# Architectural Support for Programming-in-the-Many

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

## Keywords
Programming-in-the-many (PitM), software architecture, architectural style, architectural self-awareness, deployment, distribution, mobility, dynamism, disconnected operation, Prism

## 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 involved in environment and land-use monitoring, freeway-traffic management, fire fighting, and damage surveys in times of natural disaster [28,37]. 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 approaches 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 [28] and code mobility [13] to aid application evolution "on the fly" and thus minimize application downtime; and
- disconnected operation [18,19], 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, 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, highly mobile, resource constrained setting, while others (e.g., code mobility, disconnected operation) have not been accompanied by appropriate application design support.

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 also adopted and, where necessary, adapted the results of the techniques listed above. In the process, we have developed several novel solutions to the PitM challenges, spanning six key areas:

- explicit design idioms comprising an architectural style [29,36];
- explicit meta-architecture, which enables continuous architectural monitoring, analysis, and evolution;
- a light-weight, tailorable architecture implementation, deployment, and reconfiguration infrastructure;
- explicit, first-class software connectors, which are central to both the style and the implementation infrastructure;
- native support for distributed deployment, mobility, and dynamic reconfiguration of applications; and
- 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. This combination 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 motivates our approach. Section 3 briefly describes an example application used to illustrate the concepts throughout the paper. Section 4 summarizes the rules of the PitM architectural style. Section 5 describes our infrastructure for implementing, deploying, migrating, and dynamically reconfiguring PitM applications, while Section 6 discusses our approach to disconnected operation. The paper concludes with overviews of related and future work.

## 2 PITM CHALLENGES

While PitL has for decades dealt with the engineering of large and complex software systems, PitM presents a number of additional, unique challenges. One such challenge is resource constraints. Devices on which applications reside may have limited power, network bandwidth, processor speed, memory, and display size and resolution. For example, the PalmOS operating system (OS) for the Palm Pilot device restricts the size of dynamic heap memory to 256KB. Constraints such as these demand highly efficient software systems, in terms of computation, communication, and memory footprint. They also demand more unorthodox solutions such as "off-loading" (i.e., distributing) non-essential parts of a system to other devices.

Another challenge faced by PitM is the heterogeneity of the hardware and the "utility" software (operating systems, programming languages and compilers, middleware, and software libraries and frameworks). Recent developments in the PitL arena have largely eliminated heterogeneity: while there still exist (different variants of) three dominant desktop computing platforms, much of the support on them has been standardized. For example, variants of Unix (and, to a lesser extent, Windows) are capable of running on different hardware platforms, Java and its accompanying middleware facilities (RMI and EJB) as well as different implementations of CORBA are platform-independent, as are data formats such as PDF and XML, and networking protocols such as HTTP. Although similar standardization efforts are undertaken for the emerging new class of mobile devices (e.g., various wireless network protocols), the world of PitM is substantially less homogeneous, characterized by proprietary operating systems (e.g., PalmOS), specialized dialects of existing programming languages (e.g., Java KVM [39] for the PalmOS), and device-specific data formats.

Yet another challenge of PitM is effective support for inter-device interaction and code mobility. As numerous new, small, mobile platforms emerge, their developers make trade-offs to address the computing constraints imposed by the platforms. The infrastructures of the emerging novel or experimental technologies may thus be missing certain services for reasons of efficiency (or accidental omission). For example, Java KVM does not support non-integer numerical data types or server-side sockets. Similarly, typically employed techniques for code mobility, such as Java serialization or XML encoding, may not be supported because they are computationally too expensive.

Finally, PitM inherits many of the challenges faced in PitL. Application modeling, analysis, simulation, and generation are problems on which software engineering researchers and practitioners have been working actively for several decades. These problems are only amplified in the highly distributed, heterogeneous, and mobile world of PitM. Certain techniques recently devised for dealing with these problems, such as component-based software development, might prove effective in the context of PitM. However, the manner and extent to which those techniques must be adapted remain open issues.

