.

Native support for software architecture in a middleware platform is the primary objective of our research. It aims to address the key term in the "Prism" acronym ("**pr**ogramming") in a manner that leverages the best software engineering practices: architecture-based design, component-based implementation, and middleware-based distribution. However, in order to be truly useful in the Prism setting, the resulting solution must also address the other two terms in the "Prism" acronym ("***s***mall" and "***m***any"). We do so via three additional objectives.

Objective 2. Pris***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). Our ongoing work is extending this support to include efficient access to and sharing of hardware resources (e.g., battery, peripheral devices).

Objective 3. 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.

Objective 4. Pris***m***-MW should be *extensible* and configurable in order to accommodate the many and varying development concerns across the heterogeneous Prism setting. These include multiple architectural styles (recall Objective 1), but also awareness [8], mobility [4], [14], dynamic reconfigurability [27], security [29], real-time support [13], and delivery guarantees [5].

Fig. 1. TDS application.

The discussion of Prism-MW and the extent to which it satisfies these objectives is illustrated with the example application introduced in the next section.

3 Example Application

To illustrate the concepts throughout this paper, we use an application for distributed deployment of personnel in situations such as natural disasters, search-and-rescue efforts, and military crises. The specific instance of this application depicted in Fig. 1 was developed in cooperation with a third-party organization and addresses 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 *Commanders* 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. Fig. 1 shows one possible instance of TDS with single *Headquarters*, four *Commanders*, and 36 *Soldiers*.

Through detailed analysis of TDS's requirements and inputs from domain experts, we identified the following set of candidate software components. A *Map* component maintains a model of the system's overall resources: terrain, personnel, tank units, and obstacles. These resources are permanently stored inside a *Repository* component. *StrategyAnalyzerAgent*, *DeploymentAdvisor*, and *SimulationAgent* components, respectively, 1) analyze the deployments of friendly troops with respect to enemy troops and obstacles, 2) suggest deployments of friendly troops based on their availability as well as positions of enemy troops and obstacles, and 3) incrementally simulate the outcome of the battle based on the current situation in the field. *StrategyAnalysisKB* and *SAKBUI* components store the strategy rules and provide the user interface for changing these rules, respectively. *ResourceManager*, *CommanderManager*, *SoldierManager*, and *ResourceMonitor* components enable the allocation and transfer of resources and periodically update the state of resources. *Weather* and *WeatherAnalyzer* components provide weather information and analyze the effects of weather conditions. Finally, a *RenderingAgent* provides the user interface of the application. An architect can select different subsets of these components and compose them into different variants of TDS, such as that shown in Fig. 8 and discussed below.

TDS helps to illustrate a number of Prism concepts. Several aspects of TDS embody the notion of multiplicity inherent in Prism ("many"). As will be discussed below, TDS has been designed using a combination of four architectural styles: client-server, pipe-and-filter, peer-to-peer, and C2. We have implemented it, on top of Prism-MW, in three dialects of two programming languages—Java JVM, Java KVM, and EVC++. The devices on which TDS has been deployed are of several different types (Palm Pilot Vx and VIIx, Compaq iPAQ, HP Jornada, NEC MobilePro, Sun Ultra, PC), running four OSs (PalmOS, WindowsCE, Windows XP, and Solaris). TDS has been deployed onto 105 mobile devices and mobile device emulators running on PCs, where a total of 245 software components interact via 217 software connectors. The dynamic size of the application is approximately 1 MB for the *Headquarters* subsystem, 600 KB for each *Commander*, and 90 KB for each *Soldier* subsystem.

4 Middleware Design

Prism-MW supports architectural abstractions by providing classes for representing each architectural element, enabling a direct mapping between an architecture and its implementation. Furthermore, Prism-MW employs a well-defined extensibility mechanism for addressing emerging development concerns. We discuss the middleware's structure, semantics, and extensibility in this section.

4.1 Prism-MW's Core

Fig. 2 shows the class design view of Prism-MW. The shaded classes constitute the middleware core, which represents a minimal subset of Prism-MW required for implementing and executing an architecture. Only the five dark gray classes of Prism-MW's core are directly relevant to the application developer. Our goal was to keep the core compact, which is reflected in the fact that it contains only 12 classes (four of which are abstract) and four interfaces. Furthermore, we tried to keep the design of the core (and the entire middleware) highly modular by limiting direct dependencies among the classes via abstract classes, interfaces, and inheritance as discussed below.

Brick is an abstract class that represents an architectural building block. It encapsulates common features of its subclasses (*Architecture*, *Component*, *Connector*, and *Port*). *Architecture* records the configuration of its constituent components, connectors, and ports, and provides facilities for their addition, removal, and reconnection, possibly at system runtime. A distributed application is implemented as a set of interacting *Architecture* objects.

Events are used to capture communication in an architecture. An event consists of a name and payload. An

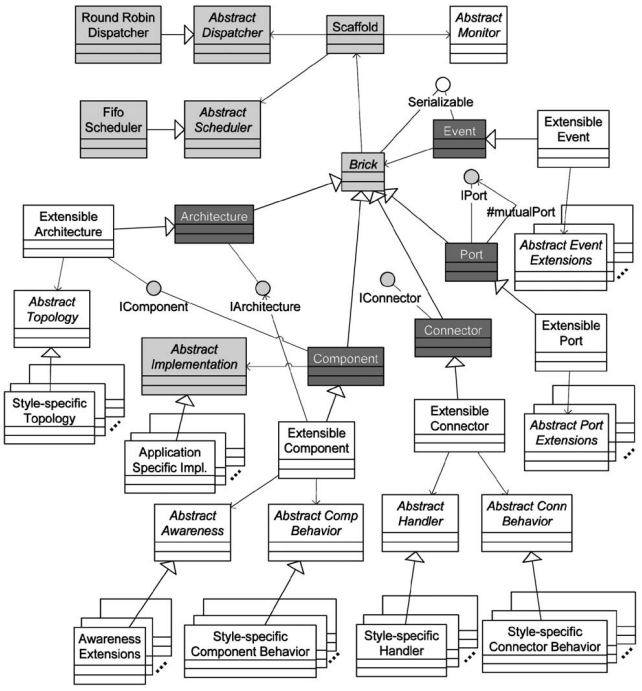


Fig. 2. UML class design view of Prism-MW. Middleware core classes are highlighted.

event's payload includes a set of typed parameters for carrying data and metalevel information (e.g., sender, type, and so on). Two base event types are *request* for a recipient component to perform an operation and *reply* that a sender component has performed an operation.

Ports are the loci of interaction in an architecture. A *link* between two ports is made by *welding* them together; the link acts as a bidirectional communication channel between the ports. A port can be welded to at most one other port. Each *Port* has a type, which is either *request* or *reply*. An event placed on one port is forwarded to the port linked to it in the manner shown in Fig. 3a: Request events are forwarded from request ports to reply ports, while reply events are forwarded in the opposite direction.

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

Components may interact either directly (through ports) or via connectors. *Connectors* are used to control the routing of events among the attached components. Like components, each connector can have an arbitrary number of attached ports. A component attaches to a connector by creating a link between one of its ports and a single connector port. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast). In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime.

Each subclass of the *Brick* class has an associated interface. The *IArchitecture* interface exposes a *weld* method for attaching two ports together. The *IComponent* interface exposes *send* and *handle* methods used for exchanging events. *Component* provides the default implementation of *IComponent*'s *send* method: Generated events are placed asynchronously on all of the component's ports of the appropriate type. As will be detailed in Section 6, we provide other implementations of this interface, including synchronous sending of events. The *IConnector* interface provides a handle method for routing events. The *Connector* class provides the default implementation of *IConnector*'s *handle* method, which forwards all request events to the connector's attached request ports and all reply events to the attached reply ports. As will be detailed in Sections 6, we have provided implementations of different routing policies, including unidirectional broadcast, bidirectional broadcast, and multicast. The *IPort* interface provides the *setMutualPort* method for creating a one-to-one association between two ports.

Finally, Prism-MW's core associates the *Scaffold* class with every *Brick*. *Scaffold* is used to schedule and queue events for delivery (via the *AbstractScheduler* class) and pool execution threads used for event dispatching (via the *AbstractDispatcher* class) in a decoupled manner. Prism-MW's core provides default implementations of *AbstractScheduler* and *AbstractDispatcher*: *FIFOScheduler* and *RoundRobinDispatcher*, respectively. The novel aspect of our design is that this separation of concerns allows us to independently select the most suitable event scheduling, queueing, and dispatching policies for a given application. Furthermore, it allows us to independently assign different scheduling, queueing, and dispatching policies to each architectural element, and possibly even change these policies at runtime. For example, a separate event queue can be assigned to each component; alternatively, a single event queue can be shared by a number of (collocated) components. Additionally, dispatching and scheduling are decoupled from the *Architecture*, allowing one to easily compose many subarchitectures (each with its own scheduling and dispatching policies) in a single application. *Scaffold* also directly aids architectural awareness [4] (also referred to as reflection) by allowing probing of the runtime behavior of a *Brick* via different implementations of the *AbstractMonitor* class, as discussed in Section 5.

