Architectural Support for Programming-in-the-Many

Nenad Medvidovic  Marija Mikic-Rakic  Mahindra B. Pai

Henry Salvatori Computer Center 338
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{neno, marija, mpai}@usc.edu

ABSTRACT
Over the past several decades software researchers and practitioners have proposed various approaches, techniques, and tools for developing large-scale software systems. The results of these efforts have been characterized as *programming-in-the-large* (*PitL*). A new set of challenges has arisen with the emergence of inexpensive, small, heterogeneous, resource-constrained, possibly embedded, highly-distributed, and highly-mobile computing platforms. We refer to software development in this new setting as *programming-in-the-many* (*Pitm*). This paper presents an approach intended to address the challenges of PitM. The centerpiece of our approach is a software architectural style with explicit support for the needs of PitM applications: self-awareness, distribution, heterogeneity, dynamism, mobility, and disconnected operation. The style is accompanied by a set of implementation, deployment, and runtime evolution tools targeted to a variety of traditional (i.e., desktop) and mobile computing platforms. Our approach has been successfully applied on a number of applications. While several issues pertaining to PitM remain areas of future work, our experience to date has been very positive.

1 INTRODUCTION
The software systems of today are rapidly growing in size, complexity, amount of distribution, heterogeneity of constituent building blocks (*components*), 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 worldwide resource, and a proliferation of hand-held consumer electronics devices (e.g., mobile phones, personal digital assistants). 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.

Consider the following scenario, representative of the above picture. 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—both manned and unmanned—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 [29,39]. Such scenarios present daunting challenges: effective understanding of existing or prospective 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 (subsystems implemented in different programming languages, for various platforms, and employing divergent interaction protocols). Furthermore, software often must execute 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.

These challenges paint a picture that traditional software technologies and development methods are unable to properly address. The traditional technologies and methods are primarily geared toward supporting development and evolution of large-scale software systems, but whose degrees of distribution, heterogeneity, dynamism, mobility, and resource constraints are substantially lower than demanded by the scenarios outlined above. The set of problems addressed by the traditional software development approaches has been characterized as *programming-in-the-large* (*PitL*) [8]. We believe that this new set of challenges can be more appropriately characterized as *programming-in-the-many* (*Pitm*): software development support for highly distributed, dynamic, mobile, heterogeneous, resource-constrained computation. This paper presents an approach to PitM whose goal is to address a number of the above challenges simultaneously.

It should be noted that, even though traditional software development approaches are unable to adequately support PitM, researchers have recently begun to attack its various aspects, including:

- resource consumption analysis [10] and optimized compilation [1], to streamline an application’s use of constrained resources;
- software deployment [16], to aid developers in properly configuring distributed applications;
- dynamic reconfiguration [29] and code mobility [13] to aid application evolution “on the fly” and thus minimize application downtime; and
- disconnected operation [18,20], to maximize application availability in the face of connectivity losses, and software and hardware failures.

Varying degrees of progress have been made in achieving support in these areas. At the same time, all of the above approaches
exhibit one, or both, of the following two shortcomings: some of them (e.g., dynamic reconfiguration, deployment) have not been tailored to or applied in this novel, heterogeneous, highly mobile, resource constrained setting, while others (e.g., code mobility, disconnected operation) have not been accompanied by appropriate application design methodologies.

In this paper we propose an approach to PitM that addresses both of these issues. Our work interweaves a number of ideas that we had explored in the past with several new ideas. We have adopted and, where necessary, adapted the results of the techniques listed above. In the process of investigating the characteristics of PitM, we have also developed several novel solutions to the PitM challenges. In particular, our approach supports PitM by explicitly leveraging the concepts and constructs of software architectures [30,38]. Our work has six key aspects:

- it leverages explicit design idioms comprising an architectural style;
- it promotes architectural self-awareness via an explicit meta-architecture, which enables continuous architectural monitoring, analysis, and evolution;
- it provides a light-weight, tailorable architecture implementation, deployment, and reconfiguration infrastructure;
- it leverages explicit, first-class software connectors, which are central to both the architectural style and the implementation infrastructure;
- it natively supports distributed deployment, mobility, and dynamic reconfiguration of applications; and
- it supports a risk-based approach to disconnected operation.

To date, these ideas have been applied on a number of applications executing on a variety of desktop and mobile computing platforms. While we believe that several of the aspects of our approach are by themselves important contributions of this work (e.g., architectural style for PitM, architectural self-awareness, risk-based support for disconnected operation), the true benefit of our work is their combination, which affords us unique insights into and a solid foundation for further studying PitM.

The remainder of the paper is organized as follows. Section 2 discusses the nature of the problem we are addressing and outlines our approach. Section 3 briefly describes an example application depicted in Figure 1 addresses military Troops Deployment and battle Simulations (TDS). A computer at Head
quarters gathers all information from the field and displays the complete current battlefield status: the locations of friendly and enemy troops, as well as obstacles such as mine fields. The Headquarters computer is networked via a secure link to a set of handheld devices used by officers in the field. The configuration in Figure 1 shows three Commanders and a General; two Commanders use Palm Pilot Vx devices, while the third uses a Compaq iPAQ; the General uses a Palm Pilot VIIx. The Commanders are capable of viewing their own quadrant of the battlefield and deploying friendly troops within that quadrant to counter enemy deployment. The General sees a summarized view of the entire battlefield; additionally, the General is capable of seeing the detailed views of each quadrant. Based on the global battlefield situation, the General can issue direct troop deployment orders to individual Commanders or request transfers of troops among the Commanders. General can also request for deployment strategy suggestions from Headquarters, based on current positions of enemy troops, mines, and the number of friendly troops at disposal. All deployments are reported to Headquarters, which is able to analyze the deployment strategy. Finally, the General can issue a “fight” command, resulting in a battle simulation that incrementally determines the likely winner given a configuration of troops and obstacles.

