Middleware for Software Architecture-Based Development in Distributed, Mobile, and Resource-Constrained Environments

Marija Mikic-Rakic Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{marija,neno}@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 provides a description and evaluation of a middleware intended to support software architecture-based development of applications in the PitM setting. The middleware provides implementation-level support for the key aspects of PitM application architectures: components, connectors, architectural configurations, and communication events. Additionally, the middleware directly facilitates several system qualities necessitated by PitM, including light weight, distribution, mobility, application context awareness, asynchrony, and support for disconnected operation. Our middleware has been applied successfully in a number of applications and used as an educational tool in a graduate-level embedded systems course. While a number of issues remain to be explored, our experience with the middleware thus far has been very positive, indicating that the principles of architecture-based software development can be successfully applied in the PitM setting.

Status: Student

Presentation Type: Regular
1. Introduction

The software systems of today are rapidly growing in size, complexity, amount of distribution, heterogeneity of constituent 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, PDAs). In turn, this has resulted in an increased demand for software applications, outpacing our ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them. One can now envision a number of complex software development scenarios involving fleets of mobile devices used in environment and land-use monitoring, freeway-traffic management, fire fighting, and damage surveys in times of natural disaster. Such scenarios present daunting technical challenges: effective understanding of existing or prospective software configurations; rapid composability and dynamic reconfigurability of software; mobility of hardware, data, and code; scalability to large amounts of data, numbers of data types, and numbers of devices; and heterogeneity of the software executing on each device and across devices. Furthermore, software often must execute 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. We refer to the development of software systems in the described setting as \textit{programming-in-the-many (PitM)}, in order to distinguish it from the commonly adopted software engineering paradigm of \textit{programming-in-the-large (PitL)} [10].

Our preliminary studies have indicated that a promising approach to developing software systems in the PitM setting is to employ the principles from the emerging body of work on software architectures [25]. \textit{Software architectures} provide high-level abstractions for representing structure, behavior, and key properties of a software system [36]. They are described in terms of components, connectors, and configurations. Architectural \textit{components} describe the computations and state of a system; \textit{connectors} describe the rules and mechanisms of interaction among the components; finally, \textit{configurations} capture topologies of components and connectors. Software \textit{architectural styles} define a vocabulary of component and connector types and a set of constraints on how instances of these types may be combined in a system or family of systems. When designing a software system, selection of an appropriate architectural style becomes a key determinant of the system’s success. Examples of styles include pipe and filter, blackboard, client-server [36], GenVoca [1], and C2 [37]. We have recently developed and applied a new architectural style, PitM [25], which is directly targeted at heterogeneous, highly distributed, highly mobile, resource constrained, possibly embedded systems.

Architectural styles provide design-level guidelines for composing software systems. For these guidelines to be truly useful in a development setting, they must be accompanied by support for their implementation (i.e., architectural “middleware” [12]). This paper provides a description and evaluation of a middleware we have developed to support the implementation of PitM-style architectures. Our middleware directly leverages a software system’s architecture in enabling the system’s implementation, distribution, and mobility. In order to be effective in the PitM setting, the middleware provides several additional capabilities that currently existing middleware solutions [11] do not support. These include (extremely) light weight, asynchrony, application context awareness [5], and support for disconnected operation [38]. The key benefit of our middleware is its support for the identified capabilities in a manner that not only preserves the system’s architectural abstraction, but uses that abstraction as the foundation of system development and evolution.

The rest of the paper is organized as follows. Section 2 presents our objectives for an architecture-based middleware for PitM. Section 3 outlines the architectural style used as the basis of this work. Section 4 presents the middleware’s design and implementation, and evaluates them with respect to the objectives identified in Section 2. The paper concludes with overviews of related and future work. Finally, the reader is encouraged to consult the example provided in Appendix A for illustrations of the main concepts introduced in the paper.

2. Middleware Objectives

Several researchers have pondered the issue of implementing software systems in highly distributed, mobile, resource constrained, possibly embedded environments [5,6,12,23,29,30,38]. Based on these studies, as well as the characteristics of applications in this setting, we have identified the following to be the key objectives of an architectural middleware for PitM.