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

4.2 Prism-MW's Semantics

A distributed system implemented in Prism-MW consists of a number of *Architecture* objects, each of which serves as a container for a single subsystem and delimits an address

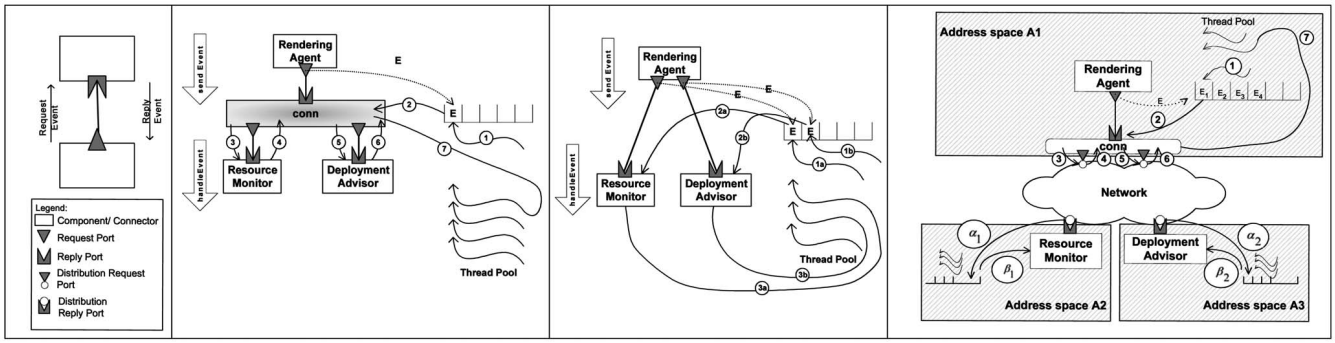


Fig. 3. Event dispatching in Prism-MW. (a) Link between two ports. (b) Steps 1-7 are performed by a single shepherd thread. (c) Steps 1-3 are performed by two shepherd threads, assuming the RenderingAgent is sending event E to both recipient components. (d) Event dispatching for the remote scenario.

space. *Components* within and across the different *Architecture* objects interact by exchanging *Events*. The default implementation of Prism-MW uses a circular array for storing all events in a single address space. This allowed us to optimize event processing by introducing a pool of shepherd threads (implemented in Prism-MW's *RoundRobinDispatcher* class) to handle events sent by any component in a given address space. The size of the thread pool is adjustable. Since the event queue is of a fixed size (determined at system construction-time), we also use a producer-consumer algorithm to keep event production under control and supply shepherd threads with a constant stream of events to process.

Figs. 3b and 3c show event processing for two alternative usage scenarios of Prism-MW. The figures show the base case of event processing within a single address space. This technique proved quite flexible and allowed us to support distributed event processing in a similar manner, as shown in Fig. 3d and further discussed in Section 5.1.

By default, Prism-MW processes events asynchronously. A shepherd thread removes the event from the head of the queue. In the scenario of Fig. 3b, the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread. If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. In the scenario of Fig. 3c, we use direct connections between component ports, which allow separate threads to be used for dispatching an event from the queue to each intended recipient component (Steps 2a, 2b, 3a, and 3b in Fig. 3c). This increases parallelism, but also resource consumption in the architecture.

This solution represents an adaptation of an existing worker thread pool technique [34] that results in several unique benefits:

1. By leveraging explicit architectural topology, an event can be routed to multiple destinations. This minimizes resource consumption since events need not be tagged with their recipients, nor do the recipients need to explicitly subscribe to events.
2. We further optimize resource consumption by using a single event queue for storing both locally and remotely generated events (depicted in Fig. 3d and discussed in Section 5.1).
3. Since Prism-MW processes local and remote events uniformly and all routing is accomplished via the multiple and explicit ports and/or connectors, Prism-MW allows for seamless redeployment and redistribution of existing applications onto different hardware topologies.

4.3 Extensibility Mechanism

One of Prism-MW's key objectives is extensibility (recall Section 2). The design of Prism-MW's core is intended to support this objective by providing extensive separation of concerns via explicit architectural constructs and use of abstract classes and interfaces. To date, we have built several specific extensions to support architectural awareness [8], real-time requirements [13], distributability, security [29], heterogeneity, data compression, delivery guarantees [5], and mobility [4], [14]. Furthermore, we have been able to support directly multiple architectural styles, even within a single application. In this section, we outline the mechanism we have employed for supporting extensibility in Prism-MW, depicted in Fig. 4. We detail how this mechanism is applied to achieve specific extensions in Sections 5 and 6. Our experience with the extensions we have built to date indicates that others can be easily added to the middleware in the manner presented here.

Our support for extensibility is built around our intent to keep Prism-MW's core unchanged. To that end, the core constructs (*Component*, *Connector*, *Port*, *Event*, and *Architecture*) are subclassed via specialized classes (*ExtensibleComponent*, *ExtensibleConnector*, *ExtensiblePort*, *ExtensibleEvent*, and *ExtensibleArchitecture*), each of which has a reference to a number of abstract classes (*AbstractExtensions* in Fig. 4a). Each *AbstractExtension* class can have multiple implementations (*Extension i,j* in Fig. 4a), thus enabling selection of the desired functionality inside each instance of a given *Extensible class*. If a reference to an *AbstractExtension* class is instantiated in a given *Extensible* class instance, that instance will exhibit the behavior realized inside the implementation of that abstract class. Multiple references

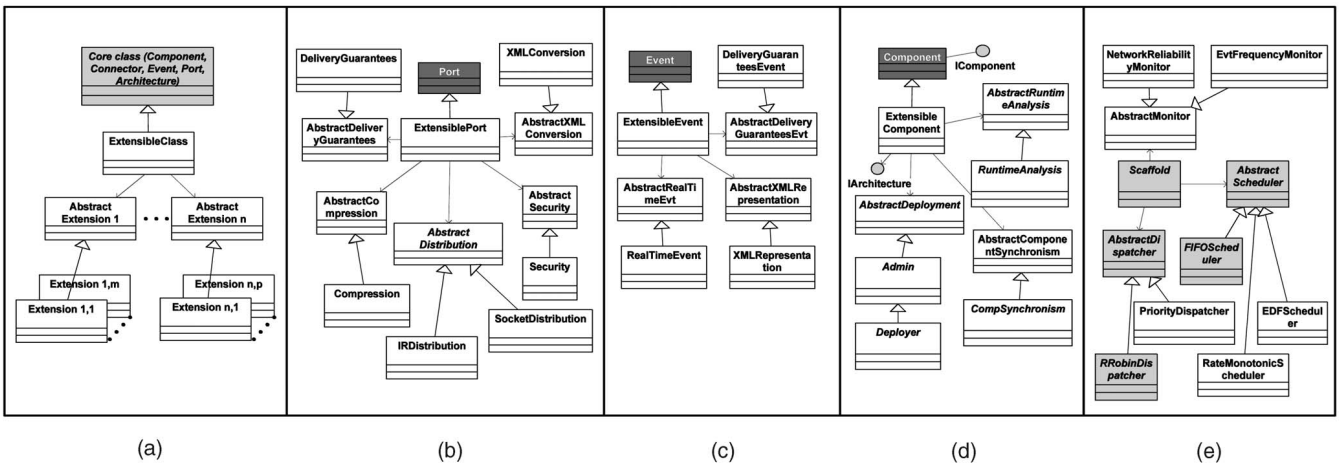


Fig. 4. Prism-MW's (a) extensibility mechanism, and extensions to (b) ports, (c) events, (d) components, and (e) scaffolds.

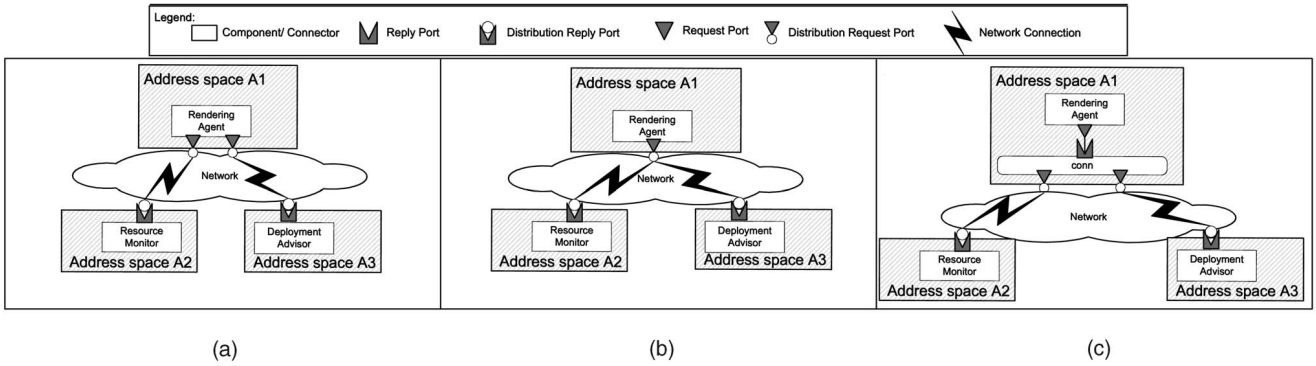


Fig. 5. *DistributionEnabledPort* usage scenarios.

to abstract classes may be instantiated in a single *Extensible* class instance. In that case, the instance will exhibit the combined behavior of the installed abstract class implementations.[1]

5 Support for the Prism Setting

As outlined in Section 2, the Prism setting imposes many stringent requirements on software applications and application developers. We have focused our attention on a number of the resulting challenges and, in this section, report on the facilities we have built into Prism-MW for dealing with several of them.