4 AN ARCHITECTURAL STYLE FOR PITM

We have developed an architectural style, which is capable of effectively capturing the characteristics of application architectures found in the PitM setting. The style is intended to address the key characteristics of PitM discussed in Section 1: architectural self-awareness, distribution, dynamism, mobility, and disconnected operation.

In formulating the PitM style, we have leveraged our previous experience with the C2 architectural style, which is intended to support highly distributed applications [40]. 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. The components may not assume a shared address space, but instead interact with other components solely by exchanging messages via their two communication ports (named top and bottom). Connectors in the C2 style mediate the interaction among components by controlling the distribution of all 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. Request messages are sent through the top ports, while notifications are sent through the bottom ports of components and connectors. The distinction between requests and notifications ensures C2’s principle of substrate independence, which mandates that a component in an architecture may have no knowledge of or dependencies on components below it.

Several characteristics of C2 (highly distributed architectures, autonomous components communicating through explicit connectors, substrate independence, and dynamism) are a good fit for the needs of PitM. However, support for other aspects of PitM (deployment, mobility, and disconnected operation) has to be built on top of C2’s existing facilities. Finally, certain aspects of PitM simply cannot be supported by C2. C2 mandates that components engage in asynchronous interactions only; this aspect of the style makes it ill suited for certain classes of applications (e.g., applications with real-time requirements). Furthermore, C2 imposes a strictly vertical topological orientation on architectures. Coupled with the semantic distinction between notification and request messages, this orientation is suited for a client-server style of interaction, but not for peer-to-peer interaction, which becomes critical as PitM applications become more widely distributed and decentralized.

For these reasons, we have chosen to use C2 as the basis of the PitM style, with three major enhancements to account for the above shortcomings. Two additional enhancements that form the foundation of our approach—support for deployment/mobility and disconnected operation—are based on these and are discussed later in the paper.

Peer-to-Peer Interaction. While we still allow the C2-style vertical topology in PitM architectures and communication via requests and notifications, we introduce a third component port (called side) and message category (called peer). Side ports allow us to address the relative topological rigidity of C2. They have proven particularly effective in component interactions across devices on a network (e.g., C_AvailableTroops and G_AvailableTroops components on Palm-I and Palm-3 devices in Figure 2). In order to maintain complete component decoupling, the side ports exchange peer messages through “vertical” connectors (e.g., SideBorderConnectors in Figure 2). Vertical connectors serve a purpose analogous to C2’s “horizontal” connectors: they...
control the distribution of peer messages. Two PitM components may not engage in interaction via peer messages if there exists a vertical topological relationship between them. Allowing such interaction would violate the principle of substrate independence inherited from C2. For example, DataRepository on the PC and G_ScenarioEditor on the Palm-1 in Figure 2 may not exchange peer messages since one component is above the other; on the other hand, no vertical topological relationship exists between C_AvailableTroops and G_AvailableTroops. Finally, the PitM style disallows the possibility of exchanging messages between a vertical and a horizontal connector (which would, in effect, convert peer messages into requests/notifications, and vice versa).

**Architectural Self-Awareness.** PitM supports architectures at two levels: application-level and meta-level. Additionally, PitM supports three types of messages: ApplicationData (similarly to C2), ComponentContent, and ArchitecturalModel. The role of components at the PitM meta-level is to observe and/or facilitate different aspects of the application-level components’ execution, dynamic evolution, mobility, and disconnected operation. For example, we have extensively used special-purpose components, referred to as Admin Components, whose task it is to exchange ComponentContent messages and facilitate the deployment and migration of application components across devices (see Section 7). Another meta-level component is the Continuous Analysis component, which leverages ArchitecturalModel messages for analyzing the (partial) architectural models during the application’s execution, assessing the validity of proposed runtime architectural changes, and possibly disallowing the changes. In support of this task, we are currently reusing the architecture modeling and analysis techniques developed for C2: the Continuous Analysis component is extracted from our DRADEL environment [27].

Meta-level components may be application independent (e.g., the Admin Components and Continuous Analysis components), or application specific. An example of an application specific meta-level component is a component monitoring the frequency of AnalyzeStrategy requests issued from a Palm to the PC in the TDS architecture in Figure 2, and the latency of responses to those requests. If either measure is above a pre-specified threshold, the meta-level component requests that the application-level StrategyAnalyzer component be migrated to the Palm (see Section 7).

