A Connector-Aware Middleware for Distributed Deployment and Mobility

Marija Mikic-Rakic and Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{marija,neno}@usc.edu

Abstract

To address the need for highly configurable and customizable distributed systems, researchers and practitioners are investigating various innovative approaches. One of the promising techniques emerging from this area of study is mobile computing. In this paper we present an architecture-based approach to supporting distributed deployment and mobility of software systems. We exploit a connector-aware architectural middleware in providing these capabilities. The approach has been successfully tested on several applications for networks of small, mobile, resource constrained devices (e.g., hand-held computers).

1. Introduction

The software systems of today are rapidly growing in size, complexity, amount of distribution, and numbers of users. We have recently witnessed a rapid increase in the speed and capacity of hardware, a decrease in its cost, the emergence of the Internet as a critical resource, and a proliferation of hand-held consumer electronics devices. In turn, this has resulted in an increased demand for software applications, outpacing our ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them. One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment monitoring, freeway-traffic management, damage surveys in times of natural disaster, and so on. Such scenarios present daunting technical challenges: effective understanding of existing or prospective software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data, numbers of data types, and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute on "small" devices, characterized by highly constrained resources such as limited power, low network bandwidth, slow CPU speed, limited memory, and small display size. We refer to the development of software systems in the described setting as *programming-in-the-small-and-many (Prism)*, in order to distinguish it from the commonly adopted software engineering paradigm of programming-in-the-large (PitL) [4].

As indicated by the above scenarios, of particular interest and importance in this setting is the ability to support *mobility* of hardware, data, and code. This paper focuses on support for *code* mobility at the level of software architectures. Specifically, we exploit an architectural middleware, called Prism-MW [12]. Prism-MW is characterized by event-based communication among autonomous software components that is mediated by first-class, active software connectors. We leverage Prism-MW's explicit connectors, their dynamic nature, and their ability to encapsulate different distribution profiles, to facilitate the mobility of both components and connectors in a given architectural configuration. Our approach natively supports two forms of code mobility: stateless and stateful. *Stateless mobility* (a.k.a. weak mobility [6]) assumes migration of code, but the state associated with that code is not transferred. This capability is usually required for the initial deployment of a distributed system onto a set of target hosts. *Stateful mobility* (a.k.a., strong mobility [6]) assumes the transfer of both the code and state from one host to another. The approach has been evaluated in the context of several applications as well as benchmark tests. In this paper, we focus specifically on the role Prism-MW's connectors play in the approach. The key contribution of this work is two-fold:

1. Support for mobility at the architectural level – this support leverages a highly flexible, efficient, scalable, and extensible architectural middleware; and
2. First-class, explicit software connectors and their higly modular design – the connectors directly facilitate the mobility of both components and connectors.

The rest of the paper is organized as follows. Section 2 overviews Prism-MW's design and implementation. Section 3 describes our support for mobility. The paper concludes with discussions of related and future work.

2. Middleware

In this section we overview the design and implementation of *Prism-MW*, a middleware developed to support the implementation, distributed deployment, and mobility of software architectures in the Prism setting. The middleware provides programming language-level constructs for implementing software architecture-level concepts such as component, connector, configuration, and event. This allows software developers to directly transfer architectural decisions into implementations, thus distinguishing Prism-MW from existing middleware solutions. Another key contribution of Prism-MW is its highly modular design that employs an extensive separation of concerns. This results in a middleware that is flexible, efficient, scalable, and

extensible. Due to space constraints, we only present the relevant subset of the Prism-MW design. Further details may be found in [12].