5.1 Distribution

In order to address different aspects of interaction, the *ExtensiblePort* class has references to a number of abstract classes that support various interaction services. In turn, each abstract class can have multiple implementations. Fig. 4b shows five different port extensions we have implemented thus far. Given the importance of distribution to the Prism domain, in this section, we focus solely on the distribution extensions. In Section 5.2, we will discuss other aspects of interaction.

1. Since Prism-MW's objective is to be arbitrarily extensible in principle, the middleware places no restrictions on such combined behaviors. It is the developers' responsibility to ensure that such behaviors make sense.

The *AbstractDistribution* class has been implemented by two concrete classes, one supporting socket-based and the other infrared port-based interprocess communication (IPC). We refer to an *ExtensiblePort* with an instantiated *AbstractDistribution* reference as a *DistributionEnabledPort*. A *DistributionEnabledPort* can be instantiated in two modes of operation: server or client. A *DistributionEnabledPort* operating in the server mode has a listening thread (e.g., socket server) that is waiting for incoming connection requests on a specified network port. A *DistributionEnabledPort* operating in the client mode does not have a listening thread and is only capable of making connection requests to other *DistributionEnabledPorts*.

Our implementation of *AbstractDistribution* allows a *DistributionEnabledPort* to have an arbitrary number of network connections to other remote *DistributionEnabledPorts* (i.e., one-to-many association between ports). When a *DistributionEnabledPort* receives an event, it broadcasts the event on all its network connections. Note that the one-to-many association between *DistributionEnabledPorts* is a deviation from the one-to-one semantics of a basic port. As will be further discussed in Section 7, this deviation is introduced for efficiency.

Fig. 5 shows three different usage scenarios of Prism-MW's *DistributionEnabledPorts*. The architectures in Figs. 5a and 5b are semantically identical (in both cases, a request from *RenderingAgent* will be sent to both recipient components), although the architecture of Fig. 5b is more efficient

as it uses one less *DistributionEnabledPort*. On the other hand, the architecture in Fig. 5c allows the connector in address space A1 to route an outgoing event either to *ResourceMonitor* in A2 or *DeploymentAdvisor* in A3, or both.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: A sending component or connector places its outgoing event on an attached port. However, instead of depositing the event to the local event queue, in this case, the *DistributionEnabledPort* deposits the event on the network, as shown in Fig. 3d. When the event is propagated across the network, the (server) *DistributionEnabledPort* on the recipient device uses its internal thread to retrieve the incoming event and place it on its local event queue. This supports distribution transparency and allows a component to be migrated between hosts with minimal impact on the system. For example, the impact of moving *ResourceMonitor* from A2 to A1 in Fig. 3d would be limited to replacing two *DistributionEnabledPorts* (one attached to the *conn* connector and the other to *ResourceMonitor*) with local *Ports*.

5.2 Communication Properties

As depicted in Figs. 4b and 4c, we have implemented several port and event extensions, respectively, which enable different aspects of communication beyond distribution. Below, we provide a brief description of each.

5.2.1 Security

The *AbstractSecurity* class (shown in Fig. 4b) is a port extension that has several implementations performing combinations of authentication, authorization, encryption, and event integrity. These services have been implemented using three major cryptographic algorithms: symmetric (secret) key, asymmetric (public) key, and event digest function [29]. We use the RSA asymmetric key algorithm for establishing a connection between each two new users (i.e., their *DistributionEnabledPorts*) and for transmitting the secret (session) key. The same session key and the more efficient DES symmetric key algorithm is used for all subsequent event exchanges. In order to prevent request tampering, a message digest function is used in combination with RSA to generate the message signature. A message digest is a kind of cryptographic checksum over a message, used to verify data integrity.

5.2.2 Delivery Guarantees

AbstractDeliveryGuarantees and *AbstractDeliveryGuaranteesEvt* are port and event extensions, respectively, that support event delivery guarantees. We have implemented support for at most once, at least once, exactly once, and best effort delivery semantics. Each Prism-MW event is tagged with the delivery policy by the component's application logic, with best effort being the default. The delivery policy of an event corresponds to the implementation of *AbstractDeliveryGuaranteesEvt* that is installed on the event. Communicating *DistributionEnabledPorts*, with the appropriate implementation of *AbstractDeliveryGuarantees* installed on them, implement a "handshaking" protocol to ensure proper event delivery across address spaces. If an event does not have any delivery guarantee requirements or if the port does not support the requested delivery guarantee, the event is delivered with the default best effort policy. In order to maximize the efficiency of the delivery guarantee support in the same address space, we make use of programing language (PL) exceptions (i.e., we assume that, if no exception is raised, the event has been delivered).

5.2.3 Real-Time Delivery

The *AbstractRealTimeEvt* class (shown in Fig. 4c) is used to assign a real-time deadline to an event. We have implemented this class to support both aperiodic and periodic real-time events. In support of real-time event delivery, we have provided two additional implementations of the *AbstractScheduler* class (see Fig. 4e). *EDFScheduler* implements scheduling of aperiodic events based on the earliest-deadline-first algorithm, while *RateMonotonicScheduler* implements scheduling of periodic events via rate monotonic scheduling [15]. Coupled with this, we have provided *PriorityDispatcher*, which is a variant implementation of *AbstractDispatcher* that supports threads with varying priorities.

5.2.4 Data Conversion

In order to support communication across PLs, we have provided the *AbstractXMLConversion* and *AbstractXMLRepresentation* extensions for port and event classes, respectively. Prism-MW events with the installed *XMLRepresentation* (our default implementation of *AbstractXMLRepresentation*) are encoded/decoded via ports with *XMLConversion* (our default implementation of *AbstractXMLConversion*).

5.2.5 Data Compression

Finally, we have provided the *AbstractCompression* extension for port with the goal of minimizing the required network bandwidth for event dispatching. To this end, we have implemented the Huffman coding technique [31] inside the *Compression* class.

We have developed the above extensions over the past several years as the need for them has arisen. Adding new extensions to Prism-MW is relatively straightforward. For example, addition of a new extension to *ExtensiblePort* requires adding a reference to the appropriate abstract class and invoking its methods inside *ExtensiblePort*'s *handle* method. Such a change to an *Extensible* class is minimal, averaging three new lines of code for each new extension. The overhead introduced by this solution is that an *ExtensiblePort* instance may have many *null* references, corresponding to the extension classes that have not been instantiated. The values of these references will be checked each time *ExtensiblePort*'s handle method is invoked. An alternative solution, which would trade-off the extensibility for efficiency, is to subclass the *Port* class directly and to have the references only to the desired extensions.

5.3 Awareness

To support various aspects of awareness (i.e., reflection), Prism-MW supports metalevel components. Typically, a metalevel component is implemented as an *ExtensibleComponent*, which contains a reference to the *Architecture* object via the *IArchitecture* interface. The *ExtensibleComponent* class

can also have references to abstract classes that provide specific (metalevel) functionality (see Fig. 4d). The role of components at the metalevel is to observe and/or facilitate different aspects of the execution of application-level components. At any point, the developer may add metalevel components to a (running) application. Metalevel components may be welded to specific application-level connectors to exercise control over a particular portion of the architecture. Alternatively, a metalevel component may remain unwelded and may instead exercise control over the entire architecture via its pointer to the *Architecture* object. The structural and interaction characteristics of metalevel components are identical to those of application-level components, eliminating the need for their separate treatment in the middleware.

To date, we have augmented *ExtensibleComponent* with three extensions. The implementation of the *AbstractRuntimeAnalysis* class is used for analyzing the architectural descriptions and assessing proposed architectural changes during the application's execution. We have implemented several versions of this interface that encapsulate different subsets of our DRADEL [19] environment. In the next section, we discuss in detail the *AbstractDeployment* class, which is used for performing component deployment and mobility. Finally, *AbstractComponentSynchronism* is not relevant to awareness and is discussed in Section 6.

5.4 Deployment and Mobility

A distributed architecture in Prism-MW is represented as a configuration of components (and possibly connectors) deployed onto a set of connected hosts. In such a setting, component migration may be required to minimize the need for remote communication, to increase the local subsystem's autonomy during disconnection, to perform component upgrade, and so forth. Many existing approaches [11] have focused on providing support for component mobility in the Prism setting. Our support for mobility exploits Prism-MW's explicit software connectors, event-based interaction, and awareness. Below, we discuss both stateless and stateful component mobility.

5.4.1 Stateless Mobility/Deployment

Prism-MW components communicate by exchanging application-level events. Prism-MW also allows components to exchange *ExtensibleEvents*, which may contain architectural elements (components and connectors) as opposed to data. Additionally, *ExtensibleEvents* implement the *Serializable* interface (recall Fig. 2), thus allowing their dispatching across address spaces.