**Border Connectors.** The third significant departure from C2 in formulating the PitM style is the key role of connectors (both horizontal and vertical) that span device boundaries. Such connectors, called border connectors, enable the interactions of components residing on one device with components on other devices. The high degrees of distribution and mobility, as well as the high probability of disconnected operation in PitM architectures have caused us to place special importance upon border connectors. A single border connector may service network links to multiple devices (e.g., BottomBorderConnector on the PC in Figure 2). A border connector marshals and unmarshals data, code, and system models; dispatches and receives messages across the network; and monitors the network links for disconnection. It may also perform data compression for efficiency and encryption for security.

**Example Application in the PitM Style.** Figure 2 shows the architectural configuration of the TDS application in the PitM style. The architecture is distributed across five devices as depicted in Figure 1. The subarchitecture on the PC device contains a model of the system’s overall resources—terrain, personnel, as well as the current and standard deployment strategies. The StrategyAnalyzer, DeploymentAdvisor, and WarManager 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. The subarchitecture on the Palm-1 device provides the General’s functionality, while Palm-2, Palm-3, and iPAQ provide the three Commanders’ functionalities. The G_AvailableTroops component in General’s subarchitecture is able to make direct orders (by sending peer messages through vertical border connectors) to the Commanders’ C_AvailableTroops components to reposition troops across battle-field quadrants.  

## 5 IMPLEMENTATION, DEPLOYMENT, AND RECONFIGURATION INFRASTRUCTURE

PitM 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 lightweight 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 JVM and KVM, C++ and Embedded Visual C++ (EVC++), and Python. The framework’s application programming interface (API) has been designed such that it is consistent with the C2 implementation framework [25]: applications implemented using the C2 framework can execute without any modifications using the PitM framework. Note that PitM applications cannot execute using C2’s framework because that framework does not support peer messages, side component ports, or vertical connectors. Furthermore, C2’s framework does not support PitM’s meta-architecture.

A subset of the PitM framework’s UML class diagram is shown in Figure 3. The classes shown are those of interest to the user of the framework (i.e., the application developer). Multiple components and connectors in an architecture 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; a meta-level or application-level component must be subclassed from them and must provide the component’s 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, further discussed in Section 6. The Architecture class records the configuration of its constituent components and connectors, and provides meta-level facilities for their addition, removal, replacement, and reconnection, possibly at system runtime. A distributed application, such as TDS, is implemented as a set of (interacting) Architecture objects. Finally, iScaffold is an interface

---

1. We are currently extending DRADEL to ensure the validity of peer interactions, in addition to the notification/request interactions.

2. The peer-to-peer links to the Palm-2 and iPAQ are elided for clarity, as are the meta-level components. The Admin Component meta-level component is shown in Figure 4.
exported by every Brick (component, connector, or entire architecture), which directly supports PitM’s meta-level and aids architectural self-awareness by allowing probes and monitors of the runtime behavior of the brick, further discussed in Section 7.

Since it is intended to support applications executing on resource-constrained platforms, we have built several optimizations into the PitM framework. These include the manner in which components, connectors, and messages are stored and processed. For example, the C2 framework’s explicit communication port objects that maintain their own private message queues have been replaced in the PitM framework with a central FIFO message queue per each address space (i.e., Architecture object). An adjustable pool of shepherd threads is kept ready to handle messages sent by any component in a given address space. For communication that spans multiple address spaces (and possibly machine boundaries), messages are transported via border connectors (recall Section 4) and added to the message queues in the recipients’ address spaces. The control over this thread pool as well as the message queue is exercised from PitM’s meta-architecture components, via (a pointer to) the Architecture object.

The PitM style and framework implementations have been used in the development of over a dozen applications involving hand-held, resource constrained devices: Palm Pilot and Compaq iPAQ. The light weight of the framework is reflected in the size of its source code (1500 SLOC) and memory usage (under 20KB at system start-up time). In a simple application, consisting of two components communicating through a connector in a single address space, it takes under 15 seconds for the components to exchange one million messages using a single thread of control.

Figure 4 depicts an implementation configuration of the TDS architecture: as depicted in Figures 1 and 2, the application is distributed across five devices. Each device is running the PitM implementation framework (depicted in the bottom planes of the five diagrams) as the local subsystem’s execution substrate. The three Palms run the Java KVM version of the framework, the iPAQ runs the EVC++ version, and the PC runs the Java JVM and C++ versions of the framework. We leverage XML-based components, described in Section 6, to enable interaction of application components across the language boundaries.

The first step a developer (or tool generating an implementation from an architectural description) takes is to subclass from the Component or ComponentThread framework 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 classes for each device and define the needed instances of thus created components, as well as the connectors selected from the connector library (see Section 6). Finally, attaching components and connectors into a configuration is achieved by using the weld and peerWeld methods of the Architecture class. At any point, the developer may add meta-level components, which may be welded to specific connectors and thus exercise control over a particular portion of the Architecture (e.g., Admin Component in Figure 4) or they may remain unwelded and instead exercise control over the entire Architecture object directly (e.g., Continuous Analysis component, not shown in Figure 4). This process is described in more detail in Section 7.

6 FIRST-CLASS SOFTWARE CONNECTORS

Similarly to its C2 ancestor, central to both the PitM architectural style and its implementation infrastructure are explicit software connectors [28]. Connectors are defined such that, in principle, they are able to facilitate the message-based interaction of any (number of) components. As implemented in the PitM framework, a connector does not have an interface at declaration-time; instead, as components are attached to it, the connector’s interface is dynamically updated to reflect the interfaces of the components that will communicate through the connector. This “polymorphic” property of connectors is directly leveraged in enabling the runtime addition, removal, and reconnection of components in an architecture. This property is thus the key enabler of our support for mobility and disconnected operation, as discussed below.