For this very reason, the basic principle of our approach to supporting PitM has been to reuse solutions, and thus reap the benefits, of existing software engineering research and practice (i.e., PitL) whenever possible, inventing new solutions only as necessary. In particular, one area from which we believe we can gain a lot of leverage is software architecture [29,36]. Several aspects of architecture-based development (component-based system composition, explicit software connectors, architectural styles, upstream system analysis and simulation, and support for dynamism) make it a particularly good fit for the needs of PitM. Software architecture is indeed the centerpiece of our approach. Our support for system modeling, analysis, implementation, resource constrained computation, distributed deployment, dynamic reconfiguration, mobility, and disconnected operation all leverage the architectural basis of the approach, as detailed in the remainder of the paper.

## 3 EXAMPLE APPLICATION

To illustrate the concepts introduced in this paper, we use an application 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 application depicted in Figure 1 addresses military Troops Deployment and battle Simulations (TDS). A computer at *Headquarters* 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 hand-held 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 deployments. The *General* sees a summarized view of the entire battlefield (shown); additionally, the *General* is capable of see-

ing 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.

The TDS application has provided an effective platform for investigating a number of PitM concepts. TDS has been designed, analyzed, implemented, deployed, migrated (both to streamline the application and as a result of network disconnection), and dynamically evolved using the techniques described in this paper. Additionally, several aspects of TDS embody the concept of multiplicity inherent in PitM ("many"). The application is implemented in four dialects of two programming languages: Java JVM, Java KVM, C++, and Embedded Visual C++ (EVC++). The application is deployed on five devices, four of which are mobile. The TDS subsystem on each device can run using an arbitrary number of threads of control. The devices are of three different types (Palm Pilot, iPAQ, PC), running three OSs (PalmOS, WindowsCE, and Windows98, respectively); in addition, several analysis components used in support of code mobility and disconnected operation run on a fourth platform (Sun) and OS (Unix). In the instance of TDS shown in Figures 1 and 2, sixteen software components deployed across the five devices interact via fifteen software connectors. The connectors enable two principal communication paradigms (client-server and peer-to-peer, discussed in Section 4) implemented in four different connector categories (see Section 5.1).

# 4 AN ARCHITECTURAL STYLE FOR PITM

We have developed an architectural style that 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 monitoring and analysis, 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 [38]. 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. 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 the key enabler of our support for runtime reconfiguration, mobility, and disconnected operation. A *message* consists of a name and a 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 and/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 (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. Moreover, 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.

## 4.1 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_iPAQ_AvailableTroops* and *G_AvailableTroops* components on *iPAQ* and *Palm-3* devices in Figure 2). In order to maintain component decoupling, the side ports exchange peer messages through "peer" connectors (unlabeled circles in Figure 2). Basic peer connectors have simple message broadcast semantics: a peer message incoming on any port is forwarded as an outgoing message through all of the connector's remaining ports. Other routing semantics can be implemented in peer connectors as discussed in Section 5.1.

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_iPAQ_AvailableTroops* and *G_AvailableTroops*. The PitM style also disallows the possibility of exchanging messages between a peer and a horizontal connector (which would, in effect, convert peer messages into requests/notifications, and vice versa).

While providing an effective solution for peer-to-peer interaction, the introduction of a third component port and an additional connector type in the PitM style allows for the construction of complex configurations. This is especially true in degenerate cases in which the architect decides to rely primarily on the side component ports and peer connectors, which the style allows. We believe that the goal of a style should be to provide architects with enough guidance to arrive at effective configurations, but also enough flexibility to address a multitude of development situations and needs. A style, therefore, cannot and should not prevent the design of "bad" architectures *a priori*. PitM attempts to provide the balance between guidance and flexibility by coupling the design rules of the C2 style, which have been proven