In order to migrate the desired set of architectural elements onto a set of target hosts, we assume that a skeleton configuration is preloaded on each host. The skeleton configuration consists of Prism-MW's *Architecture* object that contains an *AdminComponent* with a *DistributionEnabledPort* attached to it. An *AdminComponent* is an *ExtensibleComponent* with the *Admin* implementation of *AbstractDeployment* installed on it (shown in Fig. 4d).

Since the *AdminComponent* on each device contains a pointer to its *Architecture* object, it is able to effect runtime changes to its local subsystem's architecture: instantiation, addition, removal, connection, and disconnection of components and connectors. *AdminComponents* are able to send and receive from any device to which they are connected the *ExtensibleEvents* that contain application components and connectors (referred to as migrant elements below).

The process of stateless migration can be described as follows: The sending *AdminComponent* packages the migrant element into an *ExtensibleEvent*. One parameter in the event is the compiled image of the migrant element itself (e.g., a collection of Java class files); another parameter denotes the intended location of the migrant element in the destination subsystem's configuration. The *AdminComponent* then sends this event to its *DistributionEnabledPort*, which forwards the event to the attached remote *DistributionEnabledPorts*. Each receiving *DistributionEnabledPort* delivers the event to its attached *AdminComponent*, which reconstitutes the migrant elements from the event, and invokes the *IArchitecture*'s add and weld methods to insert the element into the local configuration.

5.4.2 Stateful Mobility

The technique described above provides the ability to transfer code between a set of hosts. As such, the stateless technique is useful for performing initial deployment of a set of components and connectors onto target hosts. In cases when runtime migration of architectural elements is required, the migrant element's state needs to be transferred along with the compiled image of that element. Additionally, the migrant element may need to be disconnected and deleted from the source host. We provide two complementary techniques for stateful mobility: serialization-based and event stimulus-based.

The serialization-based technique relies on the existence of Java-like serialization mechanisms in the underlying PL. Instead of sending a set of compiled images, the local *AdminComponent* possibly disconnects and removes the (active) migrant elements from its local subsystem (using the *IArchitecture*'s *unweld* and *remove* methods), serializes each migrant element, and packages them into a set of *ExtensibleEvents*, which are then forwarded by the *DistributionEnabledPort*. *AdminComponents* on each receiving host reconstitute the architectural elements from these events and attach them to the appropriate locations in their local subsystems.

In cases where the serialization-like mechanism is not available (e.g., Java KVM), we use the event stimulus-based technique: The compiled image of the migrant element(s) is sent across a network using the stateless technique. In addition, each event containing a migrant element is accompanied by a set of application-level events needed to bring the state of the migrant element to a desired point in its execution (see [25] for details of how such events are captured and recorded). Once the migrant architectural element is received at its destination, it is loaded into memory and added to the architecture, but is not attached to the running subsystem. Instead, the migrant element is stimulated by the application-level events sent with it. Any events the migrant element issues in response are not propagated since the element is detached from the rest of the architecture. Only after the migrant architectural element is brought to the desired state is it welded and

enabled to exchange events with the rest of the architecture. While less efficient than the serialization-based migration scheme, this is a simpler technique, it is PL-independent, and it is natively supported in Prism-MW.

5.5 Disconnected Operation

Highly distributed and mobile systems that are commonly found in the Prism domain are challenged by the problem of disconnected operation [41], where the system must continue functioning in the temporary absence of network connectivity. Ensuring the availability of a given system during disconnection may, thus, require the system to be redeployed such that the most critical interactions occur either locally or over reliable network links.

Prism-MW's support for disconnected operation leverages its support for the awareness and mobility discussed above and requires runtime monitoring facilities. In support of monitoring, Prism-MW provides the *AbstractMonitor* class associated through the *Scaffold* with every *Brick* (shown in Fig. 4e). This allows for autonomous, active monitoring of a *Brick*'s runtime behavior. We have provided two implementations of the *AbstractMonitor* class: *EvtFrequencyMonitor* records the frequencies of different events the associated *Brick* sends, while *NetworkReliabilityMonitor* records the reliability of connectivity between its associated *DistributionEnabledPort* and other, remote *DistributionEnabledPorts* using a common "pinging" technique.

To maximize the availability of a distributed system, we leverage two implementations of the *AbstractDeployment* class:

- *AdminComponent*, discussed in Section 5.4, is capable of sending, receiving, and installing software components via its reference to *Architecture*. An *AdminComponent* is also capable of accessing the monitoring data of its local components and connectors (recorded in the associated implementation of the *AbstractMonitor* class) and forwarding that data to interested remote hosts.
- *DeployerComponent* is an *ExtensibleComponent* with an attached *DistributionEnabledPort* and the *Deployer* implementation of *AbstractDeployment* installed on it. The *DeployerComponent* has all the capabilities of an *AdminComponent* (recall Fig. 4d) with the additional ability to calculate a new deployment architecture that improves the system's availability based on the monitored data [23]. Once an improved deployment architecture is calculated, the *DeployerComponent* orchestrates the system's redeployment by sending redeployment instructions to the *AdminComponents* on each host.

6 Support for Architectural Styles

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems' requirements [10], [28]. Therefore, a middleware platform used to implement such architectures would need to support multiple styles. Prism-MW's design can be leveraged to support a number of distributed systems styles [10], which are likely to be useful in the Prism setting [18]. In this section, we describe how Prism-MW can be configured to support different architectural styles using the mechanism introduced in Section 4.3 and leveraged in the extensions described in Section 5.

In order to effectively support architectural styles, Prism-MW should be configured to provide the following:

1. the ability to distinguish among different architectural elements of a given style (e.g, distinguishing `Clients` from `Servers` in the client-server style);
2. the ability to specify the architectural elements' stylistic behaviors (e.g., `Clients` block after sending a request while `C2Components` send requests asynchronously);
3. the ability to specify the rules and constraints that govern the architectural elements' valid configurations (e.g., disallowing `Clients` from connecting to each other in the client-server style, or allowing a `Filter` to connect only to a `Pipe` in the pipe-and-filter style); and
4. the ability to use multiple architectural styles within a single application.

We have leveraged Prism-MW's extensibility to support the above requirements. The following extensibility properties of Prism-MW have been used to satisfy the requirements:

- Brick has an attribute that identifies its style-specific type. The value of this variable corresponds to a given architectural style element, e.g., `Client`, `Server`, `Pipe`, `Filter`, etc. The default value of this variable is *Null*, corresponding to the "null" style supported by Prism-MW's core. The association of *Brick* with its style-specific type satisfies our first requirement by enabling identification of different architectural elements.
- *ExtensibleConnector* has an associated implementation of the *AbstractHandler* class to support style-specific event routing policies (see Fig. 6a). For example, `Pipe` forwards data unidirectionally, while a `C2Connector` uses bidirectional event broadcast. This partially satisfies the second requirement by allowing tailoring of a connector's style-specific behavior.
- *ExtensibleComponent* has an associated implementation of the *AbstractComponentSynchronism* class to provide synchronous component interaction (see Fig. 6b). The default, asynchronous interaction is provided by Prism-MW's core. This partially satisfies the second requirement by allowing one to tailor a component's style-specific behavior (e.g., a `Client` blocks after it sends a request to a `Server` and unblocks when it receives a response).
- *ExtensiblePort* has an associated implementation of the *AbstractDistribution* class to support interprocess communication (see Fig. 6c). This partially satisfies the second requirement by supporting architectural styles that require distribution (e.g., a `Server` may serve many distributed `Clients`).

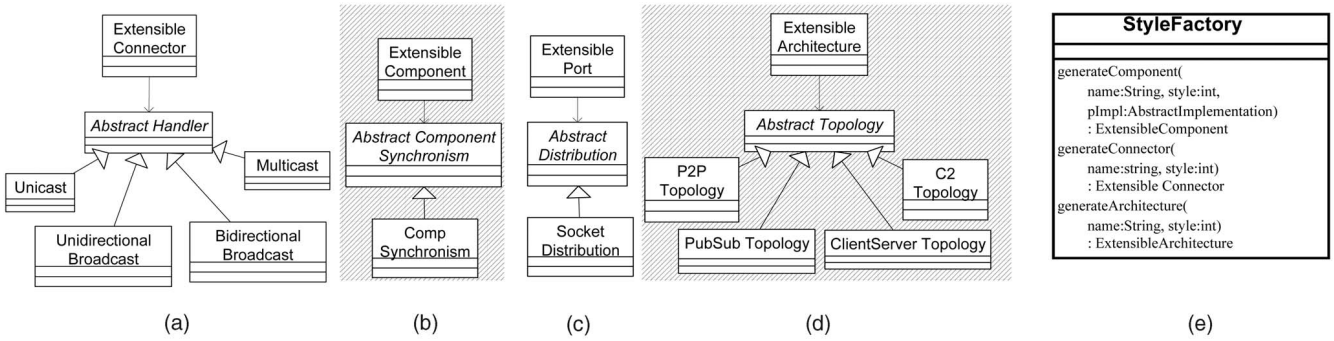


Fig. 6. Prism-MW's support for architectural styles.