To date, we have reused a number of C2 connectors, but have also developed several additional connectors that are necessitated by the nature of PitM applications. Below, we highlight several representative PitM connectors.

3. Several of the applications were developed as part of a graduate-level software architecture course at USC. These applications included distributed digital image processing, map visualization and navigation, collaborative spell checking, and instant messaging for hand-held devices.

4. The measure reflects the Java version of the framework. The benchmark was performed on an Intel Pentium II 300MHz processor with 128 MB of RAM running Sun Java JDK™ 1.2.

5. 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 [27].
Message Broadcast, Multicast, and Unicast Connectors. C2 connectors support only asynchronous message broadcast. PitM connectors also support synchronous message broadcast. Additionally, PitM connectors support both asynchronous and synchronous message multicast and unicast, which are particularly useful in peer-to-peer communication.

Inter-Process Connectors. We have reused the technique for component interactions across process and machine boundaries developed for C2 connectors [7]. The technique has been adapted to support both horizontal and vertical border connectors in PitM. A particularly interesting kind of border connector are infra-red (IR) connectors, which leverage the commonly present IR communication ports on hand-held devices and enable short-range, wireless interaction of PitM components across those devices.

XML-Based Connectors. In order to communicate across heterogeneous platforms, runtime environments, and programming languages, we have developed PitM connectors that use XML as a medium for message passing. XML-based connectors encode messages as XML strings before sending them across the network. Once received, the contents of such messages (e.g., data types of message parameters) are interpreted in the context of each receiving device’s computing environment.

Secure Connectors. Security is of paramount importance in a highly distributed, dynamic, mobile setting. For this reason, we have developed an encryption module that may be added to any border connector. For example, we have used this module with the IR border connectors to encrypt and marshal information during communication between secure parts of an architecture. Our approach uses a symmetric encryption algorithm [3]. In a symmetric algorithm, the same keys are used for both encryption and decryption, but are frequently changed. Symmetric algorithms are very fast and, thus, are preferred when encrypting large amounts of data or when used on devices with limited computing power (e.g., the Palm Pilot).

Multi-Versioning Connectors. The nature of PitM applications requires support for upgrading their functionality at runtime and in a reliable manner. To this end, we have developed multi-versioning connectors (MVCs) [32]. An MVC allows multiple component versions to execute in parallel, such that their presence does not affect the application’s functionality. MVCs collect the execution statistics (relative correctness, reliability, and performance) of the different versions. MVCs can also monitor the volume of incoming and outgoing message traffic for each component version. This can provide additional information about which version to retain in the system. For example, if the multi-versioned component is likely to be migrated to another device, its most desirable version may be the one with the lowest dependency factor (see Section 8).

7 APPLICATION DEPLOYMENT, MOBILITY, AND DYNAMIC RECONFIGURATION

Deployment
Our support for deployment, mobility, and dynamic reconfiguration directly leverages the PitM implementation infrastructure described in Section 5. In order to deploy the desired configuration on a set of target hosts, we assume that a skeleton (meta-level) configuration is preloaded on each host. The skeleton configuration consists of an Architecture object that contains a Border Connector and an Admin Component attached to the connector. Border Connectors enable inter-device communication, while Admin Components facilitate (sub)architectural self-awareness (recall Sections 4 and 5). Each subsystem’s Admin Component contains a pointer to its Architecture object and is thus able to effect runtime changes (i.e., instantiation, addition, removal, connection, and disconnection of components and connectors) to its local subsystem’s architecture. Additionally, Admin Components are able to send the meta-level Configuration messages through Border Connectors to any device to which they are connected.

We have integrated and extended the COTS Microsoft Visio tool to provide Prism, the PitM architectural modeling and deployment environment (see Figure 2). Prism contains several toolboxes (shown on the left side of Figure 2). The top toolbox enables an architect to specify a configuration of hardware devices by dragging their icons onto the canvas and connecting them. The remaining toolboxes supply the software components and connectors that may be placed atop the hardware device icons. Once a desired software configuration is created in Prism, it can be deployed onto the depicted hardware configuration with a simple button click. We currently assume that the locations of the compiled code for all the needed components and connectors are known, and specified inside Prism. Additionally, Prism currently assumes that the network address of each device is known; in the future, we plan to extend prism Prism with support for automated discovery of network nodes.

Prism creates a description of the configuration (shown in Figure 5) and directly invokes the skeleton configuration on its local device. The skeleton configuration’s Admin Component waits for each device specified in the hardware configuration to connect, reads the description generated by Prism, and sends appropriate messages to Admin Components residing on connected devices. Each Admin Component receives a set of compiled code locations for components and connectors (the source parameter of the add command in Figure 5), and information about where in the architecture’s topology the components and connectors should be placed (weld and peerWeld commands). If the desired architectural element cannot be directly instantiated from the framework and it is not available locally, a local Admin Component requests the element’s compiled code from Admin Component on the device that contains it. The code is streamed and sent to the requesting Admin Component using the PitM Configuration message. The requesting Admin Component instantiates the component or connector, and invokes its Architecture object’s add, weld or peerWeld methods to insert it into the local configuration.

