Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility

Nenad Medvidovic and Marija Rakic

Computer Science Department
Henry Salvatori Computer Center 300
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
+1-213-740-5579
{neno,marija}@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 a software architecture-based approach to supporting code mobility. We exploit the modeling and implementation infrastructure for an architectural style that supports distributed and heterogeneous applications. Our approach leverages the style's highly decoupled components via explicit software connectors in providing the ability to move both data and code across the network in essentially the same manner. The approach has been successfully applied on networks of small, mobile, resource constrained devices (i.e., hand-held computers). We illustrate our approach with the help of an example application.

1 INTRODUCTION

Consider the following scenario, representative of an emerging class of important and increasingly common applications [4,6]. A colony of mobile robots is operating in a remote setting, collaborating to gather data while processing and sending telemetry to keep a central station ("Mission Control") aware of their progress. Occasionally, the robots must adapt their behavior "on-the-fly" because of hardware or software failures, changes in the outside environment, inadequate performance, loss of some of the robots, or addition of new types of robots to the colony. Some changes may be self-initiated, others triggered by Mission Control. Further adaptation may result from development of new data processing algorithms at Mission Control, which are deployed over a communications link to the robots. All resulting adaptations must be accomplished with minimal disruption to the robots' mission.

Similar scenarios can be envisioned for fleets of mobile devices—such as unmanned aerial vehicles—involved in environment and land-use monitoring, freeway-traffic management, fire fighting, airborne cellular telephone relay stations, and damage surveys in times of natural disaster [10,15]. Such scenarios present numerous challenges that current software technologies and development methods are unable to properly address: effective understanding of existing or prospective configurations; rapid composability and dynamic reconfigurability of software; 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 (subsystems implemented in different programming languages, for various platforms, and employing divergent interaction protocols). Furthermore, software reconfiguration often must take place in the face of highly constrained resources, characterized by limited power, low network bandwidth and patchy connectivity, slow CPU speed, and limited memory and persistent storage.

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. In this paper, we propose a software architecture-based solution to the problem of code mobility in such a setting. In particular, we exploit the modeling and implementation infrastructure for an architectural style intended to support highly distributed, heterogeneous applications [17]. The style, C2, is characterized by message-based communication among autonomous components that is mediated by explicit, active software connectors. We leverage the properties of and design discipline imposed by the style to facilitate changing configurations of components and their mobility across a network.

The remainder of the paper is organized as follows. Section 2 provides a brief overview of the C2 style, which serves as the conceptual underpinning for this work. Section 3 introduces our implementation infrastructure. Section 4 discusses our support for mobility that leverages the style and implementation infrastructure. We conclude by comparing our approach to other code mobility techniques and identifying several avenues of future work.

2 THE STYLE

An architecture in the C2 style is modeled as a set of components, connectors, and the topology into which they are composed. C2-style *components* maintain state and perform application-specific computation. They interact with other components by exchanging messages via their two *interfaces* (named *top* and *bottom*). *Connectors* in the C2 style mediate the interaction among components by controlling the transmission and distribution of messages. *Messages* consist of a name and set of typed parameters. A message in the C2 style is either a *request* for a component to perform an operation, or a *notification* that a given component has performed an operation or changed its state. Each component interface consists of a set of messages that may be received and a set of messages that may be sent. Request messages are sent through the top interfaces, while notifications are sent through the bottom interfaces of components and connectors.

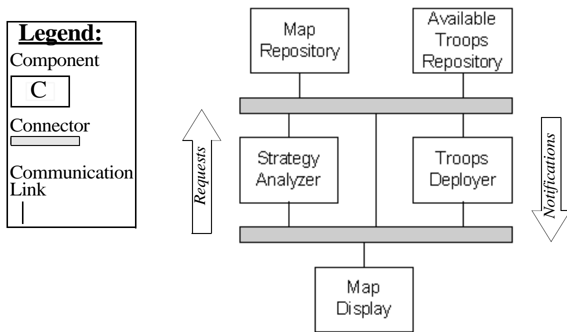

Figure 1. A C2-style architecture for a troop deployment application.

The C2 style mandates that components must always interact via connectors. The top (bottom) of a component may be attached to the bottom (top) of at most one connector; there is no bound on the number of components or connectors that may be attached to a given connector. This decoupling of components allows them to be added to or removed from an architecture with minimal effect on the rest of the system [11]. Figure 1 shows a simple architectural configuration in the C2 style; this figure is further discussed in Section 3.