2.1. Middleware Design

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

Brick is an abstract class that encapsulates common features of its subclasses (*Architecture, Component*, and *Connector*). The *Architecture* class records the configuration of its constituent components and connectors, and provides facilities for their addition, removal, and reconnection, possibly at system runtime. A distributed application is implemented as a set of interacting *Architecture* objects. *Component*s in an architecture communicate by exchanging *Event*s, which are routed by *Connector*s. Prism-MW supports arbitrary routing policies inside each *Connector* instance via the pluggable implementations of the *IHandler* interface. Finally, Prism-MW associates the *IScaffold* interface with every *Brick*. Scaffolds are used to schedule events for delivery and dispatch events using a pool of threads in a decoupled manner. Additionally, dispatching and scheduling are decoupled from the *Architecture*, allowing one to easily compose many sub-architectures (each with its own scheduling and dispatching policies) in a single application. *IScaffold* also directly aids architectural awareness [8] by allowing probing of the runtime behavior of a *Brick*.

To support capabilities that may be required for different (classes of) Prism applications, Prism-MW provides three specialized classes: *ExtensibleComponent*, *ExtensibleConnector*, and *ExtensibleEvent*. These classes extend the corresponding base classes (*Component*, *Connector*, and *Event*, respectively) and, by composing a number of interfaces, provide the ability to select the desired functionality inside each instance of an *Extensible* class. If an interface is installed in a given *Extensible* class instance, that instance will exhibit the behavior realized inside the interface's implementation. Multiple interfaces may be installed in a single *Extensible* class instance. In that case, the instance will exhibit the combined behavior of the installed interfaces. To date, we have provided support for architectural awareness, real-time, distributability, security, heterogeneity, data compression, delivery guarantees, and mobility [1,5,6,7,8,13]. The details of these extensions may be found in [12].

2.2. Foundation for the Mobility Support

Of particular interest to this paper are the extensions we have provided in Prism-MW to support distribution and architectural awareness. These extensions are direct enablers of our support for mobility, discussed in Section 3. In support of distribution, the *ExtensibleConnector* composes the *IDistribution* interface. To date, we have provided two implementations of *IDistribution*, supporting socket-based and infrared-port based inter-process communication. An *ExtensibleConnector* with the instantiated *IDistribution* interface (referred to as *DistributionConnector*) facilitates interaction across process or machine boundaries. Each established connection between two *DistributionConnectors* creates two new *Connection* objects, one inside each *DistributionConnector*. In addition to the *IDistribution* interface inside the *ExtensibleConnector* class, to support distribution and mobility we have implemented the *Serializable* interface inside each one of the *Extensible* classes. This allows us to send data as well as code across machine boundaries.

To support various aspects of architectural awareness, we have provided the *ExtensibleComponent* class, which contains a reference to *IArchitecture*. This allows instances of *ExtensibleComponent* to act as reflexive, meta-level components and to effect run-time changes on the system's architecture. To date, we have augmented *ExtensibleComponent* with several interfaces. Of interest to this paper is the *IAdmin* interface used in support of mobility as further discussed in Section 3. We refer to the *Extensible-*

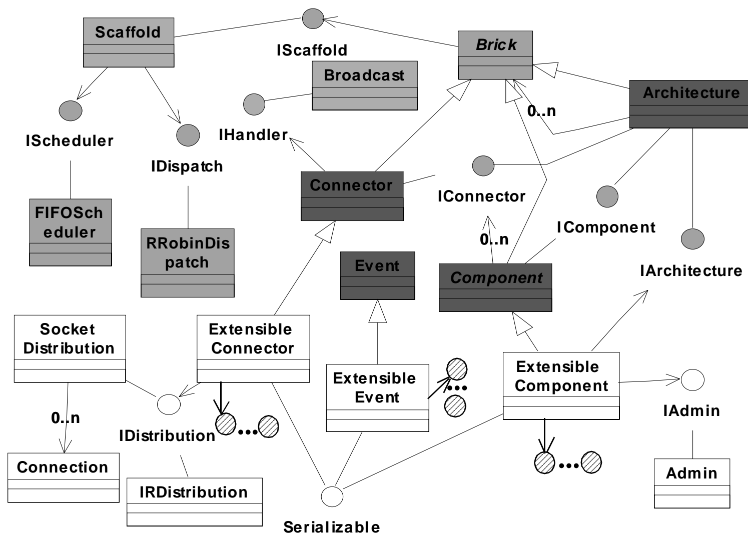
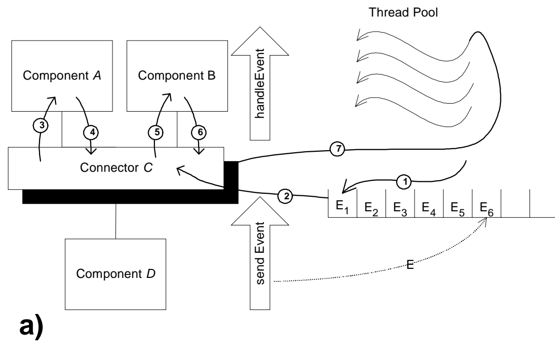