Mobility and Dynamic Reconfiguration
We use this same technique for supporting runtime component mobility: Admin Components exchanging Configuration messages containing mobile code. We illustrate the technique using the TDS application. As already discussed in Section 4, the
TDS architecture is distributed across five devices and the interaction of components across the devices is enabled by Border Connectors. If, during the application’s execution, a desired component- or system-level property is violated (e.g., as indicated by a meta-level monitoring node inserted via the Architecture’s Scaffold interface), the architecture may decide to reconfigure itself. For example, if the Strategy Analyzer component creates a bottleneck because it is located only on the PC, PC’s Admin Component may send copies of Strategy Analyzer across the network to be co-located with each subsystem’s Scenario Editor and to locally perform analyses of proposed troop deployments.

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

1. If necessary, the migrant component is disconnected from its attached connectors using the framework’s unweld and peerUnweld method. In our example, since separate copies of Strategy Analyzer are being sent, PC’s Admin Component does not need to disconnect the local Strategy Analyzer from the rest of the subsystem.
2. PC’s Admin Component may unload the migrant component from the local subsystem using the framework’s remove method or, as is the case in our example, it may access the compiled image of the migrant component from a local file.
3. PC’s Admin Component serializes the migrant component into a byte stream and sends it as a ComponentContent message via PC’s Border Connector to the Palm-1, Palm-2, Palm-3, and iPQ devices.
4. Once received by the Border Connectors on the four destination devices, the ComponentContent message is forwarded to the Admin Component running on each device. Each Admin Component reconstitutes the migrant component from the byte stream contained in the message.
5. Finally, each Admin Component invokes the add, weld, and peerWeld methods on its Architecture object to attach the received migrant component to the appropriate connectors (as specified in the ComponentContent message) in its local subsystem.

The process described above relies on the existence of Java serialization-like mechanisms. Such mechanisms are not provided by all programming languages. Furthermore, even Java implementations on certain platforms do not support serialization. We have encountered this latter limitation in Java KVM for the Palm Pilot [41]. For this reason, we have considered several additional mobility techniques [26], adopting one that directly exploits PitM’s message passing: the compiled image of the migrant component [26], adopting one that directly exploits [41]. For this reason, we have considered several additional mobility techniques [26], adopting one that directly exploits PitM’s message passing: the compiled image of the migrant component (e.g., a collection of Java .class files) is sent across the network as a byte stream packaged in a ComponentContent message. This meta-level message is accompanied by a set of ApplicationData messages needed to bring the state of the migrant component to a desired point in its execution (see [32] for details of how such messages are captured and recorded). Once the migrant component is received at its destination, it is loaded into memory7 and added to the Architecture object by the local Admin component, but is not attached to the appropriate connectors. Instead, the migrant component is stimulated by the ApplicationData messages sent with it: the Admin Component invokes the

Architecture object which in turn spawns a thread used to issue request, notification, and peer messages to the migrant component; the migrant component is unaware of the fact that these messages are not sent to it via its attached connectors; finally, any messages the migrant component issues in response are not propagated, but are “trapped” by the Architecture. Only after the migrant component is brought to the desired state is it welded and enabled to exchange messages with the rest of the local 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.

8 SUPPORT FOR DISCONNECTED OPERATION

Due to the nature of mobile devices, their connections are intermittent, with periods of disconnection. A goal of our work on PitM has been to minimize the risks associated with disconnection by maximizing the availability of an application during disconnection. Our approach to disconnected operation proposes migrating components from neighboring hosts to a local host before the disconnection occurs. The set of components to be migrated is chosen such that it maximizes the autonomy of the local subsystem during disconnection, while staying within the memory constraints posed by the device and within the time needed for migrating the components.

We identify the following factors to be pertinent in deciding which components should be migrated:

- statefulness of candidate components for migration,
- frequencies of messages present on the network link to be disconnected,
- dependency of candidate components,
- type of disconnection,
- required memory for loading candidate components [2],
- available memory on the target device,
- bandwidth of the network link to be disconnected.

Several of these factors (i.e., the ones that are not self-evident) are described below.

Statefulness. Statefulness (S) of a component describes the degree of influence the component’s state has on the outcome of its operations. Statefulness can be calculated using the following formula:

\[ S = \frac{SD}{TO} \cdot S \in [0,1] \]

where SD is the number of operations whose outcome depends on the component’s state, and TO represents the total number of operations the component exports. We refer to components whose S value is low as stateless and those whose S value is high as stateful. An example of a stateless component is a math library (a set of functions that calculate a result given a set of inputs). A simple example of a stateful component is a stack: its state comprises a set of elements that are currently on the stack; the result of a stack operation, e.g., Pop, is directly influenced by the stack’s contents.

Stateless components may be replicated onto the target host without synchronization problems when the connection is restored. However, the replication of a stateful component requires the ability to synchronize, after the connection has been restored, any updates made to the different replicas of that component during disconnection.

