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 highly efficient and scalable implementation-level support for the key aspects of Prism application architectures, including their architectural styles. Additionally, Prism-MW is easily 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.

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 programming-in-the-small-and-many (Prism), both for exposition purposes, but also in order to distinguish it from the traditional software engineering paradigm of programming-in-the-large (PitL) [9], 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 [32]. An architecture is described in terms of software components (computational elements) [39], software connectors (interaction elements) [22], and their configurations (also referred to as topologies) [21]. 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 [28,32]. Examples of widely used styles are event-based, client-server, pipe-and-filter, peer-to-peer, blackboard, and so on.

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 [14,19,36]. 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 [18,31]. This is particularly important in the Prism setting: Prism systems may be highly distributed, decentralized, mobile, and long-lived, increasing the risk of architectural drift [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 easily tailorable to provide native implementation-level support for arbitrary architectural styles. This allows software developers to directly transfer architectural decisions into implementations, thus distinguishing Prism-MW from existing middleware solutions, including a previously published version of our own work [24].

Another key contribution of Prism-MW is its highly modular design that employs an extensive separation of concerns. This results in a middleware that is flexible, efficient, scalable, and extensible. The middleware is flexible in its support for independent selection, variation, and composition of implementation-level concerns. The middleware is efficient in its size, speed, and overhead added to an application. The middleware is scalable in the numbers of components, connectors, events, execution threads, and hardware devices. Finally, the middleware is easily extensible to support new development concerns and situations in the Prism setting, 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 [11]. At the same time, our experience with and evaluations of Prism-MW have suggested several areas of further study, including an entirely novel approach to designing architectural middleware. We intend to explore these issues in our future work.

The rest of the paper is organized as follows. Section 2 presents our objectives. Section 3 briefly describes an example application used to illustrate the concepts throughout the paper. Section 4 presents the design and implementation of Prism-MW’s core capabilities. Section 5 describes our support for architectural styles. Section 6 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. We briefly highlight some of the pertinent challenges, both in order to sensitize the reader and to motivate our objectives for Prism-MW.

Devices on which Prism applications reside may have limited power, network bandwidth, processor speed, memory, and display size and resolution. Constraints such as these force the developers of small, mobile platforms to make many trade-offs, possibly resulting in the omission of certain common services for reasons of efficiency. An example is Java KVM’s lack of support for non-integer numerical data types or server-side sockets. These constraints also demand highly efficient software systems, in terms of computation, communication, and memory footprint, possibly leading to unorthodox solutions such as “off-loading” non-essential parts of a system to neighboring devices.

The trade-offs made to address the scarcity of computing resources directly result in a highly heterogeneous computing environment. The world of PitL has largely addressed heterogeneity through standardized solutions such as Java, CORBA, data formats such as PDF and XML, network protocols such as HTTP, and so on. On the other hand, the world of Prism is still characterized by proprietary operating systems (e.g., PalmOS, Symbion), 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). Providing effective software development methods, techniques, and tools for such an environment becomes imperative.

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. Certain techniques recently devised for dealing with these problems, such as a component-based focus in supporting system design and implementation, may prove effective in the context of Prism. However, the manner and extent to which those techniques must be adapted remain open issues.

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 [32]) appear to make it a good fit for the needs of Prism. Software architecture indeed forms the centerpiece of our approach:

**Objective 1.** Prism-MW should provide native support for designing and implementing architectural abstractions (components, connectors, communication events, and so on). Furthermore, as recent studies (e.g., [1,19]) 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 architectural 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 (“programming”) 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 (“small” and “many”). We do so via three additional objectives. These objectives are “secondary” in that they have significantly impacted the design choices we have made for Prism-MW, but, unlike the first objective, they are not directly tied to the software development constructs available and visible to application developers.

**Objective 2.** Prism-MW should impose minimal overhead on an application’s execution. Our current goal is to enable efficient execution of applications on platforms with varying characteristics (e.g., speed, capacity, network bandwidth). The ultimate goal is to extend this support to include efficient access to and sharing of hardware resources (e.g., battery, peripheral devices).

**Objective 3.** Prism-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.** Prism-MW should be extensible and configurable in order to accommodate the many and varying development concerns across the heterogeneous Prism domain. These include multiple architectural styles (recall Objective 1), but also awareness, mobility, dynamic reconfigurability, security, real-time support, and delivery guarantees [3,4,10,14,15,27].

The discussion of Prism-MW is organized around these four objectives. The overall design of the middleware (Section 4) and its support for architectural styles (Section 5) are discussed first, and then the middleware’s support for the last three objectives is evaluated (Section 6). This discussion is illustrated with the example Prism 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, intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. The specific instance of this application depicted in Figure 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. Figure 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.\(^1\) A Map component maintains a model of the system's overall resources: terrain, personnel, tank units, and mine fields. 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 SAKUI components store the strategy rules and provide the user interface for changing these rules, respectively. ResourceManager, CommanderManager, SoldierManager, and ResourceMonitor components enable 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.

\(^1\) Note that the identified components are required for implementing TDS’s functionality, regardless of the employed architectural style(s). We will discuss different aspects of TDS, including its architecture, throughout the paper.
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 O/Ss (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, with methods for creating, manipulating, and destroying the element. These abstractions enable direct mapping between an architecture and its implementation. Figure 2 shows the class design view of Prism-MW. The shaded classes constitute the middleware core, which represents a minimal subset of Prism-MW that enables implementation and execution of architectures in a single address space. Only the dark gray classes of Prism-MW’s core are directly relevant to the application developer, requiring a minimal effort to master the middleware’s basics. Our goal was to keep the core compact, reflected in the fact that it contains only twelve classes (four of which are abstract) and four interfaces. Furthermore, the design of the core (and the entire middleware) is highly modular: we have tried to limit direct dependencies among the classes by using abstract classes, interfaces, and inheritance as discussed below.