Figure 1. UML class design view of Prism-MW. Middleware core classes are highlighted. Only the relevant extensions are shown.



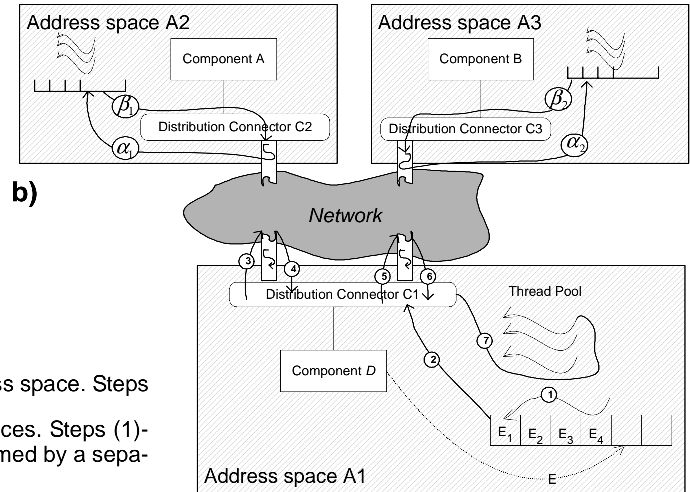


Figure 2. a) Event dispatching in Prism-MW for a single address space. Steps (1)-(7) are performed by a single thread.
b) Event dispatching in Prism-MW for multiple address spaces. Steps (1)-(7) are performed by a single thread. Steps α and β are performed by a separate *DistributionConnector* owned thread.

Component with the instantiated *IAdmin* interface as *AdminComponent* below.

2.3. Using Prism-MW

The first step a developer takes is to subclass from the *Component* class for all components in the architecture and to implement their application-specific methods. The next step is to instantiate the *Architecture* classes for each address space and define the needed instances of thus created components, and of connectors selected from the reusable connector library.[1] Finally, attaching component and connector instances into a configuration is achieved by using the *weld* method of the *Architecture* class. This process can be partially automated using our tool support described in [12].

2.4. Prism-MW Event Processing

In order to maximize the efficiency of Prism-MW, which is highly important for applications in the Prism setting, we have performed several optimizations of Prism-MW's event processing. Prism-MW uses a fixed-size circular array for storing all the generated events to be processed (implemented inside the *FIFOScheduler* class), and a pool of shepherd threads (implemented in the *RRobinDispatch* class) which handles events sent by any component in a given address space. The size of the thread pool is parameterized and, hence, adjustable. To process an event, a shepherd thread removes the event from the head of the queue. For local communication, the shepherd thread is run through the connector attached to the sending component; the connector dispatches the event to relevant components using the same thread (see Figure 2-a). 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.

Prism-MW uses the same basic mechanism for communication that spans address spaces as it does for local communication: a shepherd thread transports an event from the queue to the event's recipients via a *DistributionConnector*. Each *DistributionConnector* owns a number of *Connections*, each of which contains a single thread that listens for incoming events and places them on the local event queue. Therefore, instead of routing the event through the components attached to the connector (steps 3-6 in Figure 2-a), the shepherd thread simply deposits the event on the appropriate communication ports managed by the *DistributionConnector*. As the event is propagated across the network, a *DistributionConnector*-owned *Connection* on each recipient device uses its internal thread to retrieve the incoming event from the communication port and place it on its local event queue (steps α_1 and α_2 in Figure 2-b).