In order to simplify this task, we allow some operations to be fully accessible (allowed); other operations are delayed until the connection is restored; access to yet other operations is disallowed. For example, in the TDS application operations such as AnalyzeStrategy are allowed to be executed locally during disconnection. We defer operations such as the Deploy command issued from the General to the Commanders. Finally, we disallow operations such as Fight until the connection is restored (the positions

6. 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 [29]).
7. Java KVM does not support dynamic loading of classes, which forced us to extend KVM with a third-party class loader. We have also added this support to our C++ and EVC++ frameworks using DLLs.
of enemy troops might change significantly during disconnection. We have extended our architecture modeling support [27] with a degraded mode tag for each provided service, indicating whether the service should be allowed, disabled, or monitored during disconnection. The corresponding support in the PitM implementation framework is ensured via runtime flags for disabled operations and a separate message queue corresponding to delayed operations. These messages, which will ultimately require remote processing, will be forwarded to their destination by the local Border Connector once the connection is restored.

**Message Frequency.** We assume that the frequencies of messages (f_i) present on the network link that is going to be disconnected can be observed over a period of time, such that we can calculate the frequency of each message going across the link. We have extended our connectors with the capability to monitor message frequencies. The objective of our approach is to minimize the message traffic going through the “broken” link(s) during the period of disconnection.

**Dependency.** We define two types of operations that a component may export: dependent and independent. Independent operations execute without invoking other components in the system, while dependent operations need to invoke the operations of one or more components in the system in order to complete their task. We have extended our architecture modeling support [27] with a dependency tag, associated with each provided service of a component. Dependency of an operation (d_i) is defined as the number of external services needed for the completion of that operation. Dependency (D) of an entire component can now be defined as follows:

\[ D = \frac{\sum d_i \times f_i}{N} \]

where d_i is the dependency and f_i the frequency of i-th operation’s invocations, while N is the total number of operations a component exports.

Dependency of an entire component determines whether its migration is going to be useful. Migration of independent components is likely to reduce the message traffic present on the network link, while migration of dependent components may in fact increase the message traffic along the link.

**Type of Disconnection.** We identify two types of disconnection: anticipated and sudden. In cases of anticipated disconnection the user is aware that disconnection is going to occur, and usually can predict when it will happen (e.g., loss of mobile phone signal when driving into a tunnel). In cases of sudden disconnection, the user is unaware of the disconnection beforehand. Our approach supports both types of disconnection provided that certain assumptions hold: in the case of sudden disconnection we assume that the probability of disconnection is known; in the case of anticipated disconnection, we assume that the time remaining until disconnection is known.

Our risk-based approach addresses both types of disconnection. In case the probability of sudden disconnection is high, we propose prefetching of components that are going to be needed during disconnection, and instantiating them when the disconnection occurs. In case of anticipated disconnection we propose migrating components before the disconnection occurs. In order to select the best component set for migration, for each candidate component we need to know (1) the benefit of migration, expressed as the increase in the application’s availability on the local device if the component is migrated and (2) the required memory for loading the component.

For each candidate component the benefit can be estimated using the following procedure:

1. **Benefit** ← 0
2. Compare the static description of the candidate component with the list of messages exchanged across the link. For all operations invoked on a candidate component as a result of these messages do the following:

\[ \text{Benefit} \leftarrow \text{Benefit} + f_i \times (1 - d_i) \]

where f_i is the frequency of the message and d_i is the dependency factor of the corresponding operation. This formula states that component benefit may increase only in the case of independent operations; the benefit remains unchanged for operations that depend on exactly one external operation; it decreases in all other cases.

Total available memory (TAM) for loading components on a device is calculated using the following formula:

\[ \text{TAM} = \min(M, t \times nb) \]

where M is the actual available memory on the device (in KB), t is time remaining before disconnection (in seconds), and nb is network link’s bandwidth (in KB/second).

The benefit of a set of components is expressed as the percentage increase of application’s availability on a given host if this set of components is migrated. In order to select the best set of components for migration we would have to construct a graph whose nodes are components (with associated required memory) and edges component interdependencies, with edge weights indicating the frequencies of exchanged messages. Such a graph can be constructed using a model of the architecture (to obtain the nodes and the edges of the graph), and runtime behavior of the system (to obtain the message frequencies). Once the graph is constructed, in its simplest form the problem becomes one of dividing the graph into two subgraphs, such that the sum of edge weights between the two subgraphs is minimized (corresponding to minimized communication between two hosts). Additionally, the total required memory for all components in each subgraph has to be less than the total available memory on the corresponding device. This problem is known in the literature as the minimum k-cut problem [6], with memory as an additional constraint. The problem is NP hard and the resulting algorithm runs in exponential time. This solution is computationally too expensive if the number of candidate components is high, and/or number of hosts greater than two. We propose a simplification of the problem that becomes solvable in polynomial time in the number of components. The simplified problem can be stated as follows: We have a set of n candidate components for migration. Given the benefit and required memory for each component, select a set of components that maximizes the total benefit TB (as the sum of benefits of individual components) if the total available memory is TAM. This problem is a variant of the well studied 0-1-knapsack problem, and can be solved using dynamic programming techniques [5]. The algorithm runs in O(n×TAM) where n is the number of components. The algorithm assumes that the benefits of individual components are mutually exclusive, which becomes an approximation in the case of highly-coupled components. However, the algorithm guarantees that the actual benefit of the resulting migration set is at least TB: the benefit of migrating two or more components that share a communication link is greater than the sum of their individual benefits due to the message traffic along their (migrated) link.