3 THE IMPLEMENTATION INFRASTRUCTURE

3.1 Overview

C2 provides stylistic guidelines for composing large, distributed systems. For these guidelines to be useful in a development setting, they must be accompanied by support for their implementation. To this end, we have developed a light-weight architecture implementation infrastructure. The infrastructure comprises an extensible framework of implementation-level modules representing the key elements of the style (e.g., architectures, components, connectors, messages) and their characteristics (e.g., a message has a name and a set of parameters). An application architecture is then constructed using this base framework by extending (e.g., subclassing in an object-oriented language) the appropriate classes in the framework with application-specific detail. The framework has been implemented in several programming languages—Java, C++, Ada, and Python—and has been used in the development of over 100 applications to date.

A subset of the C2 framework's class diagram is shown in Figure 2. The classes shown are those of interest to the user of the framework (i.e., the application developer). Both components and connectors may run in a single thread of control (*Component* and *Connector* classes), or they may have their own threads (*ComponentThread* and *ConnectorThread* classes). *Component* and *ComponentThread* classes are abstract; application-level components must be subclassed from them and must provide application-specific functionality. On the other hand, connectors provide application-independent interaction services and may be directly instantiated and used in an application. In addition to the generic *Connector*, we have developed a library of special purpose connectors, such as connectors supporting specific message routing and filtering protocols (e.g., broadcast, multicast, unicast), connectors that encapsulate off-the-shelf, distributed interaction facilities (e.g., middleware), and secure connectors [1,2,9].

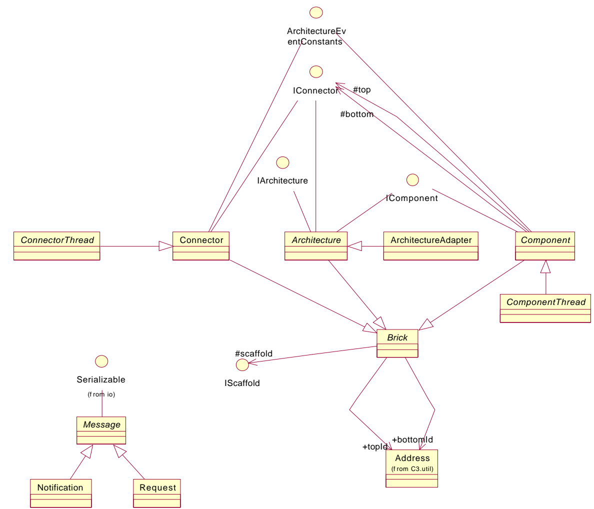

Figure 2. Class design view of the C2 implementation framework.

As discussed in Section 2, a *Request* is a message sent through a component's top port or received through its bottom port, while a *Notification* is a message sent through a component's bottom port or received through its top port. *Architecture* records the configuration of its constituent components and connectors, and provides facilities for their addition, removal, replacement, and reconnection, possibly at system run-time. Finally, *iScaffold* is an interface exported by every *Brick* (component, connector, or entire architecture), which allows probing and monitoring of the run-time behavior of the brick. This interface is used in ensuring system reliability during mobility/adaptation, as discussed in Section 4.

3.2 Example Application

To illustrate the construction of an application using the framework shown in Figure 2, we introduce an architecture for distributed, "on the fly" deployment of personnel, intended to deal with situations such as natural disasters, military crises, and search-and-rescue efforts. The specific instance of this architecture depicted in Figure 1 addresses the deployment of military troops. The two components at the top of the architecture (*Map Repository* and *Available Troops Repository*) model the system's resources—terrain and personnel. The *Strategy Analyzer* component encodes known effective deployment strategies; this component is interchangeable based on the type of mission for which the system is used. The *Troops Deployer* component combines the resources supplied by the two *Repository* components to model the deployment of the available personnel in a particular area. The *Map Display* component provides the front end for the application, updating the current application state in the *Troops Deployer* component based on user input, and issuing requests to *Strategy Analyzer* to assess the chosen deployment. The connectors in the implemented architecture enable broadcast communication between connected components.