- \textit{Architectural abstractions} – An architecture-based middleware should provide (1) implementation-level support for architectural abstractions (components, connectors, configurations, communication events, ports, and so on), (2) constraint enforcement for the rules of the style which it supports, and (3) reusable implementation of component interaction mechanisms.
Asynchrony – Since the connections found in PitM applications are intermittent, service requesters and providers may not be accessible at the same time, necessitating support for asynchronous communication [5].

Awareness – The highly distributed, decentralized, mobile, potentially long-running PitM applications must be aware of their execution context in order to properly adapt to any context changes. For this reason, the middleware must be able to monitor and inform the running application of its execution context [4].

Delivery guarantees – In resource-constrained and embedded systems, the utility of a service often directly depends on how many and at what times it has been delivered [23]. The middleware should thus provide support for service delivery guarantees and real-time constraints.

Disconnected operation – Due to the nature of mobile devices, their network connections are intermittent, with periods of disconnection [38]. The middleware should support system operation during such periods.

Efficiency – The middleware should impose minimal overhead on an application's execution. The goal is to enable efficient execution of applications on platforms with varying characteristics (e.g., speed, capacity, network bandwidth).

Heterogeneity – In order to maximize the flexibility of PitM application implementations and the potential for reuse of off-the-shelf (OTS) components, the middleware should support heterogeneous programming languages (PL), operating systems (OS), hardware platforms, and network protocols.

Mobility – Hardware devices in PitM systems are highly mobile and resource constrained. Properly addressing these characteristics may require “on-the-fly” distribution of non-essential parts of a system to external devices, and migration of data and code to accommodate context changes [15].

Reconfigurability – Constantly changing environments in which PitM applications execute require seamless adaptation to the changes without shutting down the application [29]. The middleware should thus support dynamic system reconfiguration.

Scalability – In order to effectively manage the large numbers of devices, components, and communication events present in PitM systems, the middleware should be scalable.

Security – All nodes in a (changing) network comprising a PitM application cannot be trusted a priori [6]. For this reason, the middleware should support secure communication between components.

3. Architectural Style

The PitM style is targeted at heterogeneous, highly distributed, mobile, resource constrained, possibly embedded systems [25]. In formulating the PitM style, we have leveraged our extensive experience with the C2 architectural style, which is intended to support highly distributed applications in the graphical user interface (GUI) domain [37]. PitM-style components maintain state and perform application-specific computation. The components may not assume a shared address space, but instead interact with other components by exchanging events (also referred to as messages) via their three communication ports (named top, bottom, and side); each port may be attached to at most one connector port. Connectors in the PitM style mediate all component interactions by controlling the distribution of messages. A message comprises a name and a set of typed parameters.

A message in the PitM style is either a request for a component to perform an operation, a notification that a given component has performed an operation and/or changed its state, or a peer message used in direct (peer-to-peer) communication between components. Request messages are sent through the top port, notifications through the bottom port, and peer messages through the side port of a component. The distinction between requests and notifications ensures PitM’s principle of substrate independence, which mandates that a component in an architecture may have no knowledge of or dependencies on components below it. In order to preserve this property, two PitM components may not engage in interaction via peer messages if there exists a vertical topological relationship between them.\(^1\)

A PitM-style 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 and mobility. PitM distinguishes between two types of connectors. A horizontal connector enables the request-notification type of communication among components through their top and bottom ports, while a peer connector enables peer-to-peer communication among components through their side ports. The PitM style does not allow a peer and a horizontal connector to exchange messages; this would, in effect, convert peer messages into requests/notifications, and vice versa. PitM connectors support both asyn-

\(^1\) For example, DataRepository on the PC and G.ScenarioEditor on the Palm-1 in Figure A2 of the Appendix may not exchange peer messages since one component is above the other; however, no vertical topological relationship exists between C.iPAQ.AvailableTroops on the iPAQ and G.AvailableTroops on the Palm-1, meaning that they may communicate via peer messages.
chronous and synchronous message-based communication between components.

The PitM style natively supports connectors that span device boundaries. Such connectors, called border connectors (see Figure 1), enable the interactions of components residing on one device with components on other devices. A border connector marshals and unmarshals data, code, and architectural models (further discussed in Section 4), and dispatches and receives messages across the network. It may also perform data compression for efficiency, and authentication, authorization, and encryption for security. A border connector may facilitate communication via requests and notifications (horizontal border connector), or via peer messages (peer border connector).