**Implementation Support.** We have implemented the disconnected operation facilities described above in a PitM meta-level component called Disconnection Controller. This component

---

8. The probability may be calculated by measuring the ratio between intervals of sudden disconnection and network availability over a period of time.
resides on each device that is either the source or the destination of component migration. Disconnection Controller components collaborate in estimating the best set of migrating components. Each Disconnection Controller is aware of 1. the required memory of all candidate components residing on its device, 2. the total available memory on the device, 3. the frequencies of all messages of interest, 4. the dependency factors of corresponding operations, 5. the time to disconnection and connection speed in the case of anticipated disconnection, and 6. the probability of disconnection in the case of sudden disconnection.

Based on these parameters, the Disconnection Controller estimates the optimal set of components for migration, and requests that the local Admin Component effect the migration.

We illustrate this approach in the context of the TDS application. Let us assume that General’s Palm (recall Figures 1 and 2) is going to get disconnected within a given period, and that we know how much dynamic memory remains unused on the Palm. Also, let us assume that the connection speed between the Palm and the Headquarters PC is known. The goal is to maximize the functionality of the application running on General’s Palm until the connection is restored. Table 1 shows the frequency of the message traffic present on the link between the Palm and the PC and the dependency of the operations corresponding to each message. This information is used by the Disconnection Controller components to calculate the benefit associated with candidate components, shown in Table 2. Depending on time to disconnection, connection speed, and available memory on the device, the selected set of components for migration will vary. Various combinations of these parameters and the resulting migration sets are given in Table 3.

9 RELATED WORK

Our work on PitM has been primarily influenced by four research areas: architectural styles, implementation frameworks, code mobility, and disconnected operation. Below we discuss the related approaches in these four areas.

Table 1: Message traffic

<table>
<thead>
<tr>
<th>Message</th>
<th>Processing Component</th>
<th>( f_i )</th>
<th>( d_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnalyzeStrategy</td>
<td>Strategy Analyzer</td>
<td>0.06</td>
<td>1</td>
</tr>
<tr>
<td>Simulate</td>
<td>War Manager</td>
<td>0.16</td>
<td>0</td>
</tr>
<tr>
<td>Advise</td>
<td>Deployment Advisor</td>
<td>0.18</td>
<td>0</td>
</tr>
<tr>
<td>Deploy</td>
<td>Strategy Analyzer</td>
<td>0.37</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Candidate components

<table>
<thead>
<tr>
<th>Strategy Analyzer</th>
<th>War Manager</th>
<th>Deployment Advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required memory (KB)</td>
<td>13</td>
<td>9</td>
</tr>
<tr>
<td>Benefit</td>
<td>0.31</td>
<td>0.16</td>
</tr>
</tbody>
</table>

Table 3: Resulting migration sets

<table>
<thead>
<tr>
<th>Time to Disconnection</th>
<th>Connection Speed</th>
<th>Available Memory</th>
<th>Resulting Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 s</td>
<td>13 KB/s</td>
<td>50 KB</td>
<td>Strategy Analyzer</td>
</tr>
<tr>
<td>0.5 s</td>
<td>40 KB/s</td>
<td>15 KB</td>
<td>War Manager</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Deployment Advisor</td>
</tr>
<tr>
<td>1 s</td>
<td>30 KB/s</td>
<td>40 KB</td>
<td>Strategy Analyzer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>War Manager</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Deployment Advisor</td>
</tr>
</tbody>
</table>

Architectural Styles

Several good overviews of architectural styles exist [11,17,38]. In particular, [11] studies architectural styles for distributed applications. Most of those styles are variants of the client-server style and make certain assumptions that make them a poor fit for PitM applications. These assumptions include one or more of the following: centralized ownership of the applications, purely client-server or peer-to-peer interaction (but not both), lack of topological guidelines for decomposing the architecture of the application and/or its major components (e.g., clients or servers), lack of support for formal architectural modeling and analysis, limited architectural self-awareness, lack of focus on mobility, and limited support for disconnected operation. Our goal is to tailor the assumptions and characteristics of the PitM style to best address all of these issues in a principled way.

Implementation Frameworks

Central to our investigation of the issues in PitM is our implementation framework discussed in Section 5. The research and use of frameworks can be classified into six distinct generations on the basis of the achieved level of component reuse: (1) Module interconnection languages [21] enabled the reuse of components implemented in a single programming language (PL). (2) Remote procedure calls and platform-neutral data representations (e.g., [4,33]) enabled distribution and reuse across PLs. (3) Platform-neutral runtime environments and dynamic component loading (e.g., [14,24]) allowed dynamism and reuse across computing platforms. (4) Domain-specific and GUI frameworks (e.g., [15,22,31]) enabled reuse across applications. (5) Provision of infrastructure services such as naming, threading, persistence, and transaction management (e.g., [21,36,42]) introduced the possibility of reuse of architecture-level abstractions. (6) Reuse of architecture-level abstractions became an explicit focus of architectural style-based frameworks (e.g., [35,37,40]). While it exhibits properties of frameworks spanning several generations, the PitM framework is most closely related to the sixth generation.