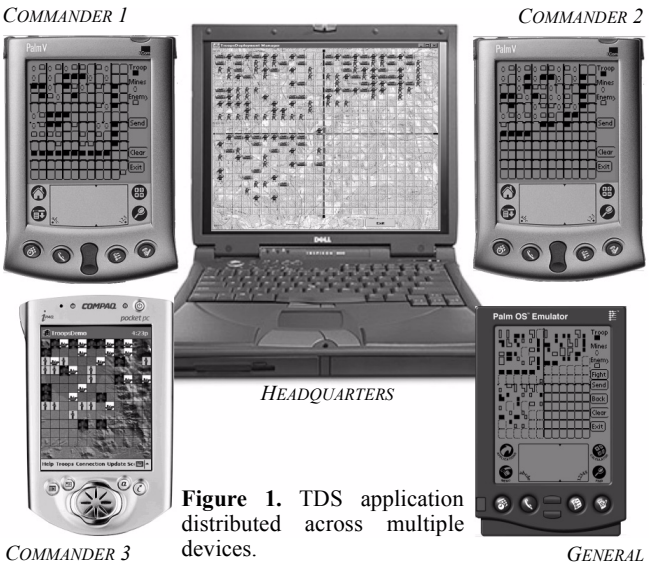


**Figure 1.** TDS application distributed across multiple devices.

effective in the construction of distributed applications, with the added facilities of PitM. It is, then, the architect's obligation to enact good design practices and apply the resulting rules most effectively.

## 4.2 Architectural Self-Awareness

PitM supports architectures at two levels: application-level and meta-level. The role of components at the PitM meta-level is to observe and/or facilitate different aspects of the execution, dynamic evolution, mobility, and disconnected operation of application-level components. Application-level and meta-level components execute side-by-side in PitM. Meta-level components are aware of application-level components and may initiate interactions with them, but not vice versa. The PitM style rules apply to both component categories: meta-level components also engage in connector-mediated, message-based interactions with each other (and with application-level components).

In support of this two-level architecture, PitM distinguishes among three types of messages. Similarly to C2, *ApplicationData* messages are used by application-level components to communicate during execution. The other two message types, *ComponentContent* and *ArchitecturalModel*, are used by PitM meta-level components. *ComponentContent* messages contain mobile code and accompanying information (e.g., the location of a migrant component in the destination configuration), while *ArchitecturalModel* messages carry information needed to perform architecture-level analyses of prospective PitM configurations.

We have extensively used special-purpose components, called *Admin Component*s, whose task is to exchange *ComponentContent* messages and facilitate the deployment and migration of application components across devices (see Section 5). 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 [26].[1]

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

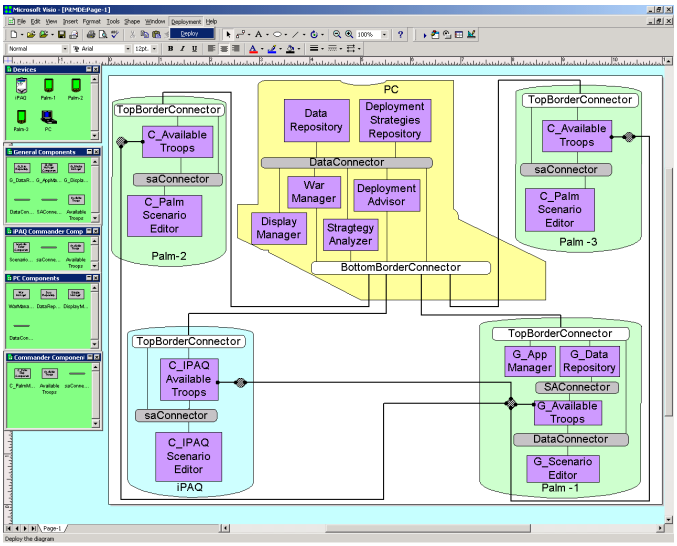


**Figure 2.** Architecture of the TDS application, displayed in *Prism*, the PitM deployment environment. The unlabeled circles connecting components across the hand-held devices represent *peer* connectors.

Meta-level components may be application independent (e.g., *Continuous Analysis*, *Admin Component*), 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*.

## 4.3 Border Connectors

The third significant departure from C2 in formulating the PitM style is the key role of connectors 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 architectural 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.