An architecture description language (ADL) [26] is used as the basis of modeling and analyzing PitM architectures. A PitM architecture is described in three parts: component types, connector types, and architectural topology (see Figure A3 of the Appendix). Each component has a name, and a set of provided and required services with associated behavior. The PitM ADL attaches dependency and degraded mode tags to each provided service of a component. The dependency tag denotes components’ services that are needed for the completion of a given service, while the degraded mode tag denotes the availability of a given service in the case of disconnected operation (see Figure A3 of the Appendix and Section 4.8 for details).

4. Architectural Middleware

The PitM middleware comprises an extensible framework of implementation-level classes representing the key elements of the PitM style (e.g., components, connectors, messages) and their characteristics (e.g., a message has a name and a set of parameters). An application architecture is then constructed by extending the appropriate classes in the middleware with application-specific detail.

4.1. Middleware Core

The design of PitM middleware’s core is depicted in Figure 1. The classes shown are those of interest to the user of the middleware (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). 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 is implemented as a set of interacting Architecture objects. Components in an architecture communicate by exchanging Request, Notification, or Peer Messages via the Component class’s send and handle methods. Messages are routed by connectors. Finally, IScaffold is an interface exported by every Brick (component, connector, or entire architecture). IScaffold directly aids architectural awareness by allowing probing the runtime behavior of a Brick. To date, this middleware has been implemented in Java JVM and KVM [39], C++ and Embedded Visual C++ (EVC++), and Python. Each implementation of the middleware core is quite small, averaging 1,750 SLOC.

The first step a developer (or tool automatically generating an implementation from an architectural description [26]) takes is to subclass from the Component or ComponentThread 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, and of connectors selected from the reusable connector library. Finally, attaching component and connector instances into a configuration is achieved by using the weld and peerWeld methods of the Architecture class.

The PitM middleware’s core satisfies several of our objectives identified in Section 2. It supports architectural abstractions by providing classes for representing each architectural element, with methods for creating, manipulating, and destroying the element. The core also enforces the structural constraints of the PitM style (e.g., a component’s bottom port may be only attached to a connector’s top port) and encapsulates all interaction mechanisms inside Connector objects. The middleware core implements asynchronous message passing (example shown in the Figure A4 of the Appendix) using the technique further discussed in Section 4.2 below. One facet of middleware core’s heterogeneity is its implementation in multiple PLs, along with special-pur-
pose connectors that leverage XML [25] and/or encapsulate OTS technologies [9] for bridging the PL boundaries. Furthermore, different implementations of the middleware have been ported to four computing platforms, two of which are mobile: Compaq iPaq, Palm Pilot, Unix, and PC. The middleware also provides a set of GraphicsBinding components that encapsulate UI services. A GraphicsBinding issues requests each time a UI event occurs and receives notifications to display information. Cross-platform portability is aided by interchanging different GraphicsBinding components. Finally, runtime reconfigurability is achieved in the middleware core with the help of three mechanisms:

1. the methods exported by the Architecture object to add, remove, weld, and unweld a component,
2. the ability of Connectors to dynamically add and remove communication ports in response to component addition and removal (recall the discussion in Section 3), and
3. dynamic class loading, either provided by the implementation platform (e.g., Java JVM and KVM) or implemented on top of it (e.g., by employing DLLs in C++ and EVC++).

Other objectives have been achieved by extending and optimizing the middleware core as discussed below.

4.2. Efficiency

Since PitM applications run on resource-constrained devices, with low amounts of memory (e.g., 256 KB of dynamic heap memory on the Palm Pilot) and processing power, we have performed several optimizations on the PitM middleware core. We observed that a large amount of dynamic system memory usage is a result of the exchange of messages among components and connectors. We minimized the required memory for message passing by replacing message queues at the component level, as required by the style, with a single, centralized message queue per each address space (i.e., Architecture object). Furthermore, read-only messages in the same address space are exchanged by reference, rather than by copy, to reduce the memory footprint. A pool of shepherd threads is kept ready to handle messages sent by any component in a given address space. The size of the thread pool is parameterized and, hence, adjustable. We further optimized memory usage by adopting a fixed-sized, circular array for storing messages. This reduces overall heap memory usage and places an upper bound on the required memory. At the same time, the concurrency management of the circular array slightly reduces the speed of processing by applying the producer-consumer algorithm to keep the message production under control, and supply shepherd threads with a constant stream of messages to process.