Code Mobility

A detailed overview of existing code mobility techniques is given in [13]. 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 7, 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 (recall Section 7).

Disconnected Operation

Ensuring availability of a system during disconnection has been explored primarily in the domain of file systems. The approach is
to make the mobile computer more autonomous (i.e., less dependent on the network) by using such methods as file caching or prefetching, and lazy writeback. Example systems such as Coda [23], D-NFS [12], and Ficus [18] use optimistic replication for file caching, and reconciliation of replicas to resolve of conflicting updates. In optimistic replication, updates can be made concurrently to different file replicas, resulting in multiple versions of a file. To recover from conflicting updates, after-the-fact conflict resolution (i.e., reconciliation) actions are required to recombine multiple versions into one. Conflict resolution can be automated [34], but it may also require the intervention of the (human) owner of the file.

Our approach to disconnected operation is more similar in its nature to FarGo [20], which is concerned with the migration of components as computational elements, rather than as files, in response to disconnection. However, while FarGo handles only anticipated disconnection, we have developed a more general, risk-based approach, which can be also used in cases of sudden disconnection.

10 CONCLUSIONS AND FUTURE WORK

Over the past several decades software researchers and practitioners have proposed various approaches, techniques, and tools for developing ever larger, more complex systems. The results of these efforts have shared a number of traits: system size and complexity, possible distribution across desktop platforms, focus on modeling and analysis before implementation, accompanying development environments, explicit software architectures, and so forth. The resulting software development paradigm has been referred to as programming-in-the-large (PitL) [8]. This paper has presented an approach to address a new set of software engineering challenges that have arisen with the emergence of inexpensive, small, heterogeneous, resource-constrained, possibly embedded, highly-distributed, and highly-mobile computing platforms. While a number of the individual challenges bear similarity to those addressed by PitL, we believe that their combination and overall novelty is more appropriately described as programming-in-the-many (PitM).

The centerpiece of our approach to PitM is an architectural style. The style and its accompanying modeling, analysis, and implementation tools ensure flexible component-based system composition and interaction; efficient implementation; fine-grained distribution and deployment; dynamic reconfiguration; mobility of system models, data, and code; and continued availability in the face of connectivity losses. Additionally, the PitM architectural style introduces facilities for system self-awareness, which are leveraged in the development and evolution of long-lived, highly distributed, dynamically evolving systems whose ownership is potentially decentralized. We have provided implementation, deployment, dynamic reconfiguration, mobility, and disconnected operation support for applications in several programming languages and computing platforms (both desktop and mobile). We have applied the PitM style and these tools in the development of a number of applications to date.

While a number of pertinent issues remain unexplored, our experience thus far has been very positive. We believe that the work described in this paper provides an excellent basis upon which to conduct future investigations. We discuss several areas of our most immediate interest below.

As applications are moving to highly distributed topologies, issues such as decentralized ownership need to be properly addressed. We have only begun to investigate possible solutions for supporting decentralized ownership of an application. Our current solution is that each device stores its subsystem’s architectural model. Before a dynamic change to the application is allowed, the model is analyzed to ensure the consistency of that change with the existing configuration. A limitation of this solution is that it assumes that each device has local analysis capabilities. It also induces decisions about architectural changes based solely on local information. An alternative is to allow a device to communicate its own application model to neighboring device(s) equipped with the needed analysis facilities. As a result, such devices may have access to a more complete model of the overall system’s architecture and thus may be able to perform more meaningful analyses. On the other hand, such a collaborative approach to analysis will accentuate the issues of system security and trust. We intend to study the applicability of and tradeoffs between these two alternative approaches.

PitM is characterized by large numbers (of both models and implementations) of components, connectors, and entire architectures, as well as varying system deployments. Effective control over such large numbers of artifacts can be ensured by adding configuration management (CM) support to PitM. We have recently demonstrated that traditional CM techniques, typically applied at the level of system implementations, may be extended to effectively capture architectural entities [19]. We plan to tailor the CM capabilities resulting from that work to support PitM applications.

Finally, our longer term goal is to develop techniques for assessing PitM applications and suggesting deployment strategies that minimize network traffic and maximize performance and availability. This includes estimation of optimal component locations in a configuration, estimation of which components should be migrated, and, finally, when the migration should occur. We intend to integrate PitM’s support for architectural self-awareness and runtime monitoring (discussed in Sections 4 and 5, respectively) with existing tools for system resource analysis [10] in order to enable these estimations.

11 ACKNOWLEDGEMENTS

The ideas described in this paper emerged and matured in the course of numerous discussions with E. Dashofy, M. Gorlick, D. Heimbigner, P. Oreizy, D. Rosenblum, R. Taylor, and A. Wolf. We also wish to acknowledge M. Bhachech, N. Mehta, S. Phadke, A. Rampurwala, T. Rangwala, and V. Viswanathan, who contributed to the development of the PitM framework and/or the TDS implementation. Several students from USC’s CSCI 578 – Software Architectures class provided us with useful experience and insights into the ideas and tools behind PitM by constructing example PitM applications. Finally, we especially thank B. Boehm and R. Taylor for their generous support in obtaining the equipment used in the described research.

This material is partly 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 numbers F30602-99-C-0174 and 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. Effort also sponsored in part by Xerox.
12 REFERENCES