## 4.4 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 maintains 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

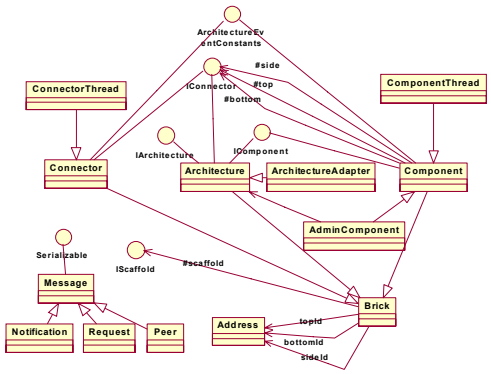


**Figure 3.** Class design view of the PitM implementation framework.

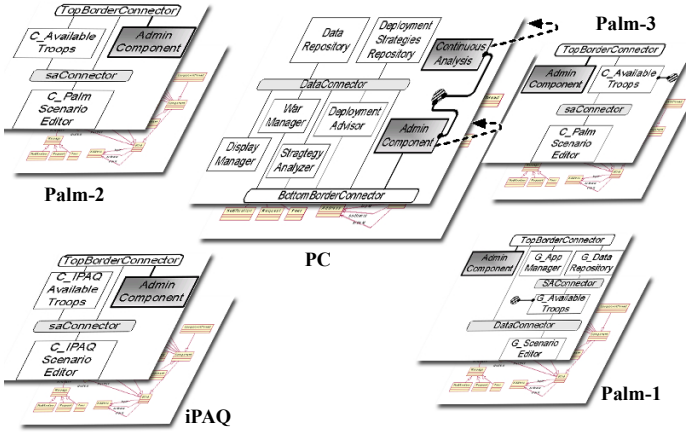


**Figure 4.** Layered construction of an application using the PitM implementation framework. The application is distributed across five devices, each of which is running the framework. Meta-level components (highlighted in the figure) may control the execution of application-level components via PitM messages or via pointers to the local *Architecture* object (shown in the subarchitecture on the *PC*).

General's subarchitecture is able to make direct orders (by sending peer messages through peer connectors) to the Commanders' *C_AvailableTroops* components to reposition troops across battlefield quadrants. The meta-level components are elided for clarity; the *Admin Component* and *Continuous Analysis* meta-level components are shown in Figure 4.

# 5 THE INFRASTRUCTURE

## 5.1 Implementation Support

PitM provides stylistic guidelines for composing large, distributed, mobile 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 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 EVC++, and Python. The framework's application programming interface (API) has been designed such that it is consistent with the C2 implementation framework [24]: applications implemented using the C2 framework can execute without any modifications on top of the PitM framework. Note that PitM applications cannot execute using C2's framework because that framework does not support side component ports, peer messages, peer connectors, or PitM's meta-architecture.

**Framework's 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. 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 exported by every *Brick* (component, connector, or entire architecture). *iScaffold* directly aids architectural self-awareness by allowing probes and monitors of the runtime behavior of a *Brick*, further discussed below.

**Connectors.** To support a variety of development situations in the PitM setting, we have implemented a library of PitM connectors. These include *basic connectors* that support both synchronous and asynchronous message broadcast, multicast, and unicast. Furthermore, we have applied our technique for component interactions across process and machine boundaries [7] in implementing *border connectors*. A particularly interesting kind of border connector is the infra-red (IR) connector, which leverages the commonly present IR communication ports on hand-held devices to enable short-range, wireless interaction of PitM components across those devices. In order to communicate across heterogeneous platforms, runtime environments, and programming languages, we use XML as a medium for message passing. To ensure *secure* communication in a highly distributed, dynamic, mobile setting, we have implemented an encryption module [3] that may be added to any connector. For example, we have used this module with the IR connectors to encrypt and marshal information during communication between secure parts of an architecture. Finally, we have developed *multi-versioning connectors* (MVCs) [31] in order to support upgrading of application functionality at runtime and in a reliable manner. 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) and monitor the volume of incoming and outgoing message traffic for each component version. These statistics provide information about which version to retain in the system. For example, if a multi-versioned component is likely to be migrated to another device, its most desirable version need not be the most correct one, but rather the version with the lowest dependency factor (see Section 6).