To process a message, a shepherd thread removes the message from the head of the queue. For communication that spans address spaces, the message is transported via a border connector to the recipient address space, and added to its message queue. For local communication, the shepherd thread is run through the connector attached to the sending component; the connector dispatches the message to relevant components using the same thread of execution for processing their handle methods (see Figure 2). If a recipient component generates further messages, they are added to the end of the message queue, and different threads are used for dispatching those messages to their intended recipients. An alternative design allows separate threads to be used for dispatching a message from the connector to each intended recipient component, thus increasing the parallelism in the architecture.

The optimizations described above have resulted in very light-weight middleware implementations that have shown reasonable performance. For illustration, we describe the results from one simple experiment used to measure the size and performance of the Java middleware implementation. The benchmarking application consisted of 50 identical components above and one component below a single connector. The application used a pool of 50 shepherd threads and a queue of 30 messages. 100,000 simple (parameter-less) messages were sent by the bottom component to all top components, resulting in 5,000,000 handled messages. The benchmark was performed on an Intel Pentium III 500 MHz processor with 256 MB of RAM running JDK 1.4 beta 2 on Microsoft Windows 2000. The time required to complete the exchange of messages was 4.7 seconds. Memory usage of the middleware, recorded at the time of architecture initialization, is 1.35 KB. The overhead of a “base” PitM component, without any application-specific methods or state, is 0.8 KB. Memory overhead of creating and sending a single message can be estimated using the following formula, obtained empirically:
message_overhead (in KB) = 0.46 + 0.02 * number_of_parameters

The formula assumes that the parameters do not contain complex objects, but may contain simple objects (e.g., Java Integer or String). Therefore, for example, the maximum memory overhead (assuming the message queue is full) induced by using the PitM middleware in our benchmark application described above is approximately:

\[1.35 + (51 \times 0.8) + (30 \times (0.46 + (0.02 \times 0))) = 56 \text{ KB}\]

4.3. Delivery Guarantees

The PitM middleware supports the following message delivery semantics: (1) at least once, (2) at most once, (3) best effort, and (4) exactly once [12]. Each message can be tagged with the delivery policy (best effort is the default). Communicating border connectors implement a “handshaking” protocol to ensure proper message delivery across address spaces. In order to maximize the efficiency of the delivery guarantee support in the same address space, we make use of PL exceptions to handle delivery guarantees (i.e., we assume that if no exception is raised, the message has been delivered). This optimization is possible since the middleware’s message passing capability in a single address space (including asynchronous message passing) is implemented using the underlying PL’s synchronous method calls.

This inherent synchronicity is also leveraged in our support for soft and hard real-time message delivery constraints [22]. The earliest-deadline-first algorithm is applied to messages in the queue in order to ensure hard real-time message delivery [22]. For soft real-time messages, the middleware provides support for specifying the delivery utility function and uses it to determine the best delivery strategy. We currently support soft real-time messages with linearly decreasing utility functions. Higher-order functions are too costly to compute on resource-constrained devices, therefore we expect that an approximation of such functions with a linear function will be applied.

4.4. Awareness

The PitM middleware supports architectures at two levels: application-level and meta-level. The role of components at the meta-level is to observe and/or facilitate different aspects of the execution, dynamic reconfiguration, and mobility of application-level components. At any point, the developer may add meta-level components to a (running) application. Meta-level components may be welded to specific application-level connectors to exercise control over a particular portion of the Architecture. Alternatively, meta-level components may remain unwelded and may instead exercise control over the entire Architecture object via its IScaffold interface. The structural and interaction characteristics of meta-level components are identical to those of application-level components, eliminating the need for their separate treatment in the middleware.

In support of this two-level architecture, the PitM middleware currently distinguishes among three types of messages. ApplicationData messages are used by application-level components to communicate during execution. The other two message types 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 ADL specifications needed to perform analyses of prospective PitM configurations (e.g., during deployment and dynamic reconfiguration).