- *ExtensibleArchitecture* has an associated implementation of the *AbstractTopology* class to ensure the topological constraints of a given style (see Fig. 6d). For example, in the client-server style, `Clients` can connect to `Servers`, but two `Clients` cannot be connected to one another. Each time an *ExtensibleArchitecture*'s *weld* (or *unweld*) method is invoked, the appropriate implementation of the *AbstractTopology* ensures that the topological rules of a given style are preserved. As a result, it either performs the *weld* (or *unweld*) operation or raises an exception. This satisfies the third requirement stated at the beginning of this section by allowing for the specification and modification of valid configurations of architectural elements. Note that, while invoking *IArchitecture*'s *add* method results in the addition of a component/connector to the *Architecture* (or *ExtensibleArchitecture*) object, it does not affect the system's architectural style until it is *welded*. Similarly, the component/connector cannot be removed before *AbstractTopology*'s *unweld* method is called, which ensures that the removal will not undermine the system's architectural style.
- *ExtensibleArchitecture* implements the *IComponent* interface, thereby allowing hierarchical composition of components (see Fig. 2). Each hierarchical component is internally composed of subarchitectures that can adhere to different architectural styles. This satisfies the fourth requirement by allowing combinations of different styles in a single system.

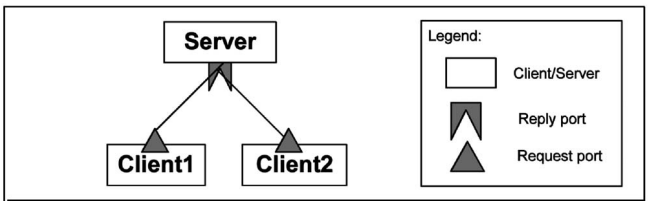
6.1 Support for Individual Styles

To produce a style-specific architectural element, the developer instantiates the corresponding *Extensible* class (recall Fig. 6) and sets the desired stylistic behavior by installing the appropriate extensions on it. To simplify this task, we have provided a *StyleFactory* utility class (shown partially in Fig. 6e) that can automatically generate style-specific architectural elements.

For illustration, we will discuss how we have developed support for the client-server style. In this style, a client is a triggering process; a server is a reactive process. Clients make requests that trigger reactions from servers. Thus, a client initiates activity at times of its choosing and, then, blocks until its request has been serviced. On the other hand, a server waits for requests to be made and then reacts to them.

Both `Client` and `Server` in Prism-MW are represented using an *ExtensibleComponent*. However, `Client` uses an implementation of *AbstractComponentSynchronism* which overrides the default nonblocking behavior of a component. `Clients` make synchronous requests by blocking until the corresponding acknowledgement reply comes back. An acknowledgement reply indicates the completion of the requested operation on the `Server`. A `Client` can have one or more request ports through which it sends request events to the `Servers`, but cannot have any reply ports. A `Server` component can have one or more reply ports through which it sends reply events back to the requesting `Clients`. Prism-MW supports client-server applications that reside in one or more address spaces.

Fig. 7 shows a simple client-server style architecture and the corresponding code in Prism-MW. A client-server architecture is composed of an *ExtensibleArchitecture* with the *ClientServerTopology* implementation of the *AbstractTopology*. *ClientServerTopology* enables welding of `Clients`




```
class ClientServer {
  static public void main(String argv[]) {
    ExtensibleArchitecture arch =
    StyleFactory.generateArchitecture("arch", CLIENT_SERVER_ARCH);

    // create components here
    AbstractImplementation client1Impl = new Client1();
    ExtensibleComponent client1 =
    StyleFactory.generateComponent("client1", CLIENT, client1Impl);
    AbstractImplementation client2Impl = new Client2();
    ExtensibleComponent client2 =
    StyleFactory.generateComponent("client2", CLIENT, client2Impl);
    AbstractImplementation serverImpl = new Server();
    ExtensibleComponent server =
    StyleFactory.generateComponent("server", SERVER, serverImpl);

    // add components to the architecture
    arch.add(client1);
    arch.add(client2);
    arch.add(server);

    // establish the interconnections
    arch.weld(client1,server);
    arch.weld(client2,server);

    arch.start(); // start the architecture

  }
}
```

Fig. 7. Client-Server style example.

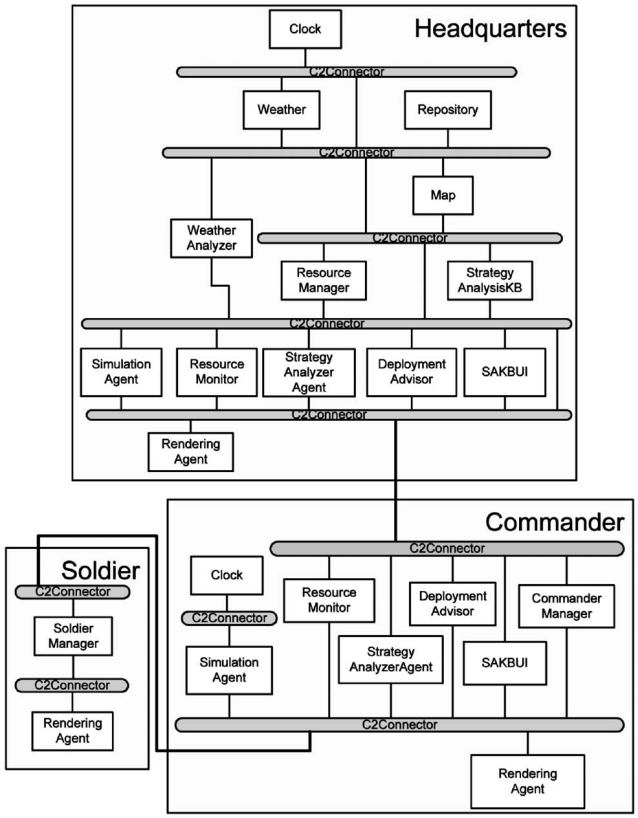


Fig. 8. Architecture of TDS in the C2 style with single *Headquarters*, *Commander*, and *Soldier* subsystems. Prism-MW Ports have been elided for clarity.

and `Servers` while enforcing the topological rules (e.g., disallowing the welding of two `Clients`).

We have implemented a number of additional styles (see Fig. 6d) in a similar manner. Each style required, on average, the addition of 80 new SLOC to Prism-MW. Changes to Prism-MW were localized to new implementations of *AbstractHandler* and *AbstractTopology* classes. On average, the described extensions for each style required less than one person-hour of effort, including testing. As the number of supported styles in Prism-MW grows, we expect that implementing a new style would require even less effort since existing style implementations (e.g., different connector routing policies) may be reused.

6.2 Multiple Styles in a Single Application

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems' requirements. Prism-MW supports the use of multiple architectural styles in a single application by leveraging hierarchical composition. *ExtensibleArchitecture* implements the *IComponent* interface (recall Fig. 2) and, therefore, allows its instances to be used as hierarchical components. The application architecture that contains multiple styles is then composed as a configuration of several hierarchical components (with their own internal architectures), each of which may adhere to a different architectural style. We discuss an example of this using the TDS application introduced in Section 3.

TDS was initially designed in the C2 style and implemented using an earlier version of Prism-MW [24]. Fig. 8 shows the initial architecture of TDS. It consists of three subsystems: *Headquarters*, *Commander*, and *Soldier*, each of which resides on a separate device and has an internal architecture. We modified the overall TDS architecture such that the *Soldier* and *Commander* subsystems, as well as *Commander* and *Headquarters* subsystems, engage in client-server relationships. Different *Commander* subsystems have also been rearchitected to communicate as peers. The details of these modifications have been elided for brevity, but can be found in [18], [24]. Another issue we faced was that, in the original C2 design, the *Clock* component was overwhelming the system by broadcasting tick messages to all the components in the system, while only the application logic of three components (*Weather*, *ResourceMonitor*, and *SimulationAgent*) needed the delivery of the tick messages. Thus, we moved the *Clock* and its three dependent components into a separate pipe-and-filter style subarchitecture. This modification resulted in over 60 percent improvement of event round-trip time for some events (e.g., events originating from the *RenderingAgent* component that are being processed by the *Map* component).

7 Evaluation

Prism-MW's support for two of its objectives, system implementation via architectural abstractions and extensibility, have been discussed in depth in the preceding sections. In this section we summarize our evaluation of Prism-MW's remaining two objectives, efficiency and scalability (recall Section 2). Our goals have been to 1) provide empirical results of the performance trade-offs that are associated with our design decisions and 2) demonstrate the middleware's efficiency and scalability in large, possibly distributed systems with different structures.

In support of the first goal, we have evaluated two types of performance trade-offs that were discussed earlier in the paper:

- The trade-off between the two alternative configurations of a local architecture (recall Fig. 3). We used architectures in which a single component communicates with a varying number of identical components, either through a single connector or via ports that directly connect the components.
- The trade-off between the three alternative configurations of a distributed architecture (recall Fig. 5). We considered a special case, where the connectors broadcast events, which results in five semantically identical configurations of a distributed architecture, as detailed below.