4.1. Middleware Core

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 event’s payload includes a set of typed parameters for carrying data and meta-level information (e.g., sender, type, and so on). An event type is either a request for a recipient component to perform an operation or a reply that a sender component has performed an operation.
Ports are the loci of interaction in an architecture. A link between two ports is made by welding them together. A port can be welded to at most one other port. Each Port has a type, which is either request or reply. An event placed on one port is forwarded to the port linked to it in the manner shown in Figure 3: 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 whose...

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

Figure 3. Link between two Ports in Prism-MW.
type corresponds to the generated event 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. Components attach to connectors by creating a link between a component port and a single connector port. Connectors may support arbitrary event delivery semantics (e.g., unicast, multicast, broadcast). In order to support the needs of dynamically changing applications, each Prism-MW component or connector is capable of adding or removing ports at runtime. This property of components and connectors, coupled with event-based interaction, has proven to be highly effective for addressing system reconfigurability.

Each subclass of the Brick class has an associated interface. The IArchitecture interface exposes a weld method for attaching two ports together. The IComponent interface exposes send and handle methods used for exchanging events. Component provides the default implementation of IComponent’s send method: generated request events are placed asynchronously on all of the request ports attached to the component, while generated reply events are placed asynchronously on all of the attached reply ports. As will be detailed in Section 6.3, we provide other implementations of this interface, including synchronous sending of events. The IConnector interface provides a handle method for routing of events. The Connector class provides the default implementation of the IConnector’s handle method, which forwards all request events to the connector’s attached request ports and all reply events to the attached reply ports. As will be detailed in Section 6.3, 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: FIFO Scheduler 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 single event queue can be instantiated for the entire architecture; alternatively, a separate event queue can be assigned to each component. Additionally, dispatching and scheduling are decoupled from the Architecture, allowing one to easily compose many sub-architectures (each with its own scheduling and dispatching policies) in a single application. Scaffold also directly aids architectural awareness [3] (also referred to as reflection) by allowing probing of the runtime behavior of a Brick via different implementations of the AbstractMonitor class.
Prism-MW’s core has been implemented in Java JVM. Subsets of the described functionality have also been implemented in Java KVM, C++, EVC++, and Python; they have been used in example applications and in evaluating Prism-MW. The implementation of the middleware core is quite small (under 900 SLOC), which aids Prism-MW’s understandability and ease of use.

Figure 4. Prism-MW application implementation fragments.
4.2. Using Prism-MW

Prism-MW’s core provides the necessary support for developing arbitrarily complex applications, so long as they rely on the default facilities (e.g., event scheduling, dispatching, and routing) and stay within a single address space. The first step a developer takes is to create the application-specific portion of each component by subclassing from the AbstractImplementation class and providing the component’s “business logic” inside its handle and start methods. The developer then instantiates the Prism-MW Components and associates each Component with its implementation. Next, the developer instantiates the Architecture class and adds the Components (and Connectors if needed) to the Architecture. The developer then creates instances of request and reply Ports for components (and connectors if they are used in a given architecture), and associates them with their container Components (or Connectors), using the addPort method. Finally, attaching component and connector Ports into a configuration is achieved by using the weld method of the Architecture class.

For illustration, Figure 4 shows two alternative usage scenarios in the Java version of Prism-MW. The code fragments correspond to two different implementations of a subset of the TDS application, where the instantiation and interaction of three components (RenderingAgent, DeploymentAdvisor, ResourceMonitor) is shown. In the first usage scenario the components are communicating directly, while in the second scenario the components are communicating through a connector. The alternative implementations of the TDSDemo class’s main method instantiate components, ports (and, in the second case, connectors) and compose (weld) them into a configuration. Figure 4 also demonstrates event-based communication between the two components (as implemented in RenderingAgentImpl and DeploymentAdvisorImpl classes). Component RenderingAgent creates and sends a request event, in response to which Component DeploymentAdvisor sends a reply event. An event need not identify its recipient components; they are uniquely defined by the topology of the architecture and routing policies of the employed connectors [24].

4.3. Extensibility Mechanism

The design of Prism-MW’s core provides extensive separation of concerns via its explicit architectural constructs and its use of abstract classes and interfaces. The design is highly extensible. To date, we have built several specific extensions to support architectural awareness, real-time requirements, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility [3,4,10,14,15,27]. Furthermore, the extensible nature of Prism-MW has enabled us to directly support multiple architectural styles, even within a single application. In this section we describe our approach to supporting extensibility in Prism-MW, depicted in Figure 5. We explicitly focus on leveraging that extensibility to support architectural styles in the next section, while other extensions are detailed in Section 6.3. Our experience
with the specific extensions we have built to date indicates that other extensions 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, Extensible-Connector, ExtensiblePort, ExtensibleEvent, and ExtensibleArchitecture), each of which has a reference to a number of abstract classes (AbstractExtensions in Figure 5). Each AbstractExtension class can have multiple implementations (Extension i,j in Figure 5), thus enabling selection of the desired functionality inside each instance of a given Extensible class. If a reference to an AbstractExtension class is instantiated in a given Extensible class instance, that instance will exhibit the behavior realized inside the implementation of that abstract class. Multiple references to abstract classes may be instantiated in a single Extensible class instance. In that case, the instance will exhibit the combined behavior of the installed abstract class implementations.

5. Support for Architectural Styles

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems’ requirements [28,32]. Therefore, a middleware platform used to implement such architectures would need to support multiple styles. Prism-MW’s flexible and extensible design can be easily leveraged to support a number of distributed systems styles [11], which are likely to be useful in the Prism setting [19]. In this section we describe how Prism-MW can be configured to support different architectural styles. We illustrate our approach on five representative distributed systems styles: client-server, pipe-and-filter, publish-subscribe, peer-to-peer, and C2 [11]. Our experience suggests that the same technique can be used to support other styles.