We have extensively used meta-level components, called Admin Components, whose task is to exchange ComponentContent messages and facilitate the mobility of application-level components across devices, further discussed in Section 4.7. The Admin Components are very light weight: at instantiation time they occupy under 3 KB. Our middleware also supports component deployment and runtime upgrade by leveraging Admin Components and ComponentContent messages [28].

Another meta-level component is the Continuous Analysis component, which leverages ArchitecturalModel messages for analyzing the architectural descriptions during the application’s execution, assessing the validity of proposed runtime architectural changes, and possibly disallowing the changes. We have recently implemented several Continuous Analysis components that encapsulate different subsets of our DRADEL [26] environment. Depending on the kind of analysis it has to perform (e.g., topological constraint checking or component compliance checking), the sizes of the unoptimized Continuous Analysis components (each of which includes an OTS parser) range between 72 KB and 95 KB.2

---

2. In this sense, the measure represents the minimum message overhead. Note that the possible use of complex objects as message parameters is independent of the middleware, but is an application-level decision.
3. Figure A5 of the Appendix shows an application architecture with two meta-level components: Admin Component and Continuous Analysis component.
There are several other aspects of awareness that we are currently not supporting, such as self-optimization and self-adaptation. However, our experience thus far indicates that the PitM middleware provides a solid, flexible basis for supporting these aspects of awareness in the future.

4.5. Security
Secure communication in the PitM architectural middleware is achieved by using composite connectors [27] that encapsulate various security protocols. To this end, we have developed authentication, authorization, and encryption modules that may be added to an arbitrary PitM connector [25]. We have developed prototype implementations of several secure connectors. For example, we have implemented a connector that uses a symmetric encryption algorithm [2] to enable secure communication across mobile devices using their infrared (IR) ports. 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).

4.6. Scalability
Scalability of the PitM middleware can be characterized in the numbers of devices, components, connectors, threads, and messages it supports. The PitM style does not require centralization, such as having a single connector in the system (unlike e.g., CORBA [40]). For this reason, the number of devices supported by the PitM middleware is unlimited in principle. Flexible connectors in the middleware allow arbitrary deployment of an architecture. Assuming unlimited resources, an entire architecture can be deployed onto a single host. However, the same architecture can be deployed onto an arbitrary number of hosts. This is achieved by repeated horizontal or vertical splitting of connectors using the techniques described in [9]. In a highly degenerate case, this would result in some devices serving only as routers, without containing any components. It should be noted, however, that the deployment choice directly affects efficiency: the performance gain of using the centralized message queue (described in Section 4.2) is achieved only if the components are residing in the same address space.

Realistically, the number of components on a given device is limited and can be estimated using the following simple formula: \( n = (M - MS) / ACS \), where \( M \) is the available memory on the device, \( MS \) is the memory occupied by the middleware, and \( ACS \) is the average component size.\(^4\) Our middleware supports as many threads as the underlying platform supports. Finally, the number of messages supported by the PitM middleware is not limited by the middleware itself, but by the properties of the underlying hardware platform. This limit can be characterized by the following two parameters: (1) the maximum number of messages that can simultaneously be present in a system and (2) the rate of message delivery. The maximum number of messages is limited by the available memory on a given host (or set of hosts) and message size (recall Section 4.2), while the rate of message delivery depends on the CPU speed, the number of threads servicing the message queue, the ratio of message production to consumption by the components, and the network bandwidth for messages that traverse machine boundaries.

4.7. Mobility
Our support for mobility directly leverages the PitM middleware services. For example, if during the application’s execution a desired component- or system-level property is violated (as indicated by a meta-level component), the architecture may decide to reconfigure itself in the following manner: \(^5\)

1. If necessary, the migrant component is disconnected from its attached connectors using the Architecture object’s unweld and peerUnweld methods.
2. The sending Admin Component may unload the migrant component from the local subsystem using the Architecture object’s remove method, or it may access the compiled image of the migrant component from a local file.
3. The sending Admin Component serializes the migrant component (or the DLL containing the component, in the case of the middleware’s C++ implementation) into a byte stream, and sends it as a ComponentContent message via its local Border Connector to the destination devices.

---

\(^4\) Recall from Section 4.2 that the impact of MS and the middleware-induced portion of ACS on the device’s memory consumption is very low.