In support of the second goal, we have measured the overhead in application size caused by Prism-MW. We have also evaluated the execution of large architectures with different topologies and processing loads:

- The sensitivity analysis of the middleware's performance to the size of the architecture. We consider an architecture configured in a manner similar to the one depicted in Fig. 3a and execute it on different

platforms for varying numbers of events and "bottom" components.
- The middleware's scalability to large architectures with different topologies.

Since we are interested in measuring the overhead induced on an application by Prism-MW, the software components used throughout the example scenarios contain minimal application logic (e.g., counting the number of events sent/received, forwarding events). Furthermore, all the events exchanged between local and remote components are simple events with no payload. We selected the size of the thread pool and event queue based on the expected load and the size of each architecture. Note that Prism-MW allows for the specification of arbitrarily large thread pools and event queues. In the case of most benchmarks, we kept a constant size thread pool and event queue to simplify the assessment of the middleware's performance under varying loads.

The environment set-up consisted of 1) mid-range PCs with Intel Pentium IV 1.5 GHz processors and 256 MB of RAM running JVM 1.4.2 on Microsoft Windows XP, 2) PDAs of type Compaq iPAQ H3870 with 200 MHz processors and 64 MB of RAM running Jeode JVM on WindowsCE 2002, and 3) a dedicated network leveraging a dual-band wireless 2.4 GHz router.

7.1 Middleware Size

Memory usage of Prism-MW's core (mw_mem), recorded at the time of architecture initialization, is 2.3 KB. The overhead of a "base" Prism-MW component ($comp\_mem$), without any application-specific methods or state, is 0.12 KB, while the overhead of a "base" connector ($conn\_mem$) is 0.09 KB. The memory overhead of a "base" port ($port\_mem$) is 0.04 KB, while the overhead of an *ExtensiblePort* is 0.2 KB. The memory overhead of each connection object is 8 KB, which leverages Java's implementation of socket-based TCP/IP communication protocol. The memory overhead of a *DistributionEnabledPort* that contains a single connection instance is 8.5 KB. The memory overhead of creating and sending a single event (evt_mem) can be estimated using the following formula, obtained empirically:

$$evt\_mem(in\ KB) = 0.04 + 0.01\ ^*\ num\_of\_parameters.$$

The formula assumes that the parameters do not contain complex objects, but may contain simple objects (e.g., Java Integer or String).

As an illustration, the memory overhead induced by using Prism-MW in the largest instantiation of the TDS architecture (recall Fig. 8), which consisted of a single *Headquarters* subsystem, four *Commander* subsystems, and 100 *Soldier* subsystems can be closely approximated as follows:

$$\begin{aligned} &num\_arch\ ^*\ (mw\_mem + (q\_size\ ^*\ evt\_mem)) \\ &+ num\_comps\ ^*\ comp\_mem + num\_conns\ ^*\ conn\_mem \\ &+ num\_ports\ ^*\ port\_mem + num\_dist_{p}orts\ ^*\ dist\_port\_mem \\ &= 105\ ^*\ (2.3 + 25\ ^*\ (0.04 + (0.01\ ^*\ 1))) + (245\ ^*\ 0.12) \\ &+ (217\ ^*\ 0.09) + (875\ ^*\ 0.04) + (109\ ^*\ 0.5) = 511.5KB. \end{aligned}$$

The above formula uses the average size of the event queue for each *Architecture* object (25) and average number of parameters for TDS events (one). The formula also assumes that each event queue is full (which we have never observed during actual execution of TDS). Recall from Section 3 that the approximate dynamic size of the *Headquarters* subsystem is 1 MB, of each *Commander* subsystem 600 KB, and of each *Soldier* subsystem 90 KB, resulting in the total application size of 12.5 MB. Therefore, Prism-MW induced at most a 4 percent overhead on the application's dynamic memory consumption.

Our measurements of the memory overhead for the awareness, deployment, mobility, and disconnected operation support (recall Section 5) showed that, on average, the Java implementation of the Prism-MW skeleton configuration (*AdminComponent*, *DistributionEnabledPort*, and Prism-MW's core) occupies around 14 KB. The *AdminComponent* itself occupies 4 KB of memory.

7.2 Middleware Performance in a Local Setting

The performance of Prism-MW's core (i.e., local architectures with no extensions) is comparable to solutions using a plain programming language (PL). Each Prism-MW event exchange causes five PL-level method invocations (typically highly optimized in a PL) and a comparatively more expensive context switch if the architecture is instantiated with more than one shepherd thread.[2] 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.

To empirically evaluate Prism-MW's core, we use an architecture where one component is communicating with a varying number (n) of identical recipient components via a connector (Fig. 3b shows such an architecture with two recipient components). Thus, all the components in this architecture are part of the same *Architecture* object and reside in a single address space. For this architecture, we use a pool of 10 shepherd threads and a queue of 1,000 events (q_size).

Table 1 shows representative benchmark results. A maximum of 100,000 simple (parameter-less) events were sent asynchronously by the single sender component to a maximum of 100 recipient components (resulting in between 100 to 10,000,000 invocations of component *handle* methods) for the application running on a PC. The 10 million events are processed in under 3 seconds on the PC. The bottom portion of Table 1 shows the results obtained on a PDA, a comparatively much less capacious and performant platform: a maximum of 10,000 events are sent to a maximum of 50 components (resulting in up to 500,000 invocations of component *handle* methods).

In addition to the above "flat" architecture, another series of benchmarks we ran involved a "chain" of n components communicating either directly through ports or

2. The five method invocations involve traversing the ports, placing the event in the queue, and dispatching it to the recipient component.

TABLE 1
Benchmarking Prism-MW on a PC and a PDA

	Events / Comps	100000	10000	1000	100	1
PC	100	2674ms	300ms	50ms	20ms	20ms
	50	1843ms	211ms	40ms	20ms	10ms
	10	1222ms	150ms	30ms	11ms	10ms
	1	1081ms	131ms	30ms	10ms	1ms
PDA	50	n/a	19649ms	1871ms	385ms	289ms
	10	n/a	17654ms	1403ms	325ms	217ms
	1	n/a	16784ms	1320ms	256ms	62ms

via $n-1$ intervening connectors. For example, the total round-trip time for a single event in the case where the architecture involved 100,001 components and 100,000 connectors was 1.1 milliseconds on a PC. In addition to demonstrating Prism-MW core's efficiency, these benchmarks also served to highlight its scalability.

To evaluate the performance trade-off between two alternative usage scenarios of Prism-MW (recall Fig. 3), we employed two variations of the "flat" architecture. In the first variation, discussed above, the communication takes place through a single connector, while the second variation employs direct links between component ports. Each one of the n components was implemented with a fixed event handling delay of 50 msec to simulate application-specific processing ($comp\_proc\_time$) and to utilize the benefits of parallel processing.

The results of the benchmark are shown in Table 2. One parameter-less event was sent asynchronously by the single sender component to all the recipient components, resulting in n events being handled. The results demonstrate that a higher degree of parallelism and, therefore, better performance, can be achieved by using direct connections among components. On the other hand, the use of a connector resulted in lower memory consumption since each outgoing event is not replicated n times. Finally, note that the total processing time in the case of direct communication (illustrated in the bottom half of Table 2) can be approximated using the following formula, where *numComps* represents the number of components and *numThreads* represents the number of shepherd threads in an architecture:

$$total\_proc\_time \approx comp\_proc\_time, \quad \text{if } numComps < numThreads$$
$$total\_proc\_time \approx comp\_proc\_time^{*} numComps/numThreads, \quad \text{if } numComps \geq numThreads.$$

7.3 Middleware Performance in a Distributed Setting

While Prism-MW's *DistributionEnabledPorts* are in principle independent of the employed communication protocols (recall Section 5.1), their performance is directly impacted by the underlying implementations of those protocols. The results presented here are based on *DistributionEnabledPorts* that leverage Java's implementation of TCP/IP sockets. In a large number of benchmarks involving architectures of varying sizes, topologies, and communication profiles, we

TABLE 2
Components Communicating through a Connector versus Directly via Ports

	Comps / Threads	10000	1000	100	10	1
Conn	2	500730ms	50072ms	5017ms	511ms	60ms
	10	500760ms	50082ms	5017ms	501ms	60ms
	50	500785ms	50082ms	5057ms	511ms	60ms
	100	500760ms	50079ms	5027ms	551ms	60ms
Ports	2	251302ms	25056ms	2514ms	261ms	60ms
	10	50102ms	5028ms	500ms	60ms	60ms
	50	10075ms	1032ms	111ms	50ms	60ms
	100	5077ms	541ms	70ms	50ms	60ms

TABLE 3
Distributed Architecture Scenarios

Events / Scenario	1	30	100
1	351kb, 210ms	351kb, 2377ms	351kb, 6679ms
2	310kb, 210ms	310kb, 2313ms	310kb, 6786ms
3	132kb, 110ms	132kb, 1155ms	132kb, 2760ms
4	123kb, 110ms	123kb, 1185ms	123kb, 2533ms
5	86kb, 86ms	86kb, 731ms	86kb, 1735ms

compared the performance of a *DistributionEnabledPort* with a "pure" Java implementation of TCP/IP. Our results indicate that a Prism-MW *DistributionEnabledPort* adds no more than 2 percent in performance overhead to Java's implementation of TCP/IP.