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, and so on. 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 Figure 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 Figure 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 inter-process communication (see Figure 6c). This partially satisfies the second requirement by supporting architectural styles that require distribution (e.g., a Server should be able to serve many distributed Clients).

- **ExtensibleArchitecture** has an associated implementation of the AbstractTopology class to ensure the topological constraints of a given style (see Figure 6d). For example, in the client-server style, Clients can connect to Servers,
but two Clients cannot be connected to one another. This satisfies the third requirement by allowing for the specification and modification of valid configurations of architectural elements.

- **ExtensibleArchitecture** implements the IComponent interface, thereby allowing hierarchical composition of components (see Figure 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.

```java
public static ExtensibleArchitecture generateArchitecture (String name, int style)
{
    AbstractTopology topology;
    switch (style)
    {
    case PrismConstants.CLIENT_SERVER_ARCH:
        topology = new ClientServerTopology();
        return (new ExtensibleArchitecture(name, topology, style));
    case PrismConstants.C2_ARCH:
        topology = new C2Topology();
        return (new ExtensibleArchitecture(name, topology, style));
    case PrismConstants.PIPE_FILTER_ARCH:
        topology = new PipeFilterTopology();
        return (new ExtensibleArchitecture(name, topology, style));
    case PrismConstants.PUB_SUB_ARCH:
        topology = new PubSubTopology();
        return (new ExtensibleArchitecture(name, topology, style));
    case PrismConstants.P2P_ARCH:
        topology = new P2PTopology();
        return (new ExtensibleArchitecture(name, topology, style));
    default:
        return null;
    }
}

public class ClientServerTopology extends AbstractTopology
{
    public void weld(Brick b1, Brick b2) throws PrismException
    {
        int b1Style = b1.getStyle();
        int b2Style = b2.getStyle();
        if ((b1Style != PrismConstants.CLIENT & b1Style != PrismConstants.SERVER) ||
            (b2Style != PrismConstants.CLIENT & b2Style != PrismConstants.SERVER))
        {
            throw new PrismException("At least one of the bricks is neither a client nor a server");
        }
        else if (b1Style == PrismConstants.CLIENT & b2Style == PrismConstants.CLIENT)
        {
            throw new PrismException("Cannot weld two clients directly");
        }
        else
        {
            if (b1Style == PrismConstants.CLIENT & b2Style == PrismConstants.SERVER)
            {
                Port clientRequest = new Port("clientRequest", PrismConstants.REQUEST);
                ((IComponent)b1).addPort(clientRequest);
                Port serverReply = new Port("serverReply", PrismConstants.REPLY);
                ((IComponent)b2).addCompPort(serverReply);
                clientRequest.setMutualPort(serverReply);
                serverReply.setMutualPort(clientRequest);
            }
            else
            {
                Port clientRequest = new Port("clientRequest", PrismConstants.REQUEST);
                ((IComponent)b1).addCompPort(clientRequest);
                Port serverReply = new Port("serverReply", PrismConstants.REPLY);
                ((IComponent)b2).addPort(serverReply);
                clientRequest.setMutualPort(serverReply);
                serverReply.setMutualPort(clientRequest);
            }
        }
    }
}
```

Figure 7. a) StyleFactory class API; b) its implementation of generateArchitecture method; c) ClientServerTopology class and its implementation of the weld method.
5.1. Tailoring Prism-MW to Support Individual Styles

We illustrate our support for architectural styles in Prism-MW by detailing the specific extensions shown in Figure 6. To produce a style-specific architectural element, the developer instantiates the corresponding Extensible class and sets the desired stylistic behavior by installing the appropriate extensions on it. To simplify this task, we have provided a StyleFactory utility class (Figure 6e) that can automatically generate style-specific architectural elements. The API of the StyleFactory class, referring to the five styles from Figure 6, is shown in Figure 7a. Figure 7b shows the implementation of StyleFactory's generateArchitecture method. For illustration, Figure 7c shows the implementation of the ClientServerTopology class, which ensures topological rules of the client-server style. In the remainder of this section we describe the support for each one of the five styles in more detail.

5.1.1. Client-Server Style

The client-server style is the most frequently encountered style in network-based applications [11]. 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 non-blocking behavior of a component in Prism-MW. 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, and 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. Figure 8 shows a distributed architecture in the client-server style, and the corresponding code in Prism-MW. To communicate with each other, Clients and Servers can use regular ports (recall Section 4.1) for applications residing in a single address space. More realistically, they would use distribution enabled ports (shown in Figure 8, and further discussed in Section 6.3). A Server's distribution enabled port has a thread that listens for incoming connection requests. A Client uses its distribution enabled port to connect to remote server ports and send request events. A client-server architecture is composed of an ExtensibleArchitecture with the ClientServerTopology implementation of the AbstractTopology. ClientServerTopology (see Figure 7c) enables welding of Clients and Servers while enforcing the topological rules (e.g., disallowing the welding of two Clients).
5.1.2. Pipe-and-Filter Style

In the pipe-and-filter style, each filter (component) reads streams of data (events) on its input ports and produces streams of data on its output ports, usually while applying a transformation on the input streams [11]. A pipe (connector) is used to connect an output port (source) of one filter to an input port (sink) of another filter.