The described design of Prism-MW is highly efficient both in terms of memory consumption and processing speed. We have performed extensive benchmarks to assess Prism-MW's performance [12]. For brevity, we present only the relevant subset of these results here. Memory usage of Prism-MW in a single address space is 4.6 KB. The base size of the *DistributionConnector* is 1.27 KB. In addition to this, each *Connection* adds 2.7 KB on average. Programming language-level support for IPC introduces additional overhead. In Java this overhead is 9.5 KB for loading the `java.net` package. Finally, the AdminComponent overhead is 5KB.

3. Support for Mobility

A distributed architecture in Prism-MW is represented as a configuration of components and connectors deployed onto a set of connected hosts. In such a setting, component migration may be required for various reasons: to minimize the need for remote communication, to increase the local subsystem's autonomy during disconnection, to perform component upgrade, and so forth. For these reasons, many

1. Recall that Prism-MW provides several connectors through the implementations of the *IHandler* interface, and through different combinations of interface implementations inside the *ExtensibleConnector* class.

existing approaches [6] have focused on providing support for component mobility. However, these approaches have not addressed connector mobility, primarily due to the fact that the connectors involved were either implicit (e.g., procedure calls, shared memory), or when explicit, they provided only the basic interaction mechanisms and were readily available on each target host. With a highly connector-aware middleware, such as Prism-MW, the need to migrate connectors that can encompass complex interaction mechanisms becomes very important. These connectors can contain different modules (e.g., security, data conversion, compression) that may not be readily available on each host, and may have internal state (e.g., due to event buffering, monitoring, and statistics gathering) that needs to be migrated. For this reason our approach to mobility supports both component and connector migration in a uniform manner.

The modular design of Prism-MW enables optimization when performing connector moblity: only the connector extensions that maintain state and/or are unavailable on the remote host are migrated; the extensions are then "glued" together by the basic connector facilities provided on each host by Prism-MW. It is also important to note that, while components do not actively participate in the migration process, affected connectors on both source and target hosts need to perform dynamic reconfiguration. The reconfiguration may, in turn, involve addition and removal of connections or entire connectors, as further detailed in Section 3.1.

We directly leverage Prism-MW in supporting mobility involving both *stateless* and *stateful* components and connectors. In particular, our support for these two capabilities directly exploits Prism-MW's explicit software connectors, architectural awareness, and event-based interaction. We have evaluated our approach on a number of applications executing on networks of small, mobile, and resource-constrained devices.

3.1. Role of Connectors

Prism-MW connectors are designed such that they are able to facilitate the event-based interaction between arbitrary numbers of components and connectors. As implemented in Prism-MW, a connector does not contain connection points at declaration-time; as components and connectors are attached to it, the connection points are added to the connector. This property is directly leveraged in enabling the addition, removal, and reconnection of components and connectors in an architecture, possibly at system run-time.

In addition to an arbitrary number of local components and connectors, each *DistributionConnector* can attach to an arbitrary number of remote hosts. As discussed above, each attachment between two *DistributionConnector*s creates a new *Connection* object, and each *Connection* spawns a single thread that listens for incoming events and places them on the local event queue (steps α and β in Figure 2-b). *DistributionConnector*s treat both local components and connectors, as well as remote *Connection*s in the same manner. This capability allows for splicing of a single conceptual connector (shown in the architecture in Figure 2-a) into an arbitrary number of *DistributionConnector*s (shown in Figure 3) in a single step. This solution improves upon the connector-based distribution technique we pioneered [3], where a single connector can only be divided into two connectors in a single step.