\(^5\) 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]). Application integrity is achieved in our approach by exchanging ArchitecturalModel messages along with ComponentContent messages and leveraging Continuous Analysis meta-level components.
4. Once received by the Border Connectors on the 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.

4.8. 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 centers on 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. The PitM middleware provides a meta-level Disconnection Controller component, which resides on each device that is either the source or the destination of component migration. Disconnection Controllers collaborate in estimating the best set of migrating components, and handle synchronization of components upon reconnection. When deciding which components should be migrated the DisconnectionController takes into account: the statefulness of candidate components for migration, frequencies of messages exchanged along the network link to be disconnected, dependencies of candidate components, type of disconnection (anticipated or sudden), required memory for loading candidate components, 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.

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.

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 other components in order to complete their task. Recall that PitM’s ADL associates a dependency tag 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 then be defined as follows:

\[
D = \frac{\sum_{i=1}^{N} d_i \times f_i}{N}
\]

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.

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. Components with a low (ideally, zero) value of \( S \) may be replicated onto the target host with fewer synchronization problems when the connection is restored. On the other hand, the replication of a component with a high \( S \) value requires the ability to synchronize, after the connection has been restored, any updates made to the different replicas of that component during disconnection.

State synchronization of component replicas can be achieved using the following algorithm.

1. Take the snapshot of a component’s state at the time of replication.
2. Log the timestamped invocations (received messages) on each replicated site.\(^{6}\)
3. Upon reconnection, unweld the original component and all its replicas from the rest of the architecture, and unload all the replicas from the application.

\(^{6}\) Carzaniga et. al.\(^{[6]}\) argue that it is reasonable to assume the global clock for message timestamps.
4. Queue any new messages intended for this component sent during the synchronization process (steps 5-8).
5. Revert the state of the component to the snapshot state.
6. Sort all the logged messages in the increasing timestamp order.
7. Invoke the component with the logged messages in the sorted order.
8. Reconnect the component to the rest of the architecture.
9. Invoke the component with the messages queued up during steps 5-8.

Alternatively, in order to eliminate the need for potentially costly state synchronization, the PitM ADL attaches a degraded mode indicator to each operation exported by a component (see Figure A3 in the Appendix). 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. The corresponding support in the PitM middleware 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.

Minimizing the Risk of Disconnection. The middleware’s DisconnectionController component tries to minimize the risk of disconnection by selecting components whose migration will give the application the best chance of performing normally on the disconnected device(s). In order to select the best component set for migration, for each candidate component the DisconnectionController component needs to have (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 is estimated using the following algorithm:
1. Benefit $\leftarrow 0$
2. Compare the ADL 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: Benefit $\leftarrow$ 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:
$$TAM = \min(M, t \times nb),$$
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 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 [8], 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. The middleware’s DisconnectionController component implements a simplification of the problem that is 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 [7]. The algorithm runs in $O(n \times TAM)$. 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)
The size of the DisconnectionController component without synchronization capabilities is about 9KB.

5. Related Work

Our work on PitM middleware has been primarily influenced by four research areas: architectural styles, middleware, code mobility, and disconnected operation. Architectural styles were discussed in the Introduction. Below we discuss the related approaches in the latter three areas.

Middleware. The research and use of middleware can be classified into six distinct generations on the basis of the achieved level of component reuse: (1) Module interconnection languages [10] enabled the reuse of components implemented in a single PL. (2) Remote procedure calls and platform-neutral data representations (e.g., [3,32]) enabled distribution and reuse across PLs. (3) Platform-neutral runtime environments and dynamic component loading (e.g., [16,24]) enabled dynamism and reuse across computing platforms. (4) Domain-specific and GUI frameworks (e.g., [17,31]) enabled reuse across applications. (5) Provision of infrastructure services such as naming, threading, reflection, persistence, and transaction management (e.g., [20,34]) introduced the possibility of reuse of architecture-level abstractions. (6) Reuse of architecture-level abstractions became an explicit focus of architectural style-based middleware (e.g., [35,37]). While it exhibits properties of middleware spanning several generations, the PitM middleware is most closely related to the sixth generation.

Code Mobility. A detailed overview of existing code mobility techniques is given in [15]. 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 4.7, our work primarily supports the code-on-demand technique, where each device is potentially a code repository.