The pipe-and-filter style is supported in Prism-MW through a style specific component (Filter) and a style specific connector (Pipe). A Filter is an ExtensibleComponent, which may use the implementation of the AbstractComponentSynchronism to block when it writes to a Pipe. A Pipe is composed from an ExtensibleConnector with single request and reply ports and the UnidirectionalBroadcastHandler implementation of AbstractHandler. UnidirectionalBroadcastHandler is configured to only broadcast request events received on one of its connector’s request ports to all of the connector’s reply ports; at the same time, it does not propagate any reply events. The connector can be further extended to support data buffering [32]. A pipe-and-filter architecture in Prism-MW is an ExtensibleArchitecture with an associated implementation of AbstractTopology called PipeFilterTopology. PipeFilterTopology provides the ability to weld a Pipe and a Filter together. It also provides the topological constraint checking logic for the pipe-and-filter.

---

2 Our implementation of the pipe-and-filter style does not use synchronous Filters.
style that disallows the welding of two Filters, or two Pipes, together. Figure 9, shows the architecture of a pipe-and-filter style application and the corresponding code in Prism-MW.

5.1.3. C2 Style

The C2 style [40] is a vertically oriented network of concurrent components that communicate via neighboring connectors. Communication is achieved through asynchronous notification events going down an architecture, and asynchronous request events going up. A C2 component has only one port on its top and one port on its bottom. The top port is used to send requests and receive notifications, while the bottom port is used to receive requests and send notifications. The C2 style is also characterized by substrate independence, i.e., a component in one layer is not dependent on components in the layers below it.

The C2 style is supported in Prism-MW through a style specific component (C2Component) and a style specific connector (C2Connector). A C2Component is an ExtensibleComponent with a single request and a single reply port. A C2Connector is an ExtensibleConnector with BidirectionalBroadcastHandler implementation of AbstractHandler. BidirectionalBroadcastHandler broadcasts request events received on one of the connector’s request
ports to all of the connector’s reply ports, and broadcasts reply events received on one of the connector’s reply ports to all of the connector’s request ports. A top port on a C2 component or connector corresponds to Prism-MW’s request port, while a bottom port corresponds to Prism-MW’s reply port. A C2 style architecture is created as an ExtensibleArchitecture with a C2Topology implementation of AbstractTopology. C2Topology provides the ability to create C2 style configurations. C2Topology’s weld method provides the ability to weld a C2Component and a C2Connector, or to weld two C2Connectors directly. The weld method disallows welding two C2Components together. It also disallows more than one request and one reply port for each C2Component, as mandated by the style. Each call of the weld method automatically creates and associates the needed request and reply ports to C2Components and C2Connectors. Figure 10 shows the architecture of a C2 style application and corresponding code in Prism-MW.

5.1.4. Publish-Subscribe Style

In the publish-subscribe style [4] components communicate by subscribing to events of interest, which are published by other components. Channels provide the framework for publication/subscription of events. Each component may be a producer as well as a consumer of events.

The publish-subscribe style is supported in Prism-MW through a style-specific component (PubSubComponent) and a style-specific connector (Channel). A PubSubComponent is composed of an ExtensibleComponent.
Channel is an ExtensibleConnector with MulticastHandler implementation of AbstractHandler. MulticastHandler routes events received on one of the connector’s ports to all of the connector’s ports whose associated brick has subscribed to that event. A PubSubComponent subscribes to an event published (routed) on a Channel by sending a subscription event to the Channel. Subscription events carry the name of the application-level event to which the PubSubComponent is subscribing. Unsubscription events are sent by PubSubComponents to unsubscribe from an event. A publish-subscribe architecture is composed of an ExtensibleArchitecture with a PubSubTopology implementation of AbstractTopology. PubSubTopology provides the ability to weld a PubSubComponent and a Channel together. PubSubTopology also provides the topological constraint checking logic for the style. For example, it disallows direct welding of two PubSubComponents together. Figure 11 shows the architecture of a publish-subscribe style application and the code required to realize it in Prism-MW.

5.1.5. Peer-to-Peer Style

In the peer-to-peer style each component can both request services from and provide services to other components [11].
The peer-to-peer style is supported in Prism-MW through a style specific component (Peer). A Peer is composed of an ExtensibleComponent. A peer-to-peer architecture is composed of an ExtensibleArchitecture with a P2PTopology implementation of AbstractTopology. P2PTopology provides the ability to weld two Peers together. Each weld results in the creation of one request and one reply port on each Peer. Since the peer-to-peer style does not have any topological rules, P2PTopology does not perform any topological constraint checking and allows for arbitrary configurations of Peers. Figure 12 shows the architecture of a peer-to-peer application and the corresponding code in Prism-MW.

5.1.6. Summary

Each one of the styles described in this section 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 1 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.

5.2. Support for Multiple Architectural Styles in a Single Application

In a complex, large-scale system, multiple architectural styles may be required to facilitate different subsystems’ requirements [28,32]. It has been shown that architectural styles employed in a system’s design and middleware plat-

---

3 The implementations of the five discussed styles were performed by persons experienced with Prism-MW.
forms selected for the system’s implementation can significantly impact one another [8]. However, existing middleware platforms do not provide any support for different styles: they either do not enforce any stylistic rules (e.g., TAO [35], .NET [23]), or only provide support for a single architectural style (e.g., Aura [36], C2 framework [18]). This can cause problems, or at least inefficient implementations if a single style is not suitable for the entire application.

Prism-MW supports the use of multiple architectural styles in a single application by leveraging hierarchical composition. ExtensibleArchitecture implements the IComponent interface (recall Figure 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 show 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]. Figure 13 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. These modifications are discussed in [19]. Here we will illustrate the

![Architecture of TDS in the C2 style with single Headquarters, Commander, and Soldier subsystems. Prism-MW Ports have been elided for clarity.](image)

**Figure 13.** Architecture of TDS in the C2 style with single Headquarters, Commander, and Soldier subsystems. Prism-MW Ports have been elided for clarity.
approach to supporting multiple styles in a single Prism-MW application by detailing the modifications made to the internal architecture of the Headquarters subsystem.