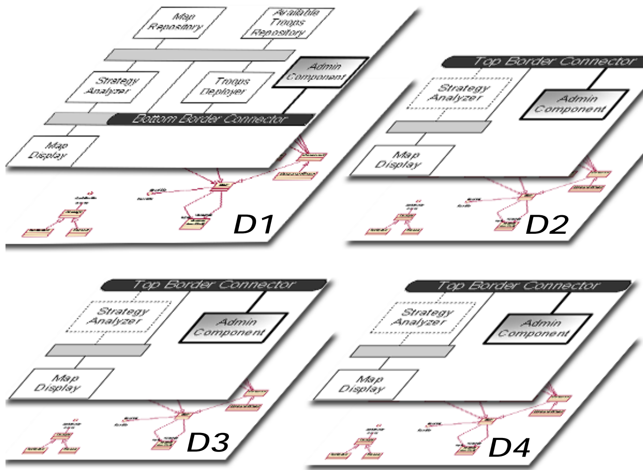

Figure 3. Layered construction of an application using the C2 implementation framework. The application is distributed across four devices. Each device is running our implementation framework.

Figure 3 depicts an implementation configuration of the troop deployment architecture: the application is distributed across four devices, each of which is running our implementation framework (depicted in the bottom planes of the four diagrams) as the local subsystem's execution substrate. One of the devices (e.g., a desktop computer at headquarters, shown in the upper left) contains the full application functionality, while the other three (e.g., palmtop computers deployed in the field) initially contain only instances of *Map Display*.[1]

Each component shown in Figure 3 is a subclass of either the *Component* or *ComponentThread* class provided by the framework. The first step a developer (or tool generating an implementation from an architectural description[2]) takes is to subclass from these base classes for all components in the architecture and to implement the application-specific functionality for each of them. The next step is to instantiate the *Architecture* class and define the needed instances of thus created components, as well as the connectors selected from the connector library. In particular, the four connectors highlighted in Figure 3, called *border connectors*, enable component interaction across the network by implementing one of the techniques described in [1]. Finally, attaching components and connectors into the configuration is achieved by using the *weld* method of the *Architecture* class.

4 SUPPORT FOR MOBILITY

We directly leverage the rules of the C2 style and the implementation infrastructure in supporting mobility of components in an architecture's implementation, during run-time. In particular, our support for mobility directly exploits C2's connectors and message passing.

1. The role of *Admin Component* on each device, as well as that of *Strategy Analyzer* on the three devices deployed in the field (*D2*, *D3*, and *D4*) will be discussed in Section 4.
2. For brevity, we will not discuss the issues of architectural description and generating an implementation from it. An in-depth treatment of these issues is given in [8].

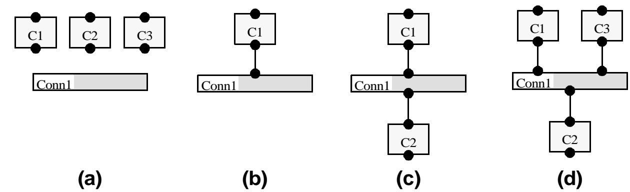

Figure 4. C2 connectors have *context reflective interfaces* and are capable of supporting any number and type of C2 components. *(a)* Software architect selects a set of components and a connector from a design palette. The connector has no interface, since no components are attached to it. *(b-d)* As components are attached to the connector to form an architecture, the connector automatically creates new interfaces to support component intercommunication.

4.1 Evolving Architectural Configurations

C2 connectors have "context reflective" (i.e., polymorphic) interfaces. Connectors are defined such that, in principle, they are able to facilitate the message-based interaction of any (number of) C2 components. As implemented in our frameworks, a connector does not have an interface at declaration-time; as components are attached to it, the connector's top and bottom interfaces are dynamically updated to reflect the interfaces of the components that will communicate through it (modulo any message filters, specified either at system design- or run-time). This property of C2 connectors is depicted conceptually in Figure 4. The property is directly leveraged in enabling the addition, removal, and reconnection of components in an architecture, even at run-time [11]. However, this property is utilized only *after* the components have migrated across the network. The next section discusses our support for the actual migration of components.

4.2 Component Mobility

C2 components typically communicate by exchanging application-level messages (e.g., the *AnalyzeDeployment* request sent from *Map Display* to *Strategy Analyzer* in the example shown in Figures 1 and 3). At the same time, the implementation framework allows components to exchange meta-level messages named *ClassContent*. The *ClassContent* messages directly enable C2 component mobility. Note that each subsystem's architecture in Figure 3 contains a special-purpose *Admin Component*. *Admin Component* contains a pointer to its *Architecture* object and is thus able to effect run-time changes to its local subsystem's architecture. *Admin Component* is used to send and receive the *ClassContent* messages containing functional elements, as opposed to data.