**Optimizations.** 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, messages are transported via border connectors and added to the message queues in the recipients' address spaces. The control over the thread pool and the message queue is exercised from PitM's meta-architecture components, via (a pointer to) the *Architecture* object, as shown in Figure 4. 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).

**Applications.** The PitM style and framework implementations have been used in the development of over a dozen applications involving Palm Pilot and Compaq iPAQ devices. 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. As an example, we show in Figure 4 an implementation configuration of the TDS architecture 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 *Palm*s 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. As discussed above, we use XML-based connectors to enable the interaction of application components across the language boundaries.

**Using the Framework.** The first step a developer (or tool generating an implementation from an architectural description)[2] 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 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. 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 application-level connectors and thus exercise control over a particular portion of the *Architecture* (e.g., *Admin Component* in Figure 4). Alternatively, meta-level components may remain unwelded and may instead exercise control over the entire *Architecture* object directly (e.g., *Continuous Analysis* component in Figure 4).

## 5.2 Deployment Support

Our support for deployment directly leverages the PitM implementation infrastructure. We have developed a custom solution for deploying applications instead of trying to reuse existing capabilities (e.g., [16]) because of the facilities provided by the PitM style (*ComponentContent* messages, meta-level architecture) and the light weight of the PitM implementation framework, which is critical for small, resource constrained devices such as the Palm Pilot. 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. 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 *ComponentContent* messages through *Border Connectors* to any device to which they are connected.

We have integrated and extended the COTS Microsoft Visio tool to develop *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 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 the 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 the *Admin Component* on the device that contains it. The code is sent to the requesting *Admin Component* using a PitM *ComponentContent* 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.

## 5.3 Mobility and Dynamic Reconfiguration Support

We use the same technique for supporting runtime component mobility as we do for deployment: *Admin Components* exchanging *ComponentContent* messages that contain mobile code. Again, the reason for choosing this approach instead of adopting an existing code mobility technique [13] is its native support in the PitM architectural style and

```
add(DataRepository: source PC): PC
add(DeploymentStrategiesRepository: source PC): PC
add(DataConnector: source none): PC
add(C_IPAQAvailableTroops: source local): iPAQ
add(C_IPAQScenarioEditor: source PC): iPAQ
add(SaConnector: source none): iPAQ
weld(DataRepository,DataConnector): PC
weld(DeploymentStrategiesRepository,DataConnector): PC
weld(C_IPAQAvailableTroops,SaConnector): iPAQ
weld(TopBorderConnector,C_IPAQAvailableTroops): iPAQ
weld(SaConnector,C_IPAQScenarioEditor): iPAQ
peerWeld(G_AvailableTroops,SideBorderConnector):Palm-1
```

**Figure 5.** Partial description of the configuration created by Prism for the TDS application, with component sources and destinations shown. Source devices are denoted as "none" in the case of connectors that use the base implementation (asynchronous message broadcast) provided by the PitM framework.

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 [26].

efficient implementation in the PitM framework. We illustrate the technique using the TDS application. 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* object's *iScaffold* 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:[3]

1. If necessary, the migrant component is disconnected from its attached connectors using the framework's *unweld* and *peerUnweld* methods. 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*.
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 attached four devices.
4. Once received by the *Border Connector*s 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. 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 PalmOS. For this reason, we have considered several additional mobility techniques [25], 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 [31] for details of how such messages are captured). Once the migrant component is received at its destination, it is loaded into memory[4] 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 *weld*ed and enabled to exchange messages with components in 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.

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 [28]). Application integrity is achieved in our approach by exchanging *ArchitecturalModel* messages along with *ComponentContent* messages and leveraging *Continuous Analysis* meta-level components.
4. 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 EVC++ framework using DLLs.

# 6 SUPPORT FOR DISCONNECTED OPERATION

Due to the nature of mobile devices, their network 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, stays within the memory constraints posed by the device, and can be migrated within the time remaining before disconnection occurs. Specifically, we identify the following factors to be pertinent in deciding which components should be migrated:

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

Several of these factors that are not self-evident are described below.