In Section 5.1, we identified three different ways of instantiating a distributed architecture with identical event routing semantics. In fact, when the routing policy is event broadcast (i.e., no filtering is performed by connectors), five semantically equivalent ways of instantiating a distributed architecture are possible. To study their performance, we created five example scenarios, comprising three distributed *Architecture* objects with five components each, that were configured as follows:

- **Scenario 1**. Each component communicates directly to every other component via a separate *DistributionEnabledPort* (see Fig. 5a).
- **Scenario 2**. Each component on the requesting device uses a single *DistributionEnabledPort* to communicate directly to every other component (see Fig. 5b, where the requesting device corresponds to address space A1).
- **Scenario 3**. Each component on the requesting device uses a local bidirectional broadcasting connector to communicate with remote components. The connector has a separate *DistributionEnabledPort* for each remote component (see Fig. 5c).
- **Scenario 4**. This is similar to the architecture of Scenario 3, with the exception that the connector has only a single *DistributionEnabledPort* to communicate with all of the remote components.
- **Scenario 5**. Local bidirectional broadcasting connectors with single *DistributionEnabledPorts* are used to mediate the communication in all three architectures.

Table 3 shows the performance measurements under each of the scenarios described above, with different event loads sent by the requesting architecture. The measurements reflect the time elapsed before the requesting architecture receives all the reply events from the remaining architectures. Given that the ports and connectors used in this example broadcast the events, each event sent by one of the five requesting components (i.e., between 5 and 500 events sent in the scenarios depicted in Table 3) results in a total of 10 replies returned (i.e., between 50 and 5,000 events returned).

We make the following two observations from the results of Table 3: First, architectures with lower numbers of *DistributionEnabledPorts* have lower memory footprints and faster running times. This is expected since each *DistributionEnabledPort* adds overhead both in terms of memory and execution. Therefore, in response to this issue, Prism-MW allows multiple connections to be associated with a single *DistributionEnabledPort*. More significantly, architectures with lower numbers of network connections have much lower memory footprints and faster running times. This is expected since each network connection has its own internal thread that reads/writes events from/to the network link. To minimize the number of network connections, one may leverage Prism-MW's connectors. For example, in Scenario 5, the usage of connectors in all the three architectures resulted in the most efficient configuration, in which a total of only three *DistributionEnabledPorts* and four connections are instantiated.

8 Related Work

Our work on Prism-MW has been primarily influenced by two research areas: architectural styles and middleware. *Architectural* styles were discussed in Sections 1 and 6. Below, we discuss three related approaches in the architectural middleware arena. Additionally, we describe several representative commercial and research middleware technologies and present a comparison of these technologies with Prism-MW.

ArchJava [2] 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 [36] is an architectural style and supporting middleware for ubiquitous computing applications with a special focus on user mobility, context awareness, and context switching. Aura is thus only applicable to certain classes of applications in the 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 implementation support, which has to date only been tested on traditional, desktop platforms.

AGILE [3] is an on-going project whose aim is to develop an architectural approach in which mobility aspects will be modeled explicitly and mapped onto the physical distribution and communication topology. AGILE comprises several different facets, such as its primitives for explicitly addressing mobility within architectural models based on CommUnity [9] and algebraic models of the evolution processes that result from system reconfiguration caused by component mobility. Of particular interest to this paper is AGILE's Open KLAIM [3], an experimental kernel programming language based on tuple-spaces, for modeling and programming distributed concurrent applications with code mobility. Unlike Prism-MW, in which network connectivity is implicit in its use of *DistributionEnabledPorts*, Open KLAIM provides constructs for explicitly modeling connectivity between network nodes and for handling changes in the network topology. At the same time, Open KLAIM lacks support for explicit architectural constructs (components and connectors) and focuses instead on processes as units of mobility. The two projects thus appear to be complementary from this perspective.

While we have found only the above three related approaches in the software architecture literature, we have performed a comparison of Prism-MW with several representative middleware solutions with respect to the objectives identified in Section 2. The cross-section of the selected middleware technologies covers commercial (Orbix/E [12] and .NET [22]), research (ACE [33], XMIDDLE [16], RCSM [42], Lime [14], and MobiPADS [6]), as well as open-source solutions (JINI [37] and JXTA [30]). Bellow, we provide a comparison of these technologies with Prism-MW.

Architectural Abstractions. TAO, Orbix/E, .NET, and MobiPADS provide partial support for architectural abstractions in the form of explicit components. However, none of these middleware solutions support multiple, explicit, and/or tailorable software connectors. Furthermore, none of them support explicit architectural styles, thus clearly distinguishing Prism-MW from them. The styles in all of the surveyed technologies are implicit and mostly fall within the distributed objects category.

Efficiency. We were unable to locate information on the performance of several of the studied middleware platforms. In this section, we report on those whose speed and size have been discussed in literature. The studied middleware platforms employ different optimization strategies and, as a result, achieve different levels of efficiency. Efficiency in Orbix/E is achieved by providing the ability to choose a subset of features for a given application. However, even the minimal Orbix/E configuration still requires 95 KB of memory. TAO has recently undergone a major redesign to improve its efficiency; however, the minimal configuration of TAO still requires over 500 KB of memory, while its commercialized version for real-time CORBA [26] requires over 1.7 MB of memory. JXTA introduces relatively high payload overheads on each event (1KB). Its' protocols are XML-based, resulting in both message composition and processing overhead. .NET's efficiency is mainly hampered due to its reliance on the underlying Windows platform. For example, compared to Unix and Linux, spawning a new process on Windows is a relatively time-consuming operation, which decreases .NET's performance. Our measurements indicate that Prism-MW introduces lower memory and performance overheads than all of the surveyed middleware solutions.

Scalability. TAO and Orbix/E support application scalability, but do so at the expense of the middleware size. This is primarily due to the fact that both middleware solutions were initially targeted for capacious desktop platforms and, therefore, their designs were not tailored to mobile and resource constrained devices. Most of the surveyed middleware technologies implicitly support the distributed objects style and their scalability is ultimately hampered by their reliance on a single connector (ORB). On the other hand, peer-to-peer solutions such as JINI address scalability through federations of groups or communities. However, JINI's lookup service has issues with scalability due to its lack of hierarchical organization of lookup servers. As discussed in Section 7, Prism-MW scales well in the numbers of components, connectors, events, and hosts.

Extensibility. Different middleware platforms support different aspects of extensibility, including mobility, reconfigurability, awareness, security, and delivery guarantees. For example, Jini supports mobility and reconfigurability via Java object serialization, .NET and XMIDDLE support mobility and reconfigurability using XML to describe units of mobility, and dynamic class loading and object serialization to perform code migration. RCSM focuses on supporting device mobility and ensuring that components on each mobile device can communicate, but does not support code mobility or software reconfigurability. MobiPADS supports mobility and reconfigurability in the form of facilities known as mobilets, which are configured as chained service objects that provide augmented services to the underlying mobile applications. LIME supports mobility in the form of agents. JINI, XMIDDLE, LIME, and MobiPADS support location awareness. XMIDDLE, JINI, and MobiPADS extend the notion of awareness beyond location to other system properties such as resource availability, battery power, etc. Several technologies (OrbixE, TAO, JXTA, .NET, RCM) support security to a significant extent. Security is highly important in the context of Prism systems, in which possibly untrusted components and hosts may enter the system at any time. For example, .NET supports security via encryption and digital signatures and provides security enforcement at the level of application users and mobile code. TAO provides extensive support for event delivery guarantees, including at most once, at least once, and exactly once delivery semantics, while Orbix/E provides event storage and playback in order to support reliable

event delivery. Other surveyed techniques do not support event delivery guarantees.

9 Conclusions and Future Work

This paper has presented the design, implementation, and evaluation of Prism-MW, a middleware targeted at applications in highly distributed, resource constrained, heterogeneous, and mobile settings. The described design and implementation are a result of close to ten years of research into effective techniques for implementing software architectures [17], [24], [27], [40]. The key properties of Prism-MW are its native and flexible support for architectural abstractions (including architectural styles), efficiency, scalability, and extensibility. These properties were enabled by Prism-MW's extensive separation of concerns that spans several dimensions:

- By adopting an explicit architectural perspective, Prism-MW has inherited the separation of computation (handled by components) from interaction (handled by ports and connectors) intrinsic to software architectures.
- By providing a simple mechanism for supporting multiple architectural styles (possibly in a single application), Prism-MW allows system developers to separate cleanly a system's design from its implementation; Prism-MW's style extensions automatically ensure all relevant architectural relationships and properties.
- Prism-MW's extensive use of abstract classes and interfaces, as well as minimized dependencies among its classes, allow 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).
- Finally, Prism-MW completely separates an application's conceptual architecture from its realization. For example, each component in an architecture may be implemented in multiple PLs; those implementations are fully interchangeable if *ExtensiblePorts* with the appropriate implementations of the *AbstractXMLConversion* class are used.

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.