The Clock component in the Headquarters subsystem sends periodic ticks to Weather, ResourceMonitor, and SimulationAgent components. However, since these components are at different levels of the architecture, addition of connector-to-connector links is required to enable the delivery of clock ticks to all its recipients. Due to the broadcast policy of the employed C2 connectors, this design resulted in substantial event traffic, much of which simply got ignored by the recipient components. For example, the Clock component was receiving all requests originating from any other component in the Headquarters subsystem, although it was not intended (or implemented) to process any requests. Similarly, clock ticks were received by seven components that do not need (or respond to) this event. Likewise, addition of connector-to-connector links increased the unnecessary event traffic for the remaining events exchanged among TDS components.

We leveraged Prism-MW’s support for multiple styles within a single application and moved the Clock component in a separate pipe-and-filter style subarchitecture. This enabled us to remove two “offending” connector-to-connector links in the remaining C2 style subarchitecture of the Headquarters and route clock ticks directly. However, we needed to create the hybrid (C2PF) style to accommodate the Weather, SimulationAgent, and ResourceMonitor components, which now needed to engage in both C2 style and pipe-and-filter style interactions. These changes were effected by modifying approximately 20 SLOC in the architecture instantiation class, and by adding another 20 SLOC for supporting the hybrid style. The modifications resulted in over 60% 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).

Figure 14. Architecture of the Headquarters subsystem in two styles: Pipe-and-Filter and C2. Ports have been elided for clarity.
6. Evaluation

The previous two sections have discussed Prism-MW’s support for our key objective ("programming" architectural abstractions). In this section, we evaluate Prism-MW with respect to the remaining three objectives.

6.1. Efficiency

Since Prism applications frequently run on resource-constrained devices, with low amounts of memory and slow processing speeds, we have performed several optimizations on Prism-MW’s core. While there are common techniques for ensuring efficient implementations of distributed systems, Prism-MW presented unique challenges in this regard because of its added objective of directly supporting architectural abstractions in highly resource-constrained settings. Some of the optimization techniques we applied are novel, while others have been adapted from existing work. A contribution of our work on optimizing Prism-MW lies in their combination: it results in a highly efficient architectural middleware that introduces minimal overhead in terms of dynamic memory usage and shows good performance. In the remainder of this section we describe these optimizations and provide a series of benchmark results that evaluate the resulting efficiency of Prism-MW.

6.1.1. Initial Implementation

In our initial implementation of Prism-MW’s core, each component maintained dynamically allocated queues of its incoming and outgoing events. Each component also owned an internal thread of control that was used to process incoming events and place outgoing events on the queue (as implemented in IComponent’s send and handle methods, respectively). The encompassing Architecture’s Dispatcher then ensured that the outgoing events are routed to their destinations. Furthermore, the Architecture’s implementation of the Scheduler was trivial since all the scheduling was handled at the individual component level. However, this implementation had several problems, including unacceptable application size and speed [18]. Prism-MW’s highly modular design allowed us to significantly improve efficiency by radically altering the manner in which events are exchanged and processed. At the same time, we were able to confine our modifications to the implementations of IComponent (specifically, its send method), AbstractScheduler, and AbstractDispatcher. These modifications are discussed below.

6.1.2. Optimizing for Size and Speed

We observed that a large amount of dynamic system memory usage was a result of the exchange of events among components and connectors. We minimized the required memory for event passing by exchanging read-only events in the same address space by reference, rather than by copy. We further optimized memory usage by adopting a fixed-sized, circular array for storing all events in a single address space. This reduced overall memory usage by a factor of 20 or more over the initial solution described above [18].
Another modification addressed event processing. A pool of shepherd threads (implemented in Prism-MW’s RoundRobinDispatcher class) was introduced to handle events sent by any component in a given address space. The size of the thread pool is parameterized and, hence, adjustable. It should be noted that concurrency management of the circular array used to implement the event queue slightly impacts the speed of processing by applying a producer-consumer algorithm to keep event production under control, and supply shepherd threads with a constant stream of events to process.

Figure 15 shows event processing for two alternative usage scenarios of Prism-MW introduced in Section 4.1. To process an event, a shepherd thread removes the event from the head of the queue. For local communication, in the first scenario the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread (see Figure 15a). If a recipient component generates further events, they are added to the tail of the event queue; different threads are used for dispatching those events to their intended recipients. The second usage scenario (shown in Figure 15b) uses direct connections between component ports and allows separate threads to be used for dispatching an event from the queue to each intended recipient component (steps 2-3 a and b in Figure 15b). This increases parallelism, but also resource consumption in the architecture.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: a sending component places its outgoing event to an attached port. However, in this case the port is a DistributionEnabledPort (illustrated in Figure 8 and further discussed in Section 6.3), which manages a network (e.g., socket or infrared) connection. Thus, instead of depositing the event to the local event queue, the distribution enabled port deposits the event on the network. As the event is propagated across the network, the DistributionEnabledPort on the recipient device uses its internal thread to retrieve the incoming event and place it on its local event queue.

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.

2. We further optimize resource consumption by using a single event queue for storing both locally and remotely generated events.

3. Since Prism-MW processes local and remote events uniformly, and all routing is accomplished via the multiple and explicit ports and/or connectors, Prism-MW allows for seamless redeployment and redistribution of existing applications onto different hardware topologies.

The above optimizations resulted in a very light-weight Prism-MW implementation that has shown several orders of magnitude in performance improvement over the original implementation outlined in Section 6.1.1. More importantly, the performance of Prism-MW is now comparable to solutions using a plain programming language (PL). Each Prism-MW event exchange causes five PL-level method invocations (typically highly optimized in a PL), and a comparatively more expensive context switch if the architecture is instantiated with more than one shepherd thread. 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.