## 6.1 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:

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

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 the synchronization of component replicas, PitM attaches a *degraded mode* indicator to each operation exported by a component. The indicator is intended to reflect an operation's dependence on component state: some operations do not depend on component state and are fully accessible during disconnection (*allowed*); other operations are *delayed* until the connection is restored; finally, 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 of enemy troops might change significantly during disconnection). We have extended our architecture modeling support [26] with a *degraded mode* tag for each service provided by a component. The corresponding support in the PitM implementation framework is ensured via runtime flags for disallowed operations and a separate message queue corresponding to delayed operations. These messages, which will ultimately require remote processing, will be forwarded to their destinations by the local *Border Connector* once the connection is restored.

### 6.2 Message Frequency

We assume that the frequency of each message ($f_i$) present on the network link that is going to be disconnected can be observed over a period of time. We have extended *Border Connectors* with the capability to monitor message frequencies. The objective of our approach is to minimize the message traffic that would need to go through the "broken" link(s) during the period of disconnection.

### 6.3 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 [26] 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:

$$D = \frac{\sum_{i=1}^{N} d_i * f_i}{N}$$

where $d_i$ is the dependency and $f_i$ the frequency of messages resulting in $i$-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.

### 6.4 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 the case of anticipated disconnection, we propose migrating components before the disconnection occurs.

### 6.5 Minimizing the Risk of Disconnection

We minimize the risk of disconnection by selecting components whose migration will give the application the best chance of performing normally on the disconnected device. 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 the candidate component as a result of these messages, do the following:

   $$\text{Benefit} \leftarrow \text{Benefit} + f_i * (1 - d_i)$$

   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:

$$TAM = \min(M, t * nb)$$

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 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 spanning 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 or equal to the total available memory on the corresponding device.

This problem is known 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 subset of the 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 [5]. The algorithm runs in $O(n*TAM)$. The algorithm assumes that the benefits of individual components are mutually exclusive, thus becoming 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 or equal to the sum of their individual benefits due to the message traffic along their (migrated) link.

### 6.6 Implementation Support

We have implemented the disconnected operation facilities described above in a PitM meta-level component called *Disconnection Controller*. This component resides on each device that is either the source or the destination of component migration. *Disconnection Controller*s collaborate in estimating the best set of migrating components. Each *Disconnection Controller* is aware of

1. the required memory of all 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. probability of disconnection, in the case of sudden disconnection. In our current implementation, each *Border Connector* calculates the probability by measuring the ratio between intervals of sudden disconnection and network availability over a period of time.

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.

### 6.7 Example Application

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 of migration associated with candidate components, shown in Table 2.

**Table 1:** Message traffic

| Message | Processing Component | $f_i$ | $d_i$ |
|---|---|---|---|
| AnalyzeStrategy | Strategy Analyzer | 0.06 | 1 |
| Simulate | War Manager | 0.16 | 0 |
| Advise | Deployment Advisor | 0.18 | 0 |
| Deploy | Strategy Analyzer | 0.37 | 0 |

**Table 2:** Candidate components

| | Strategy Analyzer | War Manager | Deployment Advisor |
|---|---|---|---|
| **Required memory (KB)** | 13 | 9 | 5 |
| **Benefit** | 0.31 | 0.16 | 0.18 |

**Table 3:** Resulting migration sets

| Time to Disconnection | Connection Speed | Available Memory | Resulting Set |
|---|---|---|---|
| 1 s | 13 KB/s | 50 KB | Strategy Analyzer |
| 0.5 s | 40 KB/s | 15 KB | War Manger<br>Deployment Advisor |
| 1s | 30 KB/s | 40 KB | Strategy Analyzer<br>War Manager<br>Deployment Advisor |

Depending on the 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.

## 7 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.

### 7.1 Architectural Styles

Several good overviews of architectural styles exist [11,17,36]. 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; and limited architectural self-awareness, focus on mobility, and 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.