Figure 3 shows different distributions of a single architecture (shown in subdiagram a) onto a set of hosts (shown in subdiagrams b, c, d, and e). Let us assume that the initial deployed configuration is the one shown in Figure 3-b, and that we want to migrate component Z from host 2 to host 1 (configuration shown in Figure 3-c). Once Z is migrated (using the techniques described below), the *DistributionConnector*s are reconfigured in the following way: *DistributionConnector* on host 1 adds a local connection to Z while *DistributionConnector* on host 2 removes a local connection to Z. Note, based on the discussion in Section 2.4, that such a migration would not change the number of threads on each host. However, if the original configuration is the one shown in Figure 3-d, and the desired configuration is the one shown in Figure 3-e, the total number of threads may change on each one of the three hosts: *DistributionConnector*s on hosts 1 and 2 could possibly gain one thread each as a result of connection to host 3, while *DistributionConnector* on host 3 would gain two threads. Note that the number of connections in the distributed architecture shown in Figure 3-e depends on the selection of the *IHandler* interface implementation inside each one of the *DistributionConnector*s (recall Section 2.1). For example, if components X and Y did not need to communicated in the configuration shown in Figure 3-d, then there would be no need for the connection between hosts 1 and 3 in the architecture shown in Figure 3-e.

3.2. Stateless Mobility

Prism-MW components communicate by exchanging application events. Prism-MW also allows components to

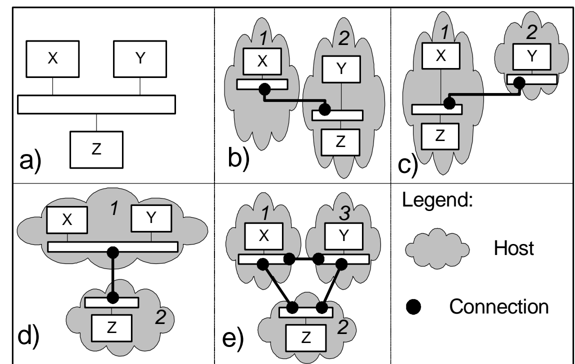


Figure 3. Different distributions of a single architecture on a set of hosts.

exchange *ExtensibleEvent*s, which may contain architectural elements (components, connectors, or connector extensions) as opposed to data. Additionally, *ExtensibleEvent*s implement the *Serializable* interface, thus allowing their dispatching across process boundaries.

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 a *DistributionConnector* and an *AdminComponent* attached to the connector. Note, based on the discussion in Section 2.4, that the Java implementation of the skeleton configuration occupies around 23KB.

As shown in Figure 1, the *ExtensibleComponent* on each device contains a pointer to its *Architecture* object and is thus able to effect run-time changes to its local subsystem's architecture with the help of *DistributionConnector*s: instantiation, addition, removal, connection, and disconnection of components and connectors. *Admin Component*s are able to send and receive from any device to which they are connected the *ExtensibleEvent*s that contain application components, connectors, or connector extensions (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 local *DistributionConnector,* which forwards the event to the attached remote *DistributionConnector*s. Each receiving *DistributionConnector* delivers the event to its attached *AdminComponent*, which reconstitutes functional modules (i.e., components and connectors) from the event, and invokes the *IArchitecture*'s *add* and *weld* methods to insert the modules into the local configuration.

3.3. 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 run-time migration of architectural elements is required (e.g., to minimize remote communication, or to increase the autonomy of the local subsystem during disconnection [9]), 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 (if the element's replication is not desired or allowed). Our approach provides 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 programming language. Instead of sending a set of compiled images, the local *AdminComponent* possibly disconnects and removes the 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 *ExtensibleEvent*s, which are then forwarded by the *DistributionConnector*s. *AdminComponent*s on each target host reconstitute the architectural elements from these events and attach them to the appropriate locations in its local subsystem.

In cases where the serialization-like mechanism is not available (e.g., Java KVM [19]), we provide the event stimulus-based technique: the compiled image of the architectural element(s) to be migrated 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 [16] 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 programming language-independent, and it is *natively* supported in Prism-MW.

4. Related Work

A detailed overview of existing code mobility techniques is given in [6]. Fuggetta et al. describe three code mobility paradigms: remote evaluation, mobile agent, and code-on-demand. *Remote evaluation* allows the proactive shipping of code to a remote host in order to be executed. *Mobile agents* are autonomous objects that carry their state and code, and proactively move across the network. In the *code-on-demand* paradigm, the client owns the resources (e.g., data) needed for the execution of a service, but lacks the functionality needed to perform the service. In this paradigm, the desired component can be retrieved from a remote host, which acts as a code repository, and then executed on the client. As described in Section 3, our work primarily supports the code-on-demand technique.