6.1.3. Performance Measurements

For illustration, we describe the results from several series of benchmark tests we have conducted to measure the size and performance of the Java Prism-MW implementation. Figures 16 and 17 show the result of running an architecture where one component is sending varying numbers of events to a varying number of identical recipient components through a connector (Figure 15a shows such an architecture with two recipient components). The architecture used a pool of 10 shepherd threads and a queue of 1000 events ($q_{size}$). In Figure 16, between 1 and 100,000 simple (parameter-less) events were sent asynchronously by the single sender component to 100 recipient components (resulting in between 100 to 10,000,000 invocations of component `handle` methods) for the application running on a mid-range desktop PC. While the 10 million events are processed in under 3 seconds on the PC, such performance clearly cannot be expected of less capacious and performant platforms. However, as Figure 17 shows, Prism-MW also exhibits very good performance on such platforms. Figure 17 depicts the result of executing the application with the same architecture on a

---

4 The five method invocations involve traversing the ports, placing the event in the event queue, and dispatching the event to the recipient component.
handheld “Pocket PC”; a maximum of 10,000 events are sent to a maximum of 100 components (resulting in up to 1,000,000 invocations of component handle methods).

To illustrate the performance trade-off between two alternative usage scenarios of Prism-MW (recall Figures 4 and 15), we created a benchmark application in which a single component communicated with a varying number \( n \) of identical components. Figure 18 shows the results of a test in which the communication took place through a single connector, while Figure 19 shows the same benchmark without the connector (i.e., components are connected via ports only). One parameter-less event was sent asynchronously by the single component to all other components, resulting in \( n \) events being handled. Each one of the \( n \) components was implemented with a fixed event handling delay in of 50 msec, to simulate processing in each component \( \text{comp_proc_time} \) and to utilize benefits of parallel processing. 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 Figure 19) can be approximated using the following formula

\[
\text{total\_proc\_time} = \begin{cases} 
\text{comp\_proc\_time}, & \text{if } \text{numComps} < \text{numThreads} \\
\text{comp\_proc\_time} \times \text{numComps} / \text{numThreads}, & \text{if } \text{numComps} \geq \text{numThreads}
\end{cases}
\]

where \( \text{numComps} \) represents the number of components, and \( \text{numThreads} \) the number of shepherd threads in an architecture.

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 (\( \text{comp\_mem} \)), without any application-specific methods or state, is 0.12 KB. Memory overhead of a “base” connector (\( \text{conn\_mem} \)) is 0.09 KB, while the memory overhead of a “base” port (\( \text{port\_mem} \)) is 0.04 KB. Memory overhead of creating and sending a single event (\( \text{evt\_mem} \)) can be estimated using the following formula, obtained empirically:

\[
\text{evt\_mem (in KB)} = 0.04 + 0.01 \times \text{num\_of\_parameters}
\]

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 Figures 13 and 14), which consisted of a single Headquarters subsystem, four Commander subsystems, and 100 Soldier subsystems can be closely approximated as follows:

\[
\text{num\_arch} \times (mw\_mem + (q\_size \times \text{evt\_mem})) + \text{num\_comps} \times \text{comp\_mem} + \text{num\_conns} \times \text{conn\_mem} + \text{num\_ports} \times \text{port\_mem} = 105 \times (2.3 + (25 \times (0.04 + (0.01 \times 1))) + (245 \times 0.12) + (217 \times 0.09) + (875 \times 0.04) = 457 \text{ KB}
\]

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 3.5% overhead on the application’s dynamic memory consumption.

6.2. Scalability

Prism-MW’s modularity and separation of concerns directly aid its scalability in the numbers of supported devices, components, connectors, ports, threads, and events. Prism-MW’s support for large numbers of devices is a consequence

\[ ^5 \] In this sense, the measure represents minimum event overhead. Use of complex objects as event parameters is independent of the middleware, but is an application-level decision.
of its support for large numbers of distribution enabled ports and explicit connectors. Similarly, its scalability in the number of events is fostered by scalability in the number of threads. The below discussion reflects these relationships.

Unlike the existing middleware platforms (e.g., CORBA [43], LIME [15], .NET [23]), which support a single, implicit connector in a system, Prism-MW supports an arbitrary number of connectors and distribution enabled ports. Prism-MW’s explicit, flexible connectors and explicit ports allow an architecture to be deployed onto an arbitrary number of hosts, by repeated splitting of the connectors using the technique described in [7]. In a highly degenerate case, this would result in some devices serving only as routers, without containing any components. For this reason, the number of devices supported by Prism-MW is unlimited in principle. It should be noted, however, that the deployment choice directly affects efficiency: the performance gain of using the centralized event queue discussed above is achieved only if the components are residing in the same address space.

Realistically, the number of components on a given device is limited and can be estimated using the following simple formula: \( n = \frac{(M - MS)}{ACS} \), where \( M \) is the available memory on the device, \( MS \) is the memory occupied by Prism-MW, and \( ACS \) is the average component size. Recall from Section 6.1 that the impact of MS and the middleware-induced portion of ACS on the device’s memory consumption is very low. We have performed a series of benchmarks in order to assess the behavior of Prism-MW in cases where large numbers (up to 1 million) of components is used. Figures 18 and 19 show the results of tests with up to 10,000 components. We have also performed benchmark tests using “chains” consisting of \( n \) components that exchange events via \( n-1 \) connectors. For example, the total round-trip time for a single event in the case where \( n=100,000 \) and the entire architecture is in a single address space was 1.1 milliseconds on the hardware configuration described in Figures 18 and 19.

Prism-MW supports as many threads as the underlying platform supports. Finally, the number of events supported by Prism-MW is not limited by the middleware itself, but by the properties of the underlying hardware platform. This limit can be characterized by the following two parameters: (1) the maximum number of events that can simultaneously be present in a system and (2) the rate of event delivery. The maximum number of events is limited by the available memory on a given host (or set of hosts) and event size (recall Section 6.1), while the rate of event delivery depends on the CPU speed, the number of threads servicing the event queue, the ratio of event production to consumption by the components, and the network bandwidth for events that traverse machine boundaries.