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 discussed in Sections 4.7 and 4.8 supports both forms of mobility.

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 [14], and Ficus [19] 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-DA [38], which has recently added support for migrating components as computational elements, rather than as files, in response to disconnection. However, while FarGo-DA handles only anticipated disconnection, we have developed a more general, risk-based approach, which can be also used in cases of sudden disconnection.

6. Conclusions and Future Work

This paper has presented a description and evaluation of an architecture-based middleware that supports development of applications in PitM’s highly distributed, mobile, resource-constrained setting. Coupled with an explicit architectural style with its accompanying ADL and analysis tools, the middleware provides a set of services that enable seamless transition from an architecture to its implementation. Furthermore, the middleware provides a set of specialized capabilities required for PitM applications (light weight, distribution, mobility, application context awareness, asynchrony, support for disconnected operation, and so on). The PitM middleware has been implemented in several PLs, and has been successfully tested and evaluated on a number of applications to date. It has also been used as an educational tool in courses on software architectures and embedded systems at USC. Finally, the middleware is currently being used in several research projects with external collaborators.

7. Tables A1, A2, and A3 in the Appendix demonstrate estimations performed by the DisconnectionController component.
While our experience thus far has been very positive, a number of pertinent questions remain unexplored. Our future work will span issues such as extending context awareness to include self-reconfiguration (e.g., by observing the message traffic and co-locating the components that communicate frequently), extending the disconnected operation support (e.g., by exploiting PtM middleware’s explicit architectural topologies to re-route communication paths during disconnection), and investigating the trade-off between message compression to reduce network traffic and the computational overhead the compression introduces.

7. References
Appendix A: Example Application

To illustrate the concepts introduced in this paper, we use an application for distributed deployment of personnel, intended to deal with situations such as natural disasters, search-and-rescue efforts, and military crises. The specific instance of this application depicted in Figure A1 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 example configuration in Figure A1 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 controlling their own quadrant of the battlefield. The General can see a summarized view of the entire battlefield (shown) or detailed views of each quadrant. 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. Finally, the General can issue a “fight” command, resulting in a battle simulation that incrementally determines the likely winner.

The TDS application has provided an effective platform for evaluating a number of PitM middleware services. 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. The application is implemented in four dialects of two programming languages: Java JVM and KVM, C++ and 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, the