Prior to sending a *ClassContent* message, an *Admin Component* may invoke its *Architecture* object's *unweld* and *remove* methods to, respectively, disconnect and delete the component to be migrated (referred to as *migrant* component below) from the local subsystem's architecture. The *Admin Component* then packages the migrant component into a *ClassContent* C2 message: one parameter in the message is the migrant component itself; another parameter denotes the intended location of the component in the destination subsystem's configuration. The *Admin Component* then sends the message to its local *Border Connector*.

On the receiving end, the *ClassContent* messages are treated differently than application-level data messages: the receiv-

ing *Border Connector* forwards the *ClassContent* messages to is attached *Admin Component*, which unmarshals the messages, reconstitutes functional modules (i.e., components) from them, and invokes its *Architecture* object's *add* and *weld* method to insert the modules into the local configuration using the support discussed in Section 4.1.

We illustrate this approach to component mobility using the troop deployment example. As already discussed in Section 3.2, the troop deployment architecture is distributed across four devices. The interaction of components across the devices is enabled by the border connectors highlighted in Figure 3. If, during the application's execution, a desired component- or system-level property is violated (e.g., as indicated by an application monitoring node inserted via the architecture's *iScaffold* interface), the architecture may decide to reconfigure itself [10]. For example, if the *Strategy Analyzer* component creates a bottleneck because it is located only on device *D1*, *D1*'s *Admin Component* may send copies of *Strategy Analyzer* across the network to be co-located with each subsystem's *Map Display* and to locally perform analyses of proposed troop deployments. This situation is indicated by the grayed-out *Strategy Analyzer* components shown in the *D2*, *D3*, and *D4* subsystems in Figure 3.

In the Java implementation of the framework, this amounts to the following process:[3]

1. If necessary, the migrant component is disconnected from the connectors on its top and bottom sides using the framework's *unweld* method. In our example, since separate copies of *Strategy Analyzer* are being sent, *D1*'s *Admin Component* does not need to disconnect the local *Strategy Analyzer* from the rest of the subsystem.
2. *D1*'s *Admin Component* may unload the migrant component from the local subsystem, or, as is the case in our example, it may access the compiled image of the migrant component from a local file.
3. *D1*'s *Admin Component* serializes the migrant component into a byte stream and sends it as a *ClassContent* C2 message via *D1*'s *Border Connector* to *D2*, *D3*, and *D4*.
4. Once received by the *Border Connector*s on *D2*, *D3*, and *D4*, the *ClassContent* message containing the migrant component is forwarded to the *Admin Component*s running on these devices. Each *Admin Component* reconstitutes the migrant component from the byte stream contained in the message.
5. Finally, each *Admin Component* invokes the *add* and *weld* methods on its *Architecture* object to attach the received migrant component to the appropriate connectors (as specified in the *ClassContent* message) in its local subsystem.

The process described above relies on the existence of Java serialization-like mechanisms. Such mechanisms are not provided by a number of programming languages (e.g., Ada). Furthermore, even Java implementations on certain platforms do not support serialization. We have encountered this latter limitation in our work on enabling component-based development on the Palm Pilot hand-held device, using the Java KVM virtual machine [16]. For this reason, we have considered two additional techniques.

The first technique consists of encoding and migrating components as XML schemas. This technique was suggested previously (e.g., [3]). However, in our work to date we have opted against this approach because of the extreme scarcity of computational resources on the Palm Pilot, which would be only further exacerbated, e.g., by relying on an active XML parser.

The second technique we have adopted as an alternative to Java's serialization directly exploits C2's message passing: the compiled image of the migrant component (e.g., a collection of Java `class` files) is sent across a network as a byte stream packaged in a C2 message. This meta-level message is accompanied by a set of application-level messages needed to bring the state of the migrant component to a desired point in its execution (see [13] for details of how such messages are captured and recorded). Once the migrant component is received at its destination, it is loaded into memory[4] and added to the architecture, but is not attached to the appropriate connectors. Instead, the migrant component is stimulated by the application-level messages sent with it: the local *Admin Component* invokes its local *Architecture* object to issue requests and notifications to the migrant component; in turn, the migrant component is unaware of the fact that these messages are not sent to it via its top and bottom connectors; finally, any messages the migrant component issues in response are not propagated, but are "trapped" by *Admin Component*. Only after the migrant component is brought to the desired state is it welded and enabled to exchange messages 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 our framework.

5 RELATED WORK

With the emergence of the Internet and the resulting highly-distributed software systems, code mobility has become an area of extensive research, resulting in a number of approaches [14]. In this section we provide a brief overview of a handful of techniques related to our work.