6.3. Extensibility

In addition to architectural styles (discussed in Section 5), we have built several specific extensions to support architectural awareness, real-time requirements, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility [3,4,10,14,15,27]. Below we describe six classes of extensions supported by Prism-MW, with an
explicit focus on the extensions we have completed to date. Further details on these extensions may be found in [26]. We do not discuss the efficiency aspects of these extensions for two reasons. First, our primary goal has been to assess the extensibility of Prism-MW, and we have not optimized our implementations of many of its extensions. Secondly, in most cases our implementations have employed known algorithms and techniques, such that any performance measures would be a function of those algorithms and techniques rather than the inherent properties of Prism-MW.

As in the case of architectural styles, the six classes of extensions we describe below have been implemented using Prism-MW’s extensibility mechanism described in Section 4.3. The diversity and substantial number of implemented extensions strongly suggest that other extensions can be implemented using the same mechanism.

6.3.1. Port Extensions

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. Figure 20 shows five different port extensions we have implemented thus far.

The AbstractDistribution class (recall Section 5) has been implemented by two concrete classes, one supporting socket-based and the other infrared port-based inter-process 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; it has the ability to make connection requests to other DistributionEnabledPorts. 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 enables a DistributionEnabledPort to have an arbitrary number of connections to remote hosts.\(^6\) Note that a DistributionEnabledPort still belongs only to a single local Component or Connector.

The AbstractSecurity class has several implementations that perform combinations of authentication, authorization, encryption, and event integrity. These services have been implemented using three major cryptographic algorithms: sym-

\(^6\) We have also implemented an alternative design which allows only a single connection to a remote host. However, this implementation has been shown less efficient in practice.
metric key, asymmetric key, and event digest function. The AbstractDeliveryGuarantees class supports event delivery guarantees. We have implemented this class to support at most once, at least once, exactly once, and best effort delivery semantics. In order to support communication across PLs, we have added the AbstractXMLConversion class and implemented XML encoding/decoding of events inside the XMLConversion class. Finally, we have added the AbstractCompression class with the goal of minimizing the required network bandwidth for event dispatching. To this end, we have implemented the Huffman coding technique [30] inside the Compression class.

Addition of a new extension to the ExtensiblePort requires adding a reference to the appropriate abstract class and performing method calls on it inside ExtensiblePort’s handle method. Such a change to the ExtensiblePort class is minimal, averaging three new lines of code for each new extension. However, it is important to know the right ordering of method calls to achieve the desired behavior. For example, when combining AbstractSecurity and AbstractXMLConversion extensions, AbstractXMLConversion’s convert method is invoked before AbstractSecurity’s encrypt method when sending the event; on the receiving end, the AbstractSecurity’s decrypt method is invoked before AbstractXMLConversion’s reconstitute method.

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. We are currently implementing a tool that would perform this task automatically, given a specification of features that a port should support.

6.3.2. Connector Extensions

As outlined in Section 5, to support different event routing policies, we provide the ExtensibleConnector class that has a reference to an AbstractHandler class. To date we have provided several implementations of this class, including event multicast (recall Section 5.1.4), event unicast, unidirectional broadcast (recall Section 5.1.2), and bidirectional broadcast. (recall Section 5.1.3). These extensions are illustrated in Figure 21.

![Figure 21. Connector extensions.](image-url)
6.3.3. Component Extensions

To support various aspects of architectural awareness we have provided the ExtensibleComponent class that has references to several abstract classes (see Figure 22). As outlined in Section 5, ExtensibleComponent also contains a reference to Architecture, allowing its instances to act as meta-level components and to effect runtime changes on the system’s architecture. To date, we have augmented the ExtensibleComponent class with three extensions. The implementation of the AbstractComponentSynchronism class, used in the client-server style, enables synchronous communication, i.e., blocking of the sending component until it receives a response event. The implementation of the AbstractDeployment class is used for performing component deployment and mobility [25]. 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 recently implemented several versions of this interface that encapsulate different subsets of our DRADEL [20] environment.

6.3.4. Event Extensions

To support various facets of event delivery we have provided the ExtensibleEvent class that can compose multiple abstract classes. To date, we have created three abstract classes for ExtensibleEvent (illustrated in Figure 23). The implementation of the AbstractDelivery GuaranteesEvent class is used to assign a delivery guarantee policy to an event (i.e., at most once, at least once, exactly once, best effort). This class is used in tandem with AbstractDeliveryGuarantees of the ExtensiblePort class (see Figure 20). The AbstractRealTimeEvent class 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 additionally provided three classes that implement the AbstractScheduler and AbstractDispatcher classes, discussed below. Finally, to support communication across PL boundaries the implementation of the AbstractXMLRepresentation class provides an XML-based representation of an event.
6.3.5. Architecture Extensions

To support construction of arbitrarily complex architectures we provide the `ExtensibleArchitecture` class, which can compose multiple abstract classes. The `ExtensibleArchitecture` class implements the `IComponent` interface, thus allowing construction of hierarchical components, with internal architectures. To date we have provided the `AbstractTopology` class with different implementations to support several architectural styles. Details of these extensions are described in Section 5.

6.3.6. Other Extensions

In addition to the `AbstractDistribution` inside the `ExtensiblePort` class, to support distribution and mobility we have implemented the `Serializable` interface for both `Events` and `Bricks`. This allows us to send data as well as code across machine boundaries.