### 7.2 Implementation Frameworks

Central to our investigation of the issues in PitM is our implementation framework. 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 [8] enabled the reuse of components implemented in a single programming language (PL). *(2)* Remote procedure calls and platform-neutral data representations (e.g., [4,32]) enabled distribution and reuse across PLs. *(3)* Platform-neutral runtime environments and dynamic component loading (e.g., [14,23]) enabled dynamism and reuse across computing platforms. *(4)* Domain-specific and GUI frameworks (e.g., [15,30]) enabled reuse across applications. *(5)* Provision of infrastructure services such as naming, threading, persistence, and transaction management (e.g., [20,34,40]) 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., [35,38]). While it exhibits properties of frameworks spanning several generations, the PitM framework is most closely related to the sixth generation.

### 7.3 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 5.3, 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 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 5.3).

### 7.4 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 [21], D-NFS [12], and Ficus [18] use optimistic replication for file caching, and reconciliation of replicas to resolve 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 [33], 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 [19], which has recently added support for migrating 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.

## 8 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, accopmanying 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 support for these capabilities in several programming languages and computing platforms (both desktop and mobile). We have applied the PitM style and tools in the development of a number of applications to date. Our evaluation of the PitM concepts covered in the paper is summarized in Table 4.

**Table 4:** Evaluation of the PitM concepts

| Concept | Property – Evaluated by |
|---|---|
| PitM Style | • viability – based on a well-known style (C2)<br>• flexibility – side ports, peer messages, peer connectors<br>• applicability – a number of applications built |
| Implement. Framework | • PL independence – Java, C++, Python<br>• simplicity – 1500 SLOC; 13 user-relevant classes<br>• efficiency – small footprint; adjustable num. of threads<br>• heterogeneity – different PLs, connector implementations |
| Dynamic Reconfig. | • simplicity – leverages style-imposed topology<br>• flexibility – adaptable connectors implemented<br>• reliability – analysis via architectural self-awareness |
| Deployment & Mobility | • portability – multiple platforms and PLs<br>• native support – *Component Content* messages<br>• flexibility – multiple connector implementations<br>• reliability – analysis via architectural self-awareness<br>• reflexivity – message-based state replication |
| Disconnected Operation | • simplicity and reflexivity – based on style and framework<br>• resource awareness – time, bandwidth, memory<br>• correctness of static behavior – *Continuous Analysis*<br>• awareness of dynamic behavior – message monitoring<br>• availability – minimizes lost/postponed messages<br>• efficiency – algorithm runs in polynomial time |

While our experience thus far has been very positive, a number of pertinent issues remain unexplored. We believe that the work described in this paper provides an excellent basis upon which to conduct future investigations. Our future work will span issues such as ensuring trust in PitM applications, supporting configuration management of the many involved artifacts, and automatically discovering the hardware devices and/or software components available on the network at a given time. We discuss two additional areas of our most immediate interest in more detail 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.

Another critical issue associated with highly distributed, mobile, possibly embedded systems is performance [22]. To date, we have deliberately chosen to provide developer support for PitM in mainstream, general-purpose programming languages. This decision induces certain performance penalties. For example, our benchmarks indicate that the Java version of the PitM implementation framework allows two components to exchange one million messages in under 15 seconds when running on an Intel Pentium II 300MHz processor. However, the same configuration is several orders of magnitude slower on the more resource constrained Palm Pilot, which may be unacceptable for many applications with real-time requirements. We thus intend to explore the use of optimized compilation [1] and special-purpose languages for embedded systems [22] to address that problem. Our longer term goal is to develop techniques for actively

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 distributed 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 with existing tools for system resource analysis [10] in order to enable these estimations.

## 9 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, M. Pai, S. Phadke, A. Rampurwala, T. Rangwala, and V. Viswanathan, who contributed to the development of the PitM implementation framework and/or the TDS application. 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. We especially thank B. Boehm and R. Taylor for their generous support in obtaining the equipment used in the described research. Finally, A. van der Hoek gave us valuable feedback on an earlier draft of this paper.

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.