Our experience with Prism-MW has been very positive thus far. We have used it in the context of graduate-level classes on software architectures and embedded systems at the University of Southern California and in collaborations with three external software development organizations. It has also been successfully used by a team of researchers in the field of mobile communication and ad hoc networks [38]. At the same time, we recognize that a number of pertinent issues remain unexplored. Our future work will span issues such as adding configuration management support to Prism-MW and automatically generating an optimized version of the middleware given a desired set of features (i.e., eliminating the need to store and check abstract class references even when they are not used in a given Prism-MW class implementation). Another alternative we are considering to address this problem is to parameterize Prism-MW's variation points instead of using abstract classes and interfaces. We are not aware of any comparable attempts at parameterizing middleware to this extent and consider this to be a very interesting research challenge.

Acknowledgments

The authors would like to thank the reviewers for their detailed and thoughtful reviews. They wish to acknowledge R. Banwait, N. Beckman, M. Bhachech, V. Jakobac, V. Kudchadkar, C. Mattmann, A. Rampurwala, E. Sanchez, V. Viswanathan, and students from USC's CSCI 578 and 589 courses who contributed to the development of Prism-MW and various applications on top of it. They thank G. Sukhatme for the many discussions on the subject of handheld, mobile, and embedded computing. Finally, they especially thank B. Boehm and R. Taylor for their generous support in obtaining the initial equipment used in the described research. The work described in this paper was supported by an equipment grant from Intel. This material is based upon work supported by the US National Science Foundation under Grant Numbers CCR-9985441 and ITR-0312780, Jet Propulsion Laboratory, US Army TACOM, and DARPA under agreement number F30602-00-2-0615.

References

[1] G. Abowd, "Programming Environments. . . Literally: Ubicomp's Grand Challenge for Software Engineering," *Proc. SIGSOFT Symp. Foundations of Software Eng.,* Nov. 2002.

[2] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," *Proc. Int'l Conf. Software Eng.,* pp. 187-197, May 2002.

[3] L. Andrade, P. Baldan, H. Baumeister, R. Bruni, A. Corradini, R. DeNicola, J.L. Fiadeiro, F. Gadducci, S. Gnesi, P. Hoffman, N. Koch, P. Kosiuczenko, A. Lapadula, D. Latella, A. Lopes, M. Loreti, M. Massink, F. Mazzanti, U. Montanari, C. Oliveira, R. Pugliese, A. Tarlecki, M. Wermelinger, M. Wirsing, and A. Zawlocki, "AGILE: Software Architecture for Mobility," *Proc. 16th Int'l Workshop Algebraic Development Techniques,* pp. 1-33, 2003.

[4] C. Mascolo, L. Capra, and W. Emmerich, "Middleware for Mobile Computing (A Survey)," *Advanced Lectures on Networking—Networking 2002 Tutorials,* pp. 20-58, May 2002.

[5] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Trans. Computer Systems,* vol. 19, no. 3, pp. 332-383, Aug. 2001.

[6] A. Chan and S. Chuang, "MobiPADS: A Reflective Middleware for Context-Aware Mobile Computing," *IEEE Trans. Software Eng.,* vol. 29, no. 12, pp. 1072-1085, Dec. 2003.

[7] F. DeRemer and H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Trans. Software Eng.,* vol. 2, no. 2, pp 321-327, June 1976.

[8] W. Emmerich, "Software Engineering and Middleware: A Roadmap," *Proc. Conf. Future of Software Eng.,* pp. 117-129, 2000.

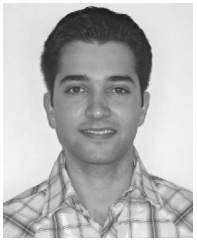
[9] J.L. Fiadeiro and T. Maibaum, "Categorical Semantics of Parallel Program Design," *Science of Computer Programming,* vol. 28, nos. 2-3, pp. 111-138, Apr. 1997.

[10] R. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," PhD thesis, Univ. of California, Irvine, June 2000.

[11] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.,* vol. 24, no. 5, pp. 342-361, May 1998.

[12] IONA Orbix/E Datasheet, http://www.iona.com/whitepapers/orbixe-DS.pdf, 2004.

[13] E.A. Lee, "Embedded Software," *Advances in Computers,* E. Zelkowitz, ed., vol. 56, 2002.
[14] LIME, http://lime.sourceforge.net/, 2004.
[15] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, Jan. 1973.
[16] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich, "XMIDDLE: A Data-Sharing Middleware for Mobile Computing," *Personal and Wireless Comm.,* vol. 21, no. 1, pp. 77-103, 2002.
[17] N. Medvidovic, N.R. Mehta, M. Mikic-Rakic, "A Family of Software Architecture Implementation Frameworks," *Proc. Working IEEE/IFIP Conf. Software Architecture,* vol. 224, pp. 221-235, Aug. 2002.
[18] N. Medvidovic, M. Mikic-Rakic, N.R. Mehta, and S. Malek, "Software Architectural Support for Handheld Computing," *IEEE Computer,* pp. 66-73, Sept. 2003.
[19] N. Medvidovic, D.S. Rosenblum, and R.N. Taylor, "A Language and Environment for Architecture-Based Software Development and Evolution," *Proc. Int'l Conf. Software Eng.,* pp. 44-53, May 1999.
[20] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 70-93, Jan. 2000.
[21] N.R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," *Proc. Int'l Conf. Software Eng.,* pp. 178-187, June 2000.
[22] Microsoft.NET, http://www.microsoft.com/net/, 2004.
[23] M. Mikic-Rakic and N. Medvidovic, "Support for Disconnected Operation via Architectural Self-Reconfiguration," *Proc. First Int'l Conf. Autonomic Computing (ICAC-04),* pp. 114-121, May 2004.
[24] M. Mikic-Rakic and N. Medvidovic, "Adaptable Architectural Middleware for Programming-in-the-Small-and-Many," *Proc. ACM/IFIP/USENIX Int'l Middleware Conf.,* vol. 2672/2003, pp. 162-181, June 2003.
[25] M. Mikic-Rakic and N. Medvidovic, "Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach," *Proc. 2001 Symp. Software Reusability (SSR 2001),* pp. 11-18, May 2001.
[26] Object Computing Inc., http://www.theaceorb.com, 2004.
[27] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Intelligent Systems,* vol. 14, no. 3, pp. 54-62, May 1999.
[28] D. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Eng. Notes,* vol. 17, no. 4, pp. 40-52, Oct. 1992.
[29] L.L. Peterson and B.S. Davie, *Computer Networks.* Morgan Kaufmann, 2000.
[30] Project JXTA, http://www.jxta.org/, 2004.
[31] D. Salomon, *Data Compression: The Complete Reference.* Springer Verlag, Dec. 1997.
[32] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Eng.,* vol. 21, no. 4, Apr. 1995.
[33] D. Schmidt, "ACE," http://www.cs.wustl.edu/schmidt/ACE-documentation.html, 2004.
[34] D.C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-Determinism in Real-Time Object Request Brokers," *J. Real-Time Systems,* vol. 21, pp. 77-125, 2001.
[35] D. Schmidt, "TAO," http://www.cs.wustl.edu/schmidt/TAO.html, 2004.
[36] J.P. Sousa and D. Garlan, "Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments," *Proc. Working IEEE/IFIP Conf. Software Architecture,* pp. 29-43, Aug. 2002.
[37] Sun Microsystems, JINI(TM) Network Technology, http://wwws.sun.com/software/jini/, 2004.
[38] V. Sankhla, "SMART: A Small World Based Reputation System for MANETs," master's thesis, Dept. of Electrical Eng., Univ. of Southern California, Oct. 2004.
[39] C. Szyperski, *Component Software—Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
[40] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., and J.E. Robbins, "A Component- and Message-Based Architectural Style for GUI Software," *IEEE Trans. Software Eng.,* vol. 22, pp. 390-406, June 1996.
[41] Y. Weinsberg and I. Ben-Shaul, "A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices," *Proc. Int'l Conf. Software Eng. (ICSE 2002),* pp. 374-384, 2002.
[42] S.S. Yau and F. Karim, "Context-Sensitive Middleware for Real-Time Software in Ubiquitous Computing Environments," *Proc. Int'l Symp. Object-Oriented Real-Time Distributed Computing,* p. 163, May 2001.



Sam Malek received the MS degree in computer science from the University of Southern California in 2004 and the BS degree from the University of California, Irvine, in 2000. Currently, he is a PhD student in the Computer Science Department at the University of Southern California. His research interests are in the design, construction, and adaptation of large-scale distributed software systems. He is a member of ACM and ACM SIGSOFT.

Marija Mikic-Rakic received the PhD degree in 2004 from the University of Southern California. She works as a software engineer at Google, Inc. Her research interests are in the area of software architectures, with specific focus on architecture-based software development support for highly distributed, mobile, and resource constrained environments. She is a member of ACM and ACM SIGSOFT.

Nenad Medvidovic received the PhD degree in 1999 from the University of California, Irvine. He is an associate professor in the Computer Science Department at the University of Southern California. He is a recipient of the US National Science Foundation CAREER award. His research interests are in the area of architecture-based software development. His work focuses on software architecture modeling and analysis; middleware facilities for architectural implementation; product-line architectures; architectural styles; and architecture-level support for software development in distributed, mobile, resource constrained, and embedded computing environments. He is a member of the ACM, ACM SIGSOFT, and IEEE.