In support of real-time event delivery we have provided two additional implementations of the `AbstractScheduler` class. `EDFScheduler` implements scheduling of aperiodic events based on the earliest-deadline-first algorithm, while `RateMonotonicScheduler` implements scheduling of periodic events. Finally, `Scaffold` also directly aids architectural awareness [3] by allowing probing of the runtime behavior of a `Brick` via different implementations of the `AbstractMonitor` class. These extensions are shown in Figure 25.

7. Related Work

Our work on Prism-MW has been primarily influenced by two research areas: architectural styles and middleware. Architectural styles were discussed in Sections 1 and 5. Below we discuss two 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.

While we have found only the above two related approaches in the software architecture research literature, we have performed a comparison of Prism-MW with several representative middleware solutions with respect to the objectives identified in Section 2. In the remainder of this section, we briefly summarize each one of these middleware technologies, and provide their side-by-side comparison.

Orbix/E [13] is a lightweight CORBA ORB optimized for embedded applications. It is designed for rapid development and deployment support in both C++ and Java. Orbix/E has a relatively small memory footprint, which enables its use in memory constrained applications. Orbix/E provides the ability to choose a subset of features for a given application in order to optimize its size and speed.

ACE [33] is an object-oriented framework that implements many core patterns for concurrent communication software. The patterns and components in the ACE framework have been applied in the ACE ORB (TAO), which is a CORBA-compliant middleware framework. TAO allows clients to invoke operations on distributed objects without concern for object location, PL, OS platform, communication protocols, or hardware.

JXTA [29] is a set of open protocols that allow any connected device on the network, ranging from cell phones and wireless PDAs to PCs and servers, to communicate and collaborate in a peer-to-peer manner. JXTA peers create a virtual network where any peer can interact with other peers and resources directly, even when some of the peers and resources are behind firewalls or on different network transports. JXTA supports multiple platforms and languages, and ensures secure communication of collaborating peers.

Microsoft .NET [23] is a mechanism for integrating the OS with the software development environment and the Web. .NET enables a high level of software integration through the use of Web services and XML. .NET is divided into three different segments: (1) .NET Framework SDK, which is a language-agnostic execution engine for applications that are built in .NET languages (C#, VB.NET, C++ .NET, JScript.NET, and J#); (2) Web Services, which are classes that expose
their functionality through the Web; and (3) Visual Studio.NET, which provides support for building any type of .NET application in any .NET language.

Jini network technology [37] is an open architecture that enables developers to create network-centric hardware or software services that are highly adaptive to change. The Jini architecture specifies a way for clients and services to discover each other and to collaborate across the network. When a service joins a Jini network, it advertises itself by publishing an object that implements the service API. A client finds services by looking for an object that supports the API. When the client gets the service’s published object, it will download any code it needs in order to communicate with the service, thereby learning how to “talk” to the particular service implementation via the API.

XMIDDLE [17] is a data-sharing middleware for mobile computing. XMIDDLE allows applications to share data that are encoded as XML with other hosts, to have complete access to the shared data when disconnected from the network, and, when possible, to reconcile any changes made with all the hosts sharing the data. The goal is to make sure that eventually all hosts will have a consistent version of the shared data. XMIDDLE is lightweight and fast, and caters to the frequent disconnection behavior that mobile devices exhibit. XMIDDLE also allows applications to influence the reconciliation process.

RCSM [42] is an adaptable and context-sensitive middleware that uses software and reconfigurable FPGA (Field Programmable Gate Arrays) hardware to address both customizability and performance for heterogeneous and embedded applications. RCSM proposes the use of biologically inspired cellular-automata-based coordination model to enable diffusion (epidemic) style information dissemination in ubiquitous computing environments. In support of this, RCSM provides situation-aware communication, establishment of device communities, and autonomous coordination for information dissemination.

Lime [15] is a Java-based middleware that provides a coordination layer that can be exploited for designing applications which exhibit either logical or physical mobility, or both. Lime is specifically targeted at the complexities of ad-hoc mobile environments. The goal of Lime is to provide the simple Linda model of coordination in mobile environments via tuple spaces.

Finally, MobiPADS [5] is a reflective middleware that supports active deployment of augmented services for mobile computing. MobiPADS supports dynamic adaptation in order to provide flexible configuration of resources and optimize the operations of mobile applications. MobiPADS configurable services (called mobilets) can be augmented to address the changing conditions of a wireless environment (e.g., CPU load, network bandwidth).
Table 1 shows the comparison of the above middleware technologies with Prism-MW. TAO and Orbix/E support application scalability, security, and delivery guarantees, but do so at the expense of the middleware size. Jini, .NET, XMIDDLE, RCSM, MobiPADS, and LIME all provide support for awareness and mobility, but lack support for data delivery guarantees. MobiPADS provides advanced dynamic reconfigurability facilities. TAO, Orbix/E, .NET, and MobiPADS provide partial support for architectural abstractions in the form of explicit components. However, none of these middleware solutions supports architectural styles, thus clearly distinguishing Prism-MW from them.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Architectural abstractions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Multiple arch. styles</td>
<td>In a single app.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Scarcity</td>
<td>16.6K</td>
<td>55K</td>
<td>55K</td>
<td>156K</td>
<td>95KB</td>
<td>500KB</td>
<td>156KB</td>
<td>95KB</td>
<td>100K</td>
</tr>
<tr>
<td>Efficiency</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Awareness</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mobility</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Reconfig.</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Security</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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

8. 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 [18,24,27,40]. 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.

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

8. Recall from Section 6.2 that an aspect of the existing middleware platforms that hampers their scalability is their support for only one (implicit) software connector.
• 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. We are also in the process of applying Prism-MW in the mobile robotics domain in collaboration with USC’s Center for Robotics and Embedded Systems. 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 the latter 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 an interesting research challenge.

References
15. LIME http://lime.sourceforge.net/
29. Project JXTA. http://www.jxta.org/