## 10 REFERENCES

1. S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. *PLDI'93*, June 1993, Albuquerque, NM, pp. 126-138
2. Y. Barbaix, et al. Tuning Parameters for Component Based Design with Memory Constraints. *ECOOP 2000*.
3. M. Bellare, et al. A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation. *38th Annual Symposium on Foundations of Computer Science, 1997.*
4. A. Birrell and B. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, Feb. 1984.
5. T. H. Cormen, et al. Introduction to Algorithms. *MIT Press*, 2000.
6. P. Crescenzi, and V. Kann. A Compendium of NP Optimization Problems. http://www.nada.kth.se/nada/theory/problemlist.html
7. 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.
8. F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE TSE*, June 1976.
9. W. Emmerich, et al. Implementing Incremental Code Migration with XML. *ICSE 2000*, Limerick, Ireland, Jun 2000.
10. P. H. Feiler and J. J. Walker. Adaptive Feedback Scheduling of Incremental and Design-To-Time Tasks. *ICSE 2001*, Toronto, Canada, May 2001.
11. R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UCI, June 2000.
12. M. E. Fiuczynski and D. Grove. A Programming Methodology for Disconnected Operation. *Technical Report*, University of Washington, March 1994.
13. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE TSE*, May 1998.
14. A. Goldberg. Smalltalk-80: The Language. *Addison-Wesley*, 1989.
15. I. F. Haddad. X/Motif Programming. *Linux Journal*, May 2000.
16. R. S. Hall, D. M. Heimbigner, and A. L. Wolf. A Cooperative Approach to Support Software Deployment Using the Software Dock. *ICSE'99*, Los Angeles, CA, May 1999.
17. M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. *ESEC/FSE'99*, Sep. 1999.
18. J. S. Heidemann et al., Primarily Disconnected Operation: Experiences with Ficus. *Second Workshop on Management of Replicated Data*. IEEE, November 1992.
19. O. Holder, I. Ben-Shaul and H. Gazit, Dynamic Layout of Distributed Applications in FarGo. *ICSE'99*, May 1999.
20. R.E. Johnson. Documenting Frameworks as Patterns. *OOPSLA'92*. Vancouver, BC, Canada. 1992.
21. J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems* Feb. 1992, Vol. 10, No. 1, pp. 3-25.
22. E. A. Lee. Embedded Software. Technical Memorandum UCB/ERL M001/26, UC Berkeley, CA, July 2001.
23. T. Lindholm and F. Yellin. The Java Virtual Machine Specification. 2nd Edition Java Series. *Addison Wesley* 1999.
24. N. Medvidovic, et al. Reuse of Off-the-Shelf Components in C2-Style Architectures. *ICSE'97,* Boston, MA, May 1997.
25. N. Medvidovic and M. Rakic. Exploiting Software Architecture Implementation Infrastructure in Facilitating Component Mobility. *Workshop on Software Engineering and Mobility*, Toronto, Canada, May 2001.
26. N. Medvidovic, et al. A Language and Environment for Architecture-Based Software Development and Evolution. *ICSE'99*, Los Angeles, CA, May 1999.
27. N. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *ICSE 2000,* Limerick, June 2000.
28. P. Oreizy, et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and Their Applications*, 14(3), May/Jun 1999.
29. D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.
30. J. Prosise. Programming Windows with MFC. *Microsoft Press*, 2nd Edition. 1999.
31. M. Rakic, and N. Medvidovic, Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Symposium on Software Reusability*, May 2001.
32. H.C. Rao. Distributed Application Framework for Large-Scale Distributed Systems. *13th International Conference on Distributed Computing Systems*, pp. 31-38, 1993.
33. P. Reiher, et al. Resolving File Conflicts in the Ficus File System. *USENIX*, pp. 183-195. Boston, MA, USENIX. June, 1994.
34. B. Shannon, et al. Java 2 Platform, Enterprise Edition: Platform and Component Specifications. *Addison Wesley* 2000.
35. M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE TSE*, 21(4), April 1995.
36. M. Shaw, and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.
37. G. S. Sukhatme and J. F. Montgomery. Heterogeneous Robot Group Control and Applications. *AUVS 99 Conference*, 1999.
38. R.N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, 22(6), June 1996.
39. Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm.
40. A Discussion of the Object Management Architecture (OMA) Guide, OMG, 1997.