---

```
architecture TDS is {
  component_types {
    component PC_StrategyAnalyzer is extern {
        C:\spec\PC_StrategyAnalyzer.pitm;}
    ...
    component C_AvailableTroops is extern {
        C:\spec\C_AvailableTroops.pitm;}
    ...
  }
  architectural_topology {
    component_instances {
      pcStratAnalyzer : PC_StrategyAnalyzer;
      pcDataRepository : PC_DataRepository;
      cAvailableTroops : C_AvailableTroops;
      cAvailableTroops1 : C_AvailableTroops;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
    connections {
      TopBorderConnector { top pcStratAnalyzer; bottom pcDataRepository;}
      BottomBorderConn : RegularConn;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
  }
  peer_connections {
    PeerCon { side cAvailableTroops, cAvailableTroops1;}
  }
  peer_connectionInstances {
    PeerCon : PeerConn;
  }
  component PC_StrategyAnalyzer is {
    state {
      map: Map;
    }
    invariant {}
    interface {
      request ipGetMapInfo : MapInfo(): FILE;
      provide ipAnalyzeStrategy: AnalyzeStrategy(r:Map); dep: ipGetMapInfo; deg: allowed;
    }
    operations { request opGetMapInfo: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opAnalyze: {
      let m: Map;
      post (map = m);}
    }
    map { ipGetMapInfo -> opGetMapInfo();
      ipAnalyzeStrategy -> opAnalyze(r-> m);}
  }
  component PC_DataRepository is {
    state {
      map: Map;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  component C_AvailableTroops is {
    state {
      availableTroops: Troops;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  component C_AvailableTroops1 is {
    state {
      availableTroops1: Troops;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  architectural_topology {
    component_instances {
      pcStratAnalyzer : PC_StrategyAnalyzer;
      pcDataRepository : PC_DataRepository;
      cAvailableTroops : C_AvailableTroops;
      cAvailableTroops1 : C_AvailableTroops;
      BottomBorderConn : RegularConn;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
    connections {
      TopBorderConnector { top pcStratAnalyzer; bottom pcDataRepository;}
      BottomBorderConn : RegularConn;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
  }
  peer_connections {
    PeerCon { side cAvailableTroops, cAvailableTroops1;}
  }
  peer_connectionInstances {
    PeerCon : PeerConn;
  }
  component PC_StrategyAnalyzer is {
    state {
      map: Map;
    }
    invariant {}
    interface {
      request ipGetMapInfo : MapInfo(): FILE;
      provide ipAnalyzeStrategy: AnalyzeStrategy(r:Map); dep: ipGetMapInfo; deg: allowed;
    }
    operations { request opGetMapInfo: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opAnalyze: {
      let m: Map;
      post (map = m);}
    }
    map { ipGetMapInfo -> opGetMapInfo();
      ipAnalyzeStrategy -> opAnalyze(r-> m);}
  }
  component PC_DataRepository is {
    state {
      map: Map;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  component C_AvailableTroops is {
    state {
      availableTroops: Troops;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  component C_AvailableTroops1 is {
    state {
      availableTroops1: Troops;
    }
    invariant {}
    interface {
      request ipGetTroops : Troops(): FILE;
      provide ipGetTroops: Troops();
    }
    operations { request opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
      provide opGetTroops: {
      let PH: STATE_VARIABLE;
      post (result = PH);}
    }
  }
  architectural_topology {
    component_instances {
      pcStratAnalyzer : PC_StrategyAnalyzer;
      pcDataRepository : PC_DataRepository;
      cAvailableTroops : C_AvailableTroops;
      cAvailableTroops1 : C_AvailableTroops;
      BottomBorderConn : RegularConn;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
    connections {
      TopBorderConnector { top pcStratAnalyzer; bottom pcDataRepository;}
      BottomBorderConn : RegularConn;
    }
    peer_connector_instances {
      PeerCon : PeerConn;
    }
    peer_connections {
      PeerCon { side cAvailableTroops, cAvailableTroops1;}
    }
  }
  peer_connections {
    PeerCon { side cAvailableTroops, cAvailableTroops1;}
  }
  peer_connectionInstances {
    PeerCon : PeerConn;
  }
}
```
Continuous Analysis and Disconnection Controller 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 A1 and A2, sixteen software components deployed across the five devices interact via fifteen software connectors.

Figure A2 shows the deployment architecture of the TDS application in the PitM style. The deployment environment is described in [28]. Left side of Figure A3 shows a partial architectural specification of the TDS application in the PitM ADL. Specifications of individual components are located in separate files, as denoted by the extern keyword. Partial specification of one component is given on the right side of Figure A3.

Figure A4 shows a subset of the Java code for the TDS application that uses our middleware in the manner depicted in Figure A5. The TDS architecture’s main method calls methods of its superclass for instantiating components and connectors and composing (welding) them into a configuration. Figure A4 also demonstrates message-based communication between two components. DisplayManager creates and sends a request

```
public void handle(Request r) {
    if (r.equals(this.analyzeStrategy)) {
        Notification n = new Notification(this.displayResults);
        n.addParameter("outcome", outcome);
        send(n);
    }
}
```

Figure A5. Layered construction of an application using the PitM middleware. The application is distributed across five devices, each of which is running the middleware. 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).

Table A1: Message traffic

<table>
<thead>
<tr>
<th>Message</th>
<th>Processing Component</th>
<th>( f_i )</th>
<th>( d_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>AnalyzeStrategy</td>
<td>Strategy Analyzer</td>
<td>0.06</td>
<td>2</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 A2: Candidate components

<table>
<thead>
<tr>
<th>Strategy Component</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 A3: 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 Deployment Advisor</td>
</tr>
<tr>
<td>1 s</td>
<td>30 KB/s</td>
<td>40 KB</td>
<td>Strategy Analyzer War Manager Deployment Advisor</td>
</tr>
</tbody>
</table>

Tables A1, A2, and A3 illustrate the estimations made by the meta-level Disconnection Controller component described in Section 4.8. Let us assume that General’s Palm (shown in Figures A1, A2, and A5) is going to get disconnected within a given period, and that we know how much dynamic memory remains unused on that device. 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 A1 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 A2.

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 A3.