A detailed overview of existing code mobility techniques is given in [5]. Fuggetta et al. propose a set of basic architectural concepts and leverage these concepts to describe three design paradigms exploiting code mobility: 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

3. Several implementation-level details of this process are elided for brevity. Also elided are the issues of ensuring application integrity during the dynamic adaptation (see [11]).

4. We should note that the implementation of Java KVM on the Palm Pilot does not support dynamic loading of classes (in this case, migrant components), which forced us to extend KVM with a class loader. It would be necessary to implement similar facilities in languages which do not support dynamic class loading. For example, we are currently adding this support to our framework implemented in Embedded Visual C++ for the COMPAQ Pocket PC hand-held device.

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. Our approach to supporting code mobility also operates at the architectural level. As described in Section 4, 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 computational 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 adoption of the Java serialization technique; weak mobility is supported by the use of Java `class` files as migrant components, and the use of application-level messages to bring a component to its desired state.

Code migration via XML has been exploited in the past [3]. The use of XML allows migration of code at various granularity levels (ranging from individual lines of code to complete programs). Unlike Java programs, which are sent in a compiled form, XML programs are transferred as source code and then interpreted remotely. As discussed in Section 4.2, we have decided against an XML-based approach to code mobility due to the unacceptable overhead it introduces on resource-constrained devices.

6 CONCLUSIONS AND FUTURE WORK

This paper has discussed an approach to code mobility that directly leverages an architecture-based development infrastructure. As such, our approach naturally supports mobility at the level of software components. We have applied the approach to networks of small, mobile, resource-constrained devices. While our results to date are very promising, several areas remain unexplored. Our support for mobility thus far has been restricted to components; we have assumed that we will always be able to leverage the *existing* library of connectors on a given host to service the migrating components. This is an unrealistic assumption in general since different devices may be running different versions of the framework (and thus provide different, possibly outdated connector libraries). Furthermore, as new connectors become available, support for their run-time deployment on remote hosts becomes necessary. Another issue we plan to investigate is the relationship between our approach and other code mobility techniques. 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., [7,12]). A related issue is heterogeneity introduced by applications implemented in multiple programming languages, according to different architectural styles, and leveraging different implementation substrates (hardware platforms, operating systems, and/or middleware). We intend to investigate these issues as part of our ongoing work.

7 ACKNOWLEDGEMENTS

The authors wish to thank N. Mehta, S. Phadke, and A. Rampurwala for their contribution to the design and implementation of the C2 framework and its code mobility support. This material is based upon work supported by the National Science Foundation under Grant No. CCR-9985441. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement number F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

8 REFERENCES

1. E. M. Dashofy, N. Medvidovic, and R. N. Taylor. Using Off-the-Shelf Middleware to Implement Connectors in Distributed Software Architectures. *ICSE'99*, Los Angeles, CA, May 1999.
2. A. Egyed, N. R. Mehta, and N. Medvidovic. Software Connectors and Refinement in Family Architectures. *Third International Workshop on Software Architectures for Product Families*, Las Palmas de Gran Canaria, Spain, Mar 2000.
3. W. Emmerich, C. Mascolo, A. Finkelstein. Implementing Incremental Code Migration with XML. *ICSE 2000*, Limerick, Ireland, Jun 2000.
4. D. Estrin, R. Govindan, and J. Heidemann, eds. Embedding the Internet, *Communications of the ACM*, May 2000.
5. A. Fuggetta, G. P. Picco, G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, May 1998.
6. *IEEE Computer*, Special Issue on Embedded Systems, Sep 2000.
7. I. Marsic. An Architecture for Heterogeneous Groupware Applications. *ICSE 2001*, Toronto, Canada, May 2001.
8. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE'99*, Los Angeles, May 1999.
9. N. R. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *ICSE 2000*, Limerick, Jun 2000.
10. P. Oreizy, et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and Their Applications*, 14(3), May/Jun 1999.
11. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. *ICSE'98,* Apr 1998, Kyoto, Japan.
12. G. P. Picco, A. L. Murphy, G. C. Roman. Developing Mobile Computing Applications with LIME. *ICSE 2000,* Limerick, Jun 2000.
13. 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.
14. G. C. Roman, G. P. Picco, A. L. Murphy. Software Engineering for Mobility: A Roadmap. In *Future of Software Engineering*, ACM Press 2000.
15. G. S. Sukhatme and J. F. Montgomery. Heterogeneous Robot Group Control and Applications. *AUVS 99 Conference,* 1999.
16. Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm
17. R. N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, 22(6), Jun 1996.