Existing mobile code systems offer two forms of mobility. *Strong mobility* allows migration of both the code and the state of an execution unit to a different computational environment. *Weak mobility* allows code transfers across different environments; the code may be accompanied by some initialization data, but the execution state is not

migrated. Our approach supports both forms of mobility: strong mobility is supported through the stateful migration technique, while weak mobility is supported by the stateless technique.

5. Conclusions and Future Work

This paper has discussed an approach to code mobility that directly leverages the explicit connectors in an architecture-based middleware. Our approach thus naturally supports mobility at the architectural level and allows the same infrastructure to be used for implementation, initial deployment, execution, run-time evolution, and mobility of application architectures.

Over twenty applications have been implemented, deployed, and migrated using Prism-MW and its native support for mobility. These applications involve traditional desktop platforms, PalmOS- and WindowsCE-compatible devices, digital cameras, and motion sensors. Several of these applications were developed in the context of three graduate-level courses at the University of Southern California (USC). They include distributed digital image capture and processing, map visualization and navigation, location tracking, and instant messaging for hand-held devices. Recently, we conducted a project involving multiple teams of graduate students who made extensive use of Prism-MW's support for mobility to develop capabilities for dynamic service discovery and access. Additionally, we have collaborated with two external organizations, which resulted in a large-scale military application in support of one organization's specific needs in the ground vehicle domain, and the other organization's distributed airborne system. Details on these evaluations can be found in [12].

The results of these evaluations are very promising. However, several areas remain to be explored. We plan to investigate the relationship between our approach and other code mobility techniques, as well as the relationship between Prism-MW and a large body of work on adaptive middleware [2]. For example, an interesting question is whether and to what extent our architecture-based approach is compatible with a number of existing techniques based on tuple spaces (e.g., [15]). A related issue is heterogeneity introduced by applications implemented in multiple programming languages, according to different architectural styles [17], and leveraging different implementation substrates (hardware platforms, operating systems, and/or middleware). Finally, we plan to investigate the use of techniques such as data compression inside *DistributionConnector*s to optimize the use of resources during mobility. In support of these tasks, we plan to explore Prism-MW's extensibility.

6. References .

[1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3), August 2001.

[2] D. Crawford ed. Adaptive Middleware. *Communications of the ACM*. June 2002.

[3] E. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, May 1999.

[4] F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE TSE*, June 1976.

[5] W. Emmerich. Software Engineering and Middleware: A Roadmap. In *The Future of Software Engineering*, ACM Press 2000.

[6] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, May 1998.

[7] E. A. Lee. Embedded Software. In M. Zelkowitz, Ed., *Advances in Computers,* Vol. 56, Academic Press, London, 2002.

[8] C. Mascolo, L. Capra, W. Emmerich. Mobile Computing Middleware. *Advanced Lectures on Networking.* Lecture Notes in Computer Science. Springer.

[9] N. Medvidovic and M. Mikic-Rakic. Architectural Support for Programming-in-the-Many. *TR USC-CSE-2001-506*.

[10] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, vol. 26, no. 1, pages 70-93 (January 2000).

[11] N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 4-11, 2000.

[12] M. Mikic-Rakic and N. Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. TR USC-CSE-2002-510.

[13] P. Oreizy, et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and Their Applications*, 14(3), May/June 1999.

[14] D. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[15] G. P. Picco, A. L. Murphy, G. C. Roman. Developing Mobile Computing Applications with LIME. *ICSE 2000,* Limerick, June 2000.

[16] M. Rakic and N. Medvidovic. Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. 2*001 Symposium on Software Reusability (SSR 2001)*, Toronto, Canada, May 2001.

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

[18] J. P. Sousa, and D. Garlan: Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. *WICSA 2002*, Montreal, Canada, August 2002.

[19] Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm.

[20] C. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 1998

[21] R.N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